EPTCS 385

Proceedings of the

39th International Conference on
Logic Programming

Imperial College London, UK, 9th July 2023 - 15th July 2023

Edited by: Enrico Pontelli, Stefania Costantini, Carmine Dodaro, Sarah Gaggl, Roberta Calegari, Artur D'Avila Garcez, Francesco Fabiano, Alessandra Mileo, Alessandra Russo and Francesca Toni

Published: 12th September 2023
DOI: 10.4204/EPTCS.385
ISSN: 2075-2180
Open Publishing Association

Table of Contents

EPTCS Volume of Technical Communications presented at ICLP 2023

Editorial

Stefania Costantini
University of L'Aquila, Italy

Enrico Pontelli
New Mexico State University, USA

Roberta Calegari
University of Bologna, Italy

Carmine Dodaro
University of Calabria, Italy

Francesco Fabiano
University of Udine, Italy

Sarah Gaggl
TU Dresden, Germany

Artur Garcez
City University of London, UK

Alessandra Mileo
DCU, Ireland

Alessandra Russo
Imperial College London, UK

Francesca Toni
Imperial College London, UK

This volume contains the Technical Communications presented at the 39th International Conference on Logic Programming (ICLP 2023), held at Imperial College London, UK from July 9 to July 15, 2023. Technical Communications included here concern the Main Track, the Doctoral Consortium, the Application and Systems/Demo track, the Recently Published Research Track, the Thematic Tracks on Logic Programming and Machine Learning, and Logic Programming and Explainability, Ethics, and Trustworthiness.

The International Conference on Logic Programming (ICLP) is the premiere international scientific venue for presenting, discussing, and disseminating fundamental and applied research in the field of logic programming and in related interdisciplinary areas. The conference was launched in 1982, with its first edition in Marseille, France, and continues to serve the broad international logic programming community.

ICLP 2023 was a special edition of the conference as it represented the closing event of the Year of Prolog—a year-long celebration of the 50th anniversary of the introduction of the Prolog programming language. The conference included a dedicated session to discuss the achievements accomplished throughout the year and the assignment of the 2023 Alan Colmerauer Prize, as well as an event on ”Teaching Prolog: the Next 50 Years”.

ICLP 2023 also included an Industry Event, with the participation of representatives of several for-profit organizations engaged in the use of logic programming technologies. The event consisted of a panel discussion on the role of symbolic AI in today's AI industrial landscape. The Conference featured Invited Speakers: Marco Maratea (University of Calabria, Italy), presenting on applications of answer set programming; Jan Wielemaker (Centrum Wiskunde & Informatica, The Netherlands), presenting on the rise, fall, and future of Prolog; Keith Stenning (The University of Edinburgh, Scotland) for the special track on Logic Programming and Machine Learning; Gopal Gupta (University of Texas at Dallas, USA) for the special track on Logic Programming and Explainability, Ethics and Trustworthiness; Mehul Bhatt (Orebro University, Sweden), tutorialist speaking on Cognitive Vision; and Joseph Townsend (Fujitsu Research of Europe, UK), tutorialist speaking on medical applications of neuro symbolic AI.

The Technical Communications (TCs) appearing in this volume are papers of great quality but presenting results that are more preliminary and not yet ready for the standards of journal publication. The

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. v–vii, doi:10.4204/EPTCS.385.0

TCs also include accepted short papers, summaries of system presentations, and accepted extended abstracts from the Recently Published Research Track. In particular: 26 papers have been accepted for regular TC publication; 5 papers have been accepted as short papers in the TC category; 8 papers have been accepted as system demonstrations. In addition, we received and accepted 11 abstracts for the Recently Published Research track. All of them received an allocation of time in the conference program for presentation. In order to foster the engagement of students and their precious contribution to the research community, ICLP also hosted a doctoral consortium with 9 accepted submissions, two of which by women. A mentoring lunch event with a dedicated panel session were specifically held in order to facilitate exchange of ideas and networking.

ICLP 2023 gratefully received generous funding from the following sponsors.

- Platinum Sponsors: Artificial Intelligence Journal; the National Science Foundation, USA.
- Gold Sponsors: SECCAI, School of Embedded Composite Artificial Intelligence, TU Dresden, and Leipzig University.
- Silver Sponsors: ILASP, DLVSystems.
- Bronze Sponsors: EurAI, European Association for Artificial Intelligence, Google DeepMind, Imperial College London and Imperial-X, LPA, New Mexico State University, Potassco Solutions.

The organizers of ICLP 2023 are listed below.

- General Chairs: Alessandra Russo and Francesca Toni, Imperial College London, UK.
- Program Chairs: Stefania Costantini, University of L'Aquila, Italy, and Enrico Pontelli, New Mexico State University, USA.
- Organizing and Publicity Chairs: Guilherme Paulino-Passos and Alex Spies, Imperial College London, UK.
- Workshop Chair: Wolfgang Faber, University of Klagenfurt, Austria.
- Thematic Tracks Chairs:
  - LP and Explainability, Ethics and Trustworthiness: Roberta Calegari, University of Bologna, Italy
  - LP and Machine Learning: Artur Garcez, City University of London, UK.
- Applications, Systems & Demos Chair: Carmine Dodaro, University of Calabria, Italy.
- Recently Published Research Chair: Sarah Gaggl, TU Dresden, Germany.
- Tutorials Chairs: Agostino Dovier, University of Udine, Italy, and Esra Erdem, Sabanci University, Turkey.
- Doctoral Consortium Chairs: Francesco Fabiano, University of Udine, Italy, and Alessandra Mileo, DCU, Ireland.
- Programming Contest Chairs: Martin Gebser, Graz University of Technology, Austria, and Mario Alviano, University of Calabria, Italy.
- Summer School Chairs: Simona Perri, University of Calabria, Italy, and Bart Bogaerts, Vrije Universiteit Brussel, Belgium.
- Women@ICLP Chair: Daniele Nantes Sobrinho, Universidade de Brasília, Brazil, and Imperial College London, UK.

- Administrative Support: Teresa Carbajo Garcia, Julitta Iranek-Osmecka, and Teresa Ng, Imperial College London, UK.

We heartily thank the ICLP program committee members and additional reviewers for their valuable work. We express our gratitude to Thomas Eiter, the current President of the Association of Logic Programming (ALP), Marco Gavanelli, in the role of conference coordinator for ALP, and all the members of the ALP Executive Committee. We would like to recognize the outstanding work of the editorial team of EPTCS for their support Last but not least, we are deeply indebted to all the authors of any submission to ICLP 2023, we thank you for choosing ICLP as the venue for sharing your ideas, your hard work, and your expertise.

ABA Learning via ASP

Emanuele De Angelis
IASI-CNR
Rome, Italy
emanuele.deangelis@iasi.cnr.it
0000-0002-7319-8439

Maurizio Proietti
IASI-CNR
Rome, Italy
maurizio.proietti@iasi.cnr.it
0000-0003-3835-4931

Francesca Toni
Department of Computing
Imperial College London, UK
ft@ic.ac.uk
0000-0001-8194-1459

Recently, ABA Learning has been proposed as a form of symbolic machine learning for drawing Assumption-Based Argumentation frameworks from background knowledge and positive and negative examples. We propose a novel method for implementing ABA Learning using Answer Set Programming as a way to help guide Rote Learning and generalisation in ABA Learning.

1 Introduction

Recently, *ABA Learning* has been proposed [12] as a methodology for learning Assumption-Based Argumentation (ABA) frameworks [1, 2] from a background knowledge, in the form of an ABA framework, and positive and negative examples, in the form of sentences in the language of the background knowledge. The goal of ABA Learning is to build a larger ABA framework than the background knowledge from which arguments for all positive examples can be "accepted" and no arguments for any of the negative examples can be "accepted". In this paper, for a specific form of ABA frameworks corresponding to logic programs [1], we focus on a specific form of "acceptance", given by cautious (or sceptical) reasoning under the argumentation semantics of stable extensions [1, 2]. We then leverage on the well known correspondence between stable extensions in the logic programming instance of ABA and answer set programs [5] to outline a novel implementation strategy for the form of ABA Learning we consider, pointing out along the way restrictions on ABA Learning enabling the use of Answer Set Programming (ASP).

Related Work Our strategy for ABA Learning differs from other works learning argumentation frameworks, e.g. [3, 11], in that it learns a different type of argumentation frameworks and it uses ASP. ABA can be seen as performing abductive reasoning (as assumptions are hypotheses open for debate). Other approaches combine abductive and inductive learning [13], but they do not learn ABA frameworks. Some approaches learn abductive logic programs [6], which rely upon assumptions, like ABA frameworks. A formal comparison with these methods is left for future work. ABA captures several non-monotonic reasoning formalisms, thus ABA Learning is related to other methods learning non-monotonic formalisms. Some of these methods, e.g. [7, 14, 17], do not make use of ASP. Some others, e.g. [8, 15, 16], do. While our use of ASP to help guide some aspects of ABA Learning (e.g. its Rote Learning transformation rule) is unique, a formal and empirical comparison with these methods is left for future work.

2 Background

ASP In this paper we use *answer set programs* (ASPs) [5] consisting of rules of the form

$$\mathtt{p} \;\mathtt{:-}\; \mathtt{q}_1,\ldots,\mathtt{q}_\mathtt{k},\ \mathtt{not}\ \mathtt{q}_{\mathtt{k}+1},\ldots,\ \mathtt{not}\ \mathtt{q}_\mathtt{n} \qquad \text{or} \qquad \mathtt{:-}\; \mathtt{q}_1,\ldots,\mathtt{q}_\mathtt{k},\ \mathtt{not}\ \mathtt{q}_{\mathtt{k}+1},\ldots,\ \mathtt{not}\ \mathtt{q}_\mathtt{m}$$

where $\mathtt{p}$, $\mathtt{q}_1, \ldots, \mathtt{q}_\mathtt{n}$, $\mathtt{q}_1, \ldots, \mathtt{q}_\mathtt{m}$ are atoms, $\mathtt{k} \geq 0$, $\mathtt{n} \geq 0$, $\mathtt{m} \geq 1$, and `not` denotes negation as failure.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 1–8, doi:10.4204/EPTCS.385.1

Given any ASP program P, by $ans(P)$, called *answer set* of P, we denote the set of ground atoms assigned to P by the answer set semantics. Let $ans_1(P),\dots,ans_l(P)$ be the answer sets of P, for $l \geq 1$ (if $l = 0$, then P is *unsatisfiable*). By $\mathscr{C}(P) = \bigcap_i ans_i(P)$, we denote the set of *cautious* consequences of P.

ABA An *ABA framework* (as originally proposed in [1], but presented here following recent accounts in [4, 18] and [2]) is a tuple $\langle \mathscr{L}, \mathscr{R}, \mathscr{A}, \overline{} \rangle$ such that

- $\langle \mathscr{L}, \mathscr{R} \rangle$ is a deductive system, where $\mathscr{L}$ is a *language* and $\mathscr{R}$ is a set of *(inference) rules* of the form $s_0 \leftarrow s_1,\dots,s_m$ ($m \geq 0, s_i \in \mathscr{L}$, for $1 \leq i \leq m$);
- $\mathscr{A} \subseteq \mathscr{L}$ is a (non-empty) set of *assumptions*;[1]
- $\overline{}$ is a total mapping from $\mathscr{A}$ into $\mathscr{L}$, where $\overline{a}$ is the *contrary* of a, for $a \in \mathscr{A}$.

Given a rule $s_0 \leftarrow s_1,\dots,s_m$, s_0 is the *head* and $s_1,\dots,s_m$ is the *body*; if $m = 0$ then the body is said to be *empty* (represented as $s_0 \leftarrow$ or $s_0 \leftarrow true$) and the rule is called a *fact*. If assumptions are not heads of rules then the ABA framework is called *flat*. In this paper we focus on flat ABA frameworks. Elements of $\mathscr{L}$ can be any sentences, but in this paper we focus on (flat) ABA frameworks where $\mathscr{L}$ is a set of ground atoms. However, in the spirit of logic programming, we will use *schemata* for rules, assumptions and contraries, using variables to represent compactly all instances over some underlying universe.

Example 1. *The following is a flat ABA framework with $\mathscr{L}$ a set of atoms.*

$$\begin{aligned}
\mathscr{R} = \quad & \{innocent(X) \leftarrow person(X), not\_guilty(X), \quad guilty(X) \leftarrow witness\_con(X,Y), \\
& person(mary) \leftarrow, \quad person(alex) \leftarrow, \quad witness\_con(mary, alex) \leftarrow\} \\
\mathscr{L} = \quad & \{innocent(X), person(X), not\_guilty(X), guilty(X), witness\_con(X,Y) | X,Y \in \{mary, alex\}\} \\
\mathscr{A} = \quad & \{not\_guilty(mary)\} \quad \text{where} \quad \overline{not\_guilty(mary)} = guilty(mary).
\end{aligned}$$

The semantics of flat ABA frameworks is given in terms of "acceptable" extensions, i.e. sets of *arguments* able to "defend" themselves against *attacks*, in some sense, as determined by the chosen semantics. Intuitively, arguments are deductions of claims using rules and supported by assumptions, and attacks are directed at the assumptions in the support of arguments. For illustration, in the case of Example 1, there are, amongst others, arguments $\{not\_guilty(mary)\} \vdash_{\{\rho_1,\phi_1\}} innocent(mary)$ (with ρ_1 the first rule in $\mathscr{R}$ and ϕ_1 the fact $person(mary) \leftarrow$ in $\mathscr{R}$) and $\emptyset \vdash_{\{\rho_2,\phi_3\}} guilty(mary)$ (with ρ_2 the second rule in $\mathscr{R}$ and ϕ_3 the fact $witness\_con(mary, alex) \leftarrow$ in $\mathscr{R}$), with the latter argument attacking the former.

Given a flat ABA framework $\langle \mathscr{L}, \mathscr{R}, \mathscr{A}, \overline{} \rangle$, let $Args$ be the set of all arguments and $Att = \{(\alpha,\beta) \in Args \times Args \mid \alpha \text{ attacks } \beta\}$. Then, the notion of "acceptable" extensions we will focus on in this paper is as follows: $\Delta \subseteq Args$ is a *stable extension* iff (i) $\nexists \alpha,\beta \in \Delta$ such that $(\alpha,\beta) \in Att$ (i.e. Δ is *conflict-free*) and (ii) $\forall \beta \in Args \setminus \Delta, \exists \alpha \in \Delta$ such that $(\alpha,\beta) \in Att$ (i.e. Δ "attacks" all arguments it does not contain, thus pre-emptively "defending" itself against potential attacks). We will consider the *cautious* (a.k.a. *sceptical*) consequences of (flat) ABA frameworks $\langle \mathscr{L}, \mathscr{R}, \mathscr{A}, \overline{} \rangle$, i.e. the sets of sentences in $\mathscr{L}$ that are claims of arguments in all stable extensions for $\langle \mathscr{L}, \mathscr{R}, \mathscr{A}, \overline{} \rangle$. The cautious consequences of the ABA framework in Example 1 include $guilty(mary)$ and $innocent(alex)$.

Here we will work with ABA frameworks admitting at least one stable extension. Also, without loss of generality, we will leave the language component of all ABA frameworks implicit, and use, e.g., $\langle \mathscr{R}, \mathscr{A}, \overline{} \rangle$ to stand for $\langle \mathscr{L}, \mathscr{R}, \mathscr{A}, \overline{} \rangle$ where $\mathscr{L}$ is the set of all sentences in $\mathscr{R}$, $\mathscr{A}$ and in the range of $\overline{}$. We will also use $\langle \mathscr{R}, \mathscr{A}, \overline{} \rangle \models s$ to indicate that $s \in \mathscr{L}$ is a cautious consequence of $\langle \mathscr{R}, \mathscr{A}, \overline{} \rangle$.

---

[1] The non-emptiness requirement can always be satisfied by including in $\mathscr{A}$ a *bogus assumption*, with its own contrary, neither occurring elsewhere in the ABA framework. For conciseness, we will leave this assumption and its contrary implicit.

3 Preliminaries: Cautious ABA Learning under Stable Extensions

Here we recap the instance of the ABA Learning method proposed in [12] that we focus on implementing using ASP in this paper, while stating restrictions on ABA Learning required by the implementation.

In [12], the *background knowledge* is *any* ABA framework $\langle \mathcal{R}, \mathcal{A}, \overline{} \rangle$. Here, we assume that (i) it is restricted so that each assumptions occurs in the body of at most one rule schema in $\mathcal{R}$, (ii) for each non-ground $\alpha(X) \in \mathcal{A}$ in the body of any $\rho \in \mathcal{R}$, for X a tuple of variables, for each variable X' in X, there is at least one *non-assumption* (in $\mathcal{L} \setminus \mathcal{A}$) $p(Y)$ in the body of ρ with X' in Y, and (iii) each fact in $\mathcal{R}$ is ground. Restriction (i) is without loss of generality; the other two derive from the use of schemata.

In [12], *positive/negative examples* are ground atoms of the form $p(c)$, for p a predicate with arity $n \geq 0$ and c a tuple of n constants. Here, we impose that examples are non-assumptions (in the background knowledge $\langle \mathcal{R}, \mathcal{A}, \overline{} \rangle$). So, for $\mathcal{L}$ as in Example 1, *not_guilty* cannot appear in examples (but contraries, e.g. $guilty(mary)$, can be examples). The exclusion of assumptions from examples is derived from the flatness restriction. We also assume that for each example $p(c)$ and c' in c, $\exists q(d) \leftarrow \in \mathcal{R}$ such that c' is in d. We impose the same restriction on constants c' anywhere in the background knowledge.

Given background knowledge $\langle \mathcal{R}, \mathcal{A}, \overline{} \rangle$, positive examples $\mathcal{E}^+$ and negative examples $\mathcal{E}^-$ with $\mathcal{E}^+ \cap \mathcal{E}^- = \emptyset$, the *goal of ABA Learning* is to construct (a flat ABA framework) $\langle \mathcal{R}', \mathcal{A}', \overline{}' \rangle$ such that $\mathcal{R} \subseteq \mathcal{R}'$, $\mathcal{A} \subseteq \mathcal{A}'$, and $\overline{\alpha}' = \overline{\alpha}$ for all $\alpha \in \mathcal{A}$, so that $\langle \mathcal{R}', \mathcal{A}', \overline{}' \rangle$ *entails* $\langle \mathcal{E}^+, \mathcal{E}^- \rangle$, that is:
(*Existence*) $\langle \mathcal{R}', \mathcal{A}', \overline{}' \rangle$ admits at least one stable extension,
(*Completeness*) for all $e \in \mathcal{E}^+$, $\langle \mathcal{R}', \mathcal{A}', \overline{}' \rangle \models e$, and
(*Consistency*) for all $e \in \mathcal{E}^-$, $\langle \mathcal{R}', \mathcal{A}', \overline{}' \rangle \not\models e$.

$\langle \mathcal{R}', \mathcal{A}', \overline{}' \rangle$ is called a *solution* of the ABA Learning problem $(\langle \mathcal{R}, \mathcal{A}, \overline{} \rangle, \langle \mathcal{E}^+, \mathcal{E}^- \rangle)$. The second condition implies, when $\mathcal{E}^+ \neq \emptyset$, that the set of cautious consequences of a solution is non-empty.

In this paper we strive towards what we may call *intensional solutions*, namely such that $\mathcal{R}' \setminus \mathcal{R}$ comprises of *intentional rules* (i.e. non-ground rule schemata), to avoid or limit "lazy" learning of facts covering the positive examples alone and none of the negative examples, leading to poor generalisation.

Example 2. *Consider the background knowledge:*

$$
\begin{array}{rl}
\mathcal{R} = & \{innocent(X) \leftarrow defendant(X), not\_guilty(X), \\
& witness\_con(mary, alex) \leftarrow, \quad witness\_con(david, carol) \leftarrow, \quad witness\_con(john, carol) \leftarrow, \\
& defendant(mary) \leftarrow, defendant(david) \leftarrow, defendant(john) \leftarrow, liar(alex) \leftarrow, away(bob) \leftarrow, \\
& person(alex) \leftarrow, \quad person(bob) \leftarrow, \quad person(carol) \leftarrow, \\
& person(mary) \leftarrow, \quad person(david) \leftarrow, \quad person(john) \leftarrow\} \\
\mathcal{A} = & \{not\_guilty(X) \mid X \in \{mary, david, john\}\} \quad \text{where} \quad \overline{not\_guilty(X)} = guilty(X)
\end{array}
$$

and examples $\mathcal{E}^+ = \{innocent(mary),\ innocent(bob)\}$, $\mathcal{E}^- = \{innocent(david),\ innocent(john)\}$. *Then, solutions of* $(\langle \mathcal{R}, \mathcal{A}, \overline{} \rangle, \langle \mathcal{E}^+, \mathcal{E}^- \rangle)$ *include ABA frameworks with* $\mathcal{R}'_1$ *and* $\mathcal{R}'_2$ *whereby*
$\mathcal{R}'_1 \setminus \mathcal{R} = \{innocent(bob) \leftarrow, \quad guilty(david) \leftarrow, \quad guilty(john) \leftarrow\}$ *and*
$\mathcal{R}'_2 \setminus \mathcal{R} = \{innocent(X) \leftarrow away(X), \quad guilty(X) \leftarrow witness\_con(X,Y), person(Y), a(X,Y),$
$c\_a(X,Y) \leftarrow defendant(X), liar(Y)\}.$
The latter can be deemed to be intensional, whereas the former is not.

In the remainder, as in [12], we will represent ground facts $p(t) \leftarrow$ as $p(X) \leftarrow X = t$. We will also use $vars(A)$ to denote the set of all variables occurring in assumption, rule or rule body A.

4 Learning ABA Frameworks via Transformation Rules and ASP Solving

In order to learn ABA frameworks from examples, we follow the approach based on *transformation rules* presented in [12], but only consider a subset of those rules: *Rote Learning*, *Folding*, *Assumption Introduction*, and (a special case of) *Subsumption* (thus ignoring *Equality Removal*). Some rules (Folding and Subsumption) are borrowed from logic program transformation [10], while others (Rote Learning and Assumption Introduction) are specific for ABA. Given an ABA framework $\langle \mathcal{R},\mathcal{A},\overline{}\rangle$, a transformation rule constructs a new ABA framework $\langle \mathcal{R}',\mathcal{A}',\overline{}'\rangle$ (in the remainder, we will mention explicitly only the modified components). The application of the transformation rules is guided by the *ASP-ABALearn* strategy (see Figure 1), a variant of the strategy in [12] amenable to be implemented via an ASP solver, towards the goal of deriving an intensional solution of the given ABA Learning problem.

```
Strategy ASP-ABAlearn. Input: An ABA Learning problem $(\langle \mathcal{R},\mathcal{A},\overline{\phantom{x}}\rangle,\langle \mathcal{E}^+,\mathcal{E}^-\rangle)$;
  $RoLe(\langle \mathcal{R},\mathcal{A},\overline{\phantom{x}}\rangle,\langle \mathcal{E}^+,\mathcal{E}^-\rangle)$;   $GEN(\langle \mathcal{R},\mathcal{A},\overline{\phantom{x}}\rangle,\langle \mathcal{E}^+,\mathcal{E}^-\rangle)$.

Procedure $RoLe(\langle \mathcal{R},\mathcal{A},\overline{\phantom{x}}\rangle,\langle \mathcal{E}^+,\mathcal{E}^-\rangle)$.
$P := ASP^*(\langle \mathcal{R},\mathcal{A},\overline{\phantom{x}}\rangle,\langle \mathcal{E}^+,\mathcal{E}^-\rangle,\mathcal{A})$;
if $P$ is unsatisfiable then fail;
for all $\mathtt{c\_}\alpha(\mathtt{t}) \in \mathscr{C}(P) \setminus \mathscr{C}(ASP(\langle \mathcal{R},\mathcal{A},\overline{\phantom{x}}\rangle))$, where $c\_\alpha(t) = \overline{\alpha(t)}$ for some $\alpha(t) \in \mathcal{A}$ do
  apply Rote Learning and get $\mathcal{R} := \mathcal{R} \cup \{c\_\alpha(X) \leftarrow X = t\}$;
for all $p(u) \in \mathcal{E}^+$ such that $\mathtt{p(u)} \notin \mathscr{C}(ASP(\langle \mathcal{R},\mathcal{A},\overline{\phantom{x}}\rangle))$ do
  apply Rote Learning and get $\mathcal{R} := \mathcal{R} \cup \{p(X) \leftarrow X = u\}$;
if $\langle \mathcal{R},\mathcal{A},\overline{\phantom{x}}\rangle$ does not entail $\langle \mathcal{E}^+,\mathcal{E}^-\rangle$ then fail;

Procedure $GEN(\langle \mathcal{R},\mathcal{A},\overline{\phantom{x}}\rangle,\langle \mathcal{E}^+,\mathcal{E}^-\rangle)$.
while there exists a non-intensional rule $\rho_1 \in \mathcal{R}_{learnt}$ do
  // Folding:
  $\rho_2 := fold\text{-}all(\rho_1)$;   $\mathcal{R} := (\mathcal{R} \setminus \{\rho_1\}) \cup \{\rho_2\}$;
  // Assumption Introduction:
  if $\langle \mathcal{R},\mathcal{A},\overline{\phantom{x}}\rangle$ does not entail $\langle \mathcal{E}^+,\mathcal{E}^-\rangle$ then
    let $\rho_2$ be of the form $H \leftarrow B$; apply Assumption Introduction and get $\rho_3$: $H \leftarrow B, \alpha(X)$
    where $\alpha$ is a new predicate symbol and $X = vars(H \leftarrow B)$;
    $\mathcal{R} := (\mathcal{R} \setminus \{\rho_2\}) \cup \{\rho_3\}$;   $\mathcal{A} := \mathcal{A} \cup \{\alpha(X)\}$, with: $\overline{\alpha(X)} = c\_\alpha(X)$;
    // Rote Learning:
    for all $\mathtt{c\_}\alpha(\mathtt{t}) \in \mathscr{C}(ASP^+(\langle \mathcal{R},\mathcal{A},\overline{\phantom{x}}\rangle,\langle \mathcal{E}^+,\mathcal{E}^-\rangle,\{\alpha(X)\}))$ do
      apply Rote Learning and get $\mathcal{R} := \mathcal{R} \cup \{c\_\alpha(X) \leftarrow X = t\}$;
    if $\langle \mathcal{R},\mathcal{A},\overline{\phantom{x}}\rangle$ does not entail $\langle \mathcal{E}^+,\mathcal{E}^-\rangle$ then fail;
    // Subsumption:
    for all $\rho$: $p(X) \leftarrow X = t \in \mathcal{R}_{learnt}$ do
      if $\mathtt{p(t)} \in \mathscr{C}(ASP(\langle \mathcal{R} \setminus \{\rho\},\mathcal{A},\overline{\phantom{x}}\rangle))$ then apply Subsumption and delete $\rho$: $\mathcal{R} := \mathcal{R} \setminus \{\rho\}$.
```

Figure 1: *ASP-ABAlearn* strategy. By $\mathcal{R}_{learnt}$ we denote the subset of the rules in $\mathcal{R}$ that do not belong to the original background knowledge. The function *fold-all*, given a non-intensional rule ρ_1, returns an intensional rule ρ_2 obtained by applying once or more times the Folding rule to ρ_1 using rules in $\mathcal{R}$.

(a) Each rule in $\mathcal{R}$ is rewritten in the ASP syntax (e.g. $innocent(X) \leftarrow defendant(X), not\_guilty(X)$ in Ex. 2 becomes `innocent(X) :- defendant(X), not_guilty(X).`). In the following, we use the teletype font for the ASP translation of the rules.

(b.1) Each $\alpha_i \in \mathcal{A}$ occurring in the body of $H \leftarrow B, \alpha_1, \ldots, \alpha_n \in \mathcal{R}$ is encoded as the following ASP rule, where `c_`α_i is an ASP atom encoding $\overline{\alpha_i}$, and $vars(\alpha_{\mathtt{i}}) \subseteq \mathtt{X}$:

$\alpha_{\mathtt{i}}$ `:- dom(X), not c_`$\alpha_{\mathtt{i}}$`.`

(b.2) Each $\alpha_i \in \mathcal{K}$ occurring in the body of $H \leftarrow B, \alpha_1, \ldots, \alpha_n \in \mathcal{R}$ is encoded as the following pair of ASP rules:

`c_`$\alpha_{\mathtt{i}}$ `:- dom(X), not` $\alpha_{\mathtt{i}}$`.` `:-` $\alpha_{\mathtt{i}}$`, c_`$\alpha_{\mathtt{i}}$`.`

(c) Each $e \in \mathcal{E}^+$ is encoded as the ASP rule `:- not e.`

(d) Each $e \in \mathcal{E}^-$ is encoded as the ASP rule `:- e.`

(e) Each atom $p(X) \in \mathcal{E}^+ \cup \mathcal{E}^-$ such that $p(X)$ is not the contrary of any assumption in $\mathcal{A}$ is additionally encoded as the following triple of ASP rules, where `neg_p` is a new predicate name:

`p(X) :- dom(X), not neg_p(X).` `neg_p(X) :- dom(X), not p(X).` `:- p(X), neg_p(X).`

Figure 2: ASP-encodings for a given ABA Learning problem $(\langle \mathcal{R}, \mathcal{A}, \overline{} \rangle, \langle \mathcal{E}^+, \mathcal{E}^- \rangle)$ and a set $\mathcal{K} \subseteq \mathcal{A}$ of assumptions. Here, `dom` is chosen so that `dom(X)` holds for all `X` encoding tuples of constants of $\mathcal{L}$. Note that, without loss of generality, in (b.1)-(b.2) we can assume $vars(\alpha_1, \ldots, \alpha_n) \subseteq vars(B)$. So, we could replace `dom(X)` by any subset of B that contains all variables of $\alpha_{\mathtt{i}}$ and `dom` may already occur in $\langle \mathcal{R}, \mathcal{A}, \overline{} \rangle$ (this is an optimisation, as fewer ground instances of the rule may be given by the ASP solver). Also, in (e) `dom` may already occur in $\langle \mathcal{R}, \mathcal{A}, \overline{} \rangle$.

The *ASP-ABAlearn* strategy is the composition of two procedures: (1) *RoLe*, which has the goal of adding suitable facts to the initial background knowledge $\langle \mathcal{R}, \mathcal{A}, \overline{} \rangle$ so that the new ABA framework $\langle \mathcal{R}', \mathcal{A}', \overline{}' \rangle$ is a (non-intensional) solution of the learning problem $(\langle \mathcal{R}, \mathcal{A}, \overline{} \rangle, \langle \mathcal{E}^+, \mathcal{E}^- \rangle)$ given in input, and (2) *GEN*, which has the objective of transforming $\langle \mathcal{R}', \mathcal{A}', \overline{}' \rangle$ into an intensional solution. In general, it is not obvious which fact should be added to $\mathcal{R}$ in *RoLe* and how to generalise a non-intensional solution to obtain an intentional one in *GEN*. For these purposes, we use various encodings into ASP defined in Fig.2 to obtain the following sets of ASP rules:

- The set of ASP rules at points (a) and (b.1) of Fig. 2 is denoted by $ASP(\langle \mathcal{R}, \mathcal{A}, \overline{} \rangle)$. For a claim $s \in \mathcal{L}$, we can check that $\langle \mathcal{R}, \mathcal{A}, \overline{} \rangle \models s$ (i.e., s is a cautious consequence of $\langle \mathcal{R}, \mathcal{A}, \overline{} \rangle$ under stable extensions) by checking that $\mathtt{s} \in \mathcal{C}(ASP(\langle \mathcal{R}, \mathcal{A}, \overline{} \rangle))$.
- The set of ASP rules at points (a)–(d) is denoted $ASP^+(\langle \mathcal{R}, \mathcal{A}, \overline{} \rangle, \langle \mathcal{E}^+, \mathcal{E}^- \rangle, \mathcal{K})$. By computing $\mathcal{C}(ASP^+(\langle \mathcal{R}, \mathcal{A}, \overline{} \rangle, \langle \mathcal{E}^+, \mathcal{E}^- \rangle, \mathcal{K}))$ we generate facts, if at all possible, for contraries of assumptions in $\mathcal{K}$ that, when added to $\mathcal{R}$, enable the entailment of the examples in $\langle \mathcal{E}^+, \mathcal{E}^- \rangle$.
- The set of ASP rules at points (a)–(e) is denoted $ASP^*(\langle \mathcal{R}, \mathcal{A}, \overline{} \rangle, \langle \mathcal{E}^+, \mathcal{E}^- \rangle, \mathcal{K})$. It can be used similarly to $ASP^+(\langle \mathcal{R}, \mathcal{A}, \overline{} \rangle, \langle \mathcal{E}^+, \mathcal{E}^- \rangle, \mathcal{K})$, but also generates facts for positive examples that cannot be obtained by $ASP^+(\langle \mathcal{R}, \mathcal{A}, \overline{} \rangle, \langle \mathcal{E}^+, \mathcal{E}^- \rangle, \mathcal{K})$.

The *RoLe* procedure repeatedly applies Rote Learning, which adds a fact $\rho : p(X) \leftarrow X = t$, where t is a tuple of constants, to $\mathcal{R}$ (thus, $\mathcal{R}' = \mathcal{R} \cup \{\rho\}$). We illustrate with the *innocent* running example.

Example 3. *Let us consider the learning problem* $(\langle \mathscr{R},\mathscr{A},\overline{}\rangle,\langle \mathscr{E}^+,\mathscr{E}^-\rangle)$ *of Example 2. In this case,* $ASP^*(\langle \mathscr{R},\mathscr{A},\overline{}\rangle,\langle \mathscr{E}^+,\mathscr{E}^-\rangle,\mathscr{A})$ *consists of* $\mathscr{R}$ *rewritten in ASP syntax and of the following ASP rules:*

```
not_guilty(X) :- person(X), not guilty(X).          :- not innocent(mary).
guilty(X) :- person(X), not not_guilty(X).          :- not innocent(bob).
:- not_guilty(X), guilty(X).                        :- innocent(john).
innocent(X) :- person(X), not neg_innocent(X).      :- innocent(david).
neg_innocent(X) :- person(X), not innocent(X).
:- innocent(X), neg_innocent(X).
```

In this example, $\mathscr{C}(ASP^*(\langle \mathscr{R},\mathscr{A},\overline{}\rangle,\langle \mathscr{E}^+,\mathscr{E}^-\rangle,\mathscr{A}))\setminus \mathscr{C}(ASP(\langle \mathscr{R},\mathscr{A},\overline{}\rangle))$ *includes* `innocent(bob)`, `guilty(david)`, `guilty(john)`*, and by Rote Learning, we obtain:*

$$\mathscr{R}_1 = \mathscr{R} \cup \{innocent(X) \leftarrow X = bob,\ guilty(X) \leftarrow X = david,\ guilty(X) \leftarrow X = john\}$$

$\langle \mathscr{R}_1,\mathscr{A},\overline{}\rangle$ *is a (non-intensional, due to the added ground facts) solution of our ABA Learning problem.*

Now, the *ASP-ABAlearn* learning strategy proceeds by applying the *GEN* procedure, for transforming a non-intensional rule into an intensional one. First, *GEN* applies (once or more times) Folding, which, given rules ρ_1: $H \leftarrow Eqs_1, B_1, B_2$ and ρ_2: $K \leftarrow Eqs_1, Eqs_2, B_1$ in $\mathscr{R}$, replaces ρ_1 by ρ_3: $H \leftarrow Eqs_2, K, B_2$ (hence, $\mathscr{R}' = (\mathscr{R}\setminus\{\rho_1\})\cup\{\rho_3\}$). Folding is a form of *inverse resolution* [9], which generalises a rule by replacing some atoms in its body with their 'consequence' using a rule in $\mathscr{R}$.

Example 4. *By applying Folding to* $innocent(X) \leftarrow X = bob$ *and* $guilty(X) \leftarrow X = david$*, using* $away(X) \leftarrow X = bob$*,* $witness\_con(X,Y) \leftarrow X = david,\ Y = carol$*, and* $person(Y) \leftarrow Y = carol$ *in* $\mathscr{R}_1$*, we get:*

$$\begin{aligned}\mathscr{R}_2 = \mathscr{R} \cup \{ & innocent(X) \leftarrow away(X),\ guilty(X) \leftarrow witness\_con(X,Y),\ person(Y),\\ & guilty(X) \leftarrow X = john\}.\end{aligned}$$

The ABA framework $\langle \mathscr{R}_2,\mathscr{A},\overline{}\rangle$*, though, is not a solution, as it does no longer entail* $innocent(mary)$*.*

If the effect of Folding a rule is that the resulting ABA framework is no longer a solution of the given learning problem, *GEN* applies Assumption Introduction and Rote Learning with the goal of deriving a new ABA framework which is again a (non-intensional) solution. Assumption Introduction replaces a rule $\rho_1 : H \leftarrow B$ in $\mathscr{R}$ by $\rho_2 : H \leftarrow B, \alpha(X)$, where $X = vars(H) \cup vars(B)$ and $\alpha(X)$ is a new assumption with contrary $c\_\alpha(X)$ (thus, $\mathscr{R}' = (\mathscr{R}\setminus\{\rho_1\})\cup\{\rho_2\}$, $\mathscr{A}' = \mathscr{A}\cup\{\alpha(X)\}$, $\overline{\alpha(X)}' = c\_\alpha(X)$, and $\overline{\beta}' = \overline{\beta}$ for all $\beta \in \mathscr{A}$). New facts for $c\_\alpha(X)$ are learnt by Rote Learning by using $ASP^+(\langle \mathscr{R}',\mathscr{A}',\overline{}'\rangle,\langle \mathscr{E}^+,\mathscr{E}^-\rangle,\{\alpha(X)\}$. The facts for $c\_\alpha(X)$ can be seen as the *exceptions* to the *defeasible* rule ρ_2.

Example 5. *By Assumption Introduction, we get:*

$$\begin{aligned}\mathscr{R}_3 = \mathscr{R} \cup \{ & innocent(X) \leftarrow away(X),\quad guilty(X) \leftarrow witness\_con(X,Y),\ person(Y),\ a(X,Y),\\ & guilty(X) \leftarrow X = john\}\end{aligned}$$

with $\mathscr{A}' = \mathscr{A} \cup \{a(X,Y) \mid X,Y \in \{alex,\ bob, carol, david, john, mary\}\}$ *and* $\overline{a(X,Y)}' = c\_a(X,Y)$*. To determine the facts for* $c\_a(X,Y)$*, we use* $ASP^+(\langle \mathscr{R}_3,\mathscr{A},\overline{}\rangle,\langle \mathscr{E}^+,\mathscr{E}^-\rangle,\{a(X,Y)\})$ *(we omit the encoding of the background knowledge* $\mathscr{R}$*):*

```
innocent(X) :- away(X).                                  :- not innocent(mary).
guilty(X) :- X=john.                                     :- not innocent(bob).
guilty(X) :- witness_con(X,Y), person(Y), alpha1(X,Y).   :- innocent(david).
alpha1(X,Y) :- witness_con(X,Y), not c_alpha1(X,Y).      :- innocent(john).
c_alpha1(X,Y) :- witness_con(X,Y), not alpha1(X,Y).
:- alpha1(X,Y), c_alpha1(X,Y).
```

$\mathscr{C}(ASP^{+}(\langle \mathscr{R}_3, \mathscr{A}, \overline{} \rangle, \langle \mathscr{E}^{+}, \mathscr{E}^{-} \rangle, \{a(X,Y)\}))$ *contains the atom* `c_alpha1(mary,alex)` *and thus, by Rote Learning, we obtain again a (non-intensional) solution, by adding a fact for predicate* $c\_a(X,Y)$*:*

$$\mathscr{R}_4 = \mathscr{R}_3 \cup \{c\_a(X,Y) \leftarrow X = mary,\ Y = alex\}$$

GEN proceeds by applying the Subsumption rule, which gets rid of redundant facts. Indeed, suppose that $\mathscr{R}$ contains $\rho : p(X) \leftarrow X = t$ and let $\mathscr{R}' = \mathscr{R} \setminus \{\rho\}$. If $\langle \mathscr{R}', \mathscr{A}', \overline{}' \rangle \models p(t)$, then, by Subsumption, ρ can be deleted from $\mathscr{R}$. Subsumption is applicable if $\mathtt{p(t)} \in \mathscr{C}(ASP(\langle \mathscr{R}', \mathscr{A}', \overline{}' \rangle))$.

Example 6. *The rule* $\rho = guilty(X) \leftarrow X = john$ *can be deleted, as* $\mathtt{guilty(john)} \in \mathscr{C}(ASP(\langle \mathscr{R}_4 \setminus \{\rho\} \rangle))$.

ASP-ABAlearn halts when *GEN* generates no new contrary, as Folding yields an intensional solution.

Example 7. *By two final applications of the Folding rule, GEN gets:*

$$\begin{aligned}\mathscr{R}_5 = \mathscr{R} \cup \{\, & innocent(X) \leftarrow away(X), \quad guilty(X) \leftarrow witness\_con(X,Y),\ person(Y),\ a(X,Y),\\ & c\_a(X,Y) \leftarrow defendant(X), liar(Y)\}\end{aligned}$$

Now, $\langle \mathscr{R}_5, \mathscr{A}', \overline{}' \rangle$ *is an intensional solution of the given learning problem.*

5 Discussion and Conclusion

We have revisited a strategy recently proposed for learning ABA frameworks based on transformation rules [12], and we have shown that, in the case of the stable extension semantics, many of the reasoning tasks used by that strategy can be implemented through an ASP solver.

A proof-of-concept implementation of our *ASP-ABAlearn* strategy is ongoing using SWI-Prolog (v. 9.0.4) and the Clingo ASP solver (v. 5.6.2). It consists of two Prolog modules implementing *RoLe* and *GEN* and two further modules implementing (i) the *ASP*, ASP^{+}, and ASP^{*} encodings, and (ii) the API to invoke Clingo from SWI-Prolog and collect the cautious consequences to be used by *RoLe* and *GEN*. The most critical issue for implementing *GEN* is that the application of Folding is non-deterministic, as there may be different choices for the rules to be used for applying that transformation. Currently, we simply make use of a bound to limit the number of alternatives. The design of more sophisticated mechanisms to control Folding, e.g., based on the notion of *information gain*[17], is left as future work.

In addition to refining the implementation, we also plan to perform an experimental comparison to non-monotonic ILP systems (such as Fold [17] and ILASP [8]). On the theoretical side, further work is needed to investigate conditions under which *ASP-ABAlearn* is complete, in the sense that it terminates and finds a solution if it exists. A simple way of guaranteeing termination is based on a mechanism for avoiding the generation of contraries that are "equivalent" to previously generated ones. However, the solution obtained in this way is not guaranteed to be intensional.

Acknowledgments

We thank support from the Royal Society, UK (IEC\R2\222045 - International Exchanges 2022). De Angelis and Proietti are members of the INDAM-GNCS research group; they were partially supported by the PNRR MUR project PE0000013-FAIR, Italy. Toni was partially funded by the ERC under the EU's Horizon 2020 research and innovation programme (grant agreement No. 101020934) and by J.P. Morgan and the Royal Academy of Engineering, UK, under the Research Chairs and Senior Research Fellowships scheme.

References

[1] A. Bondarenko, P.M. Dung, R.A. Kowalski & F. Toni (1997): *An Abstract, Argumentation-Theoretic Approach to Default Reasoning*. Artif. Intell. 93, pp. 63–101, doi:`10.1016/S0004-3702(97)00015-5`.

[2] K. Cyras, X. Fan, C. Schulz & F. Toni (2017): *Assumption-based Argumentation: Disputes, Explanations, Preferences*. FLAP 4(8).

[3] Y. Dimopoulos & A. C. Kakas (1995): *Learning Non-Monotonic Logic Programs: Learning Exceptions*. In: ECML 1995, pp. 122–137, doi:`10.1007/3-540-59286-5_53`.

[4] P.M. Dung, R.A. Kowalski & F. Toni (2009): *Assumption-Based Argumentation*. In: Argumentation in Artificial Intelligence, Springer, pp. 199–218, doi:`10.1007/978-0-387-98197-0_10`.

[5] M. Gelfond & V. Lifschitz (1991): *Classical negation in logic programs and disjunctive databases*. New Generation Computing 9, pp. 365–385, doi:`10.1007/BF03037169`.

[6] K. Inoue & H. Haneda (2000): *Learning Abductive and Nonmonotonic Logic Programs*. In: Abduction and Induction: Essays on their Relation and Integration, Kluwer Academic, pp. 213–231, doi:`10.1007/978-94-017-0606-3_14`.

[7] K. Inoue & Y. Kudoh (1997): *Learning Extended Logic Programs*. In: IJCAI, Morgan Kaufmann, pp. 176–181.

[8] M. Law, A. Russo & K. Broda (2014): *Inductive Learning of Answer Set Programs*. In: JELIA 2014, LNCS 8761, pp. 311–325, doi:`10.1007/978-3-319-11558-0_22`.

[9] S. Muggleton (1995): *Inverse entailment and Progol*. New generation computing 13(3-4), pp. 245–286, doi:`10.1007/BF03037227`.

[10] A. Pettorossi & M. Proietti (1994): *Transformation of Logic Programs: Foundations and Techniques*. J. Log. Program. 19/20, pp. 261–320, doi:`10.1016/0743-1066(94)90028-0`.

[11] N. Prentzas, A. Gavrielidou, M. Neofytou & A. C. Kakas (2022): *Argumentation-based Explainable Machine Learning (ArgEML): A Real-life Use Case on Gynecological Cancer*. In: ArgML 2022, CEUR Workshop Proceedings, 3208, pp. 1–13.

[12] M. Proietti & F. Toni (2023): *Learning Assumption-based Argumentation Frameworks*. CoRR, doi:`10.48550/arXiv.2305.15921`. To appear in Proc. ILP '22, LNCS, Springer.

[13] O. Ray (2009): *Nonmonotonic abductive inductive learning*. J. Appl. Log. 7(3), pp. 329–340, doi:`10.1016/j.jal.2008.10.007`.

[14] C. Sakama (2000): *Inverse Entailment in Nonmonotonic Logic Programs*. In: Proc. ILP 2000, doi:`10.1007/3-540-44960-4_13`.

[15] C. Sakama (2005): *Induction from answer sets in nonmonotonic logic programs*. ACM TOCL 6(2), pp. 203–231, doi:`10.1145/1055686.1055687`.

[16] C. Sakama & K. Inoue (2009): *Brave induction: A logical framework for learning from incomplete information*. Mach. Learn. 76(1), pp. 3–35, doi:`10.1007/s10994-009-5113-y`.

[17] F. Shakerin, E. Salazar & G. Gupta (2017): *A new algorithm to automate inductive learning of default theories*. TPLP 17(5-6), pp. 1010–1026, doi:`10.1017/S1471068417000333`.

[18] F. Toni (2014): *A tutorial on Assumption-based Argumentation*. Argument & Computation 5(1), pp. 89–117, doi:`10.1080/19462166.2013.869878`.

Complexity and Scalability of Reasoning in many-valued Weighted Knowledge Bases with Typicality
Extended Abstract

Mario Alviano
University of Calabria, Italy
`mario.alviano@unical.it`

Laura Giordano
Daniele Theseider Dupré
University of Piemonte Orientale, Italy
`laura.giordano@uniupo.it` `dtd@uniupo.it`

In this extended abstract we report about new results about reasoning on weighted knowledge bases for Description Logics (DLs) with typicality under a "concept-wise" multi-preferential semantics, which have been shown to provide a logical interpretation of MultiLayer Perceptrons [18]. In particular, we provided a completeness result for the complexity of the problem, and new Answer Set Programming encodings in the same complexity class, that are shown to be able to deal with weighted knowledge bases with large search spaces. A detailed description in available in the full paper [1].

Description logics are widely used for knowledge representation, often to verify and discover properties of individuals in a concept by means of DLs inference services [2, 22]. Many properties of real world concepts, however, are *defeasible*, that is, they are not universally true, but have exceptions, and actually hold only for some *prototypical* individuals in the concept. This led to research on *defeasible DLs* [8, 15, 10, 16]. Specifically, to represent the defeasible properties of a concept, DLs have been extended with a *typicality operator* **T** that is applied to concepts to obtain *typicality inclusions* of the form $\mathbf{T}(C) \sqsubseteq D$ [15, 16]. Intuitively, $\mathbf{T}(C) \sqsubseteq D$ means that the typical individuals in the concept C also belong to concept D (that, *normally* C's are D's), and corresponds to a *conditional implication* $C \mathrel{|\!\sim} D$ in KLM preferential logics [24, 25]. From the semantic point of view a preference relation is considered as in KLM preferential semantics, to select the most typical individual in a class (concept). A (conditional) knowledge base (KB) comprising typicality inclusions enables *defeasible reasoning*, as in fact properties holding for typical individuals in C are not necessarily enforced on all individuals in C.

Based on the idea that each concept C can be associated a set of prototypical properties, *ranked DL knowledge bases* [17] — reminiscent of ranked KBs by Brewka [6] — and *weighted DL KBs* with typicality [18] have been proposed. In particular, in weighted knowledge bases, a set $\mathscr{T}_C$ of *weighted typicality inclusions* $(\mathbf{T}(C) \sqsubseteq D_1,\ w_1), \ldots, (\mathbf{T}(C) \sqsubseteq D_k,\ w_k)$ is associated to a concept C to describe the prototypical properties of C-elements. For each property, the associated weight w_i is a real number, representing the plausibility/ implausibility of the property D_i for C-elements.

Semantically, these formalisms have led to a *concept-wise multi-preferential semantics*, as the relative typicality of two domain individuals usually depends on the aspects we are considering for comparison [20, 17]: Bob may be a more typical as sport lover than Jim ($bob <_{SportLover} jim$), but Jim may be a more typical than Bob as a swimmer ($jim <_{Swimmer} bob$). It has been proven that this semantics has interesting properties in both the two-valued and in the fuzzy case (e.g., it is able to deal with specificity, irrelevance and satisfies the KLM properties).

The concept-wise multi-preferential semantics has been used to provide a logical interpretation to some neural network models, such as Self-Organising Maps [23], psychologically and biologically plausible neural network models, and for *MultiLayer Perceptrons* (MLPs) [21]. In both cases, a preferential interpretation can be constructed from a trained network (see [14] and [18]) and exploited for verification of typicality properties of the network through *model checking*. For MLPs a trained network can as

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 9–11, doi:10.4204/EPTCS.385.2

well be regarded as a weighted knowledge base by encoding synaptic connections as weighted typicality inclusions under a fuzzy semantics [18, 13]. This concrete application and the widespread interest in neural networks strongly motivates the development of proof methods for reasoning with weighted KBs.

Entailment for fuzzy DLs is in general undecidable [11, 4], and this motivates the investigation of many-valued approximations of fuzzy multi-preferential entailment. In particular, the finitely many-valued case is widely studied [12, 3, 5], and has been recently considered in the context of weighted DL KBs [19] by means of the notions of *coherent*, *faithful* and *φ-coherent* models of such KBs, previously considered in the fuzzy case [18, 13]. A proof-of-concept implementation in Answer Set Programming (ASP) and *asprin* [7] has been provided for the $\mathcal{LC}$ fragment of $\mathcal{ALC}$, which is obtained by disabling roles, and universal and existential restrictions. The approach adopts Gödel (alternatively, Łukasiewicz) connectives and addresses φ-coherent entailment, a form of defeasible reasoning based on canonical φ-coherent models. The φ-coherence condition makes weighted KBs correspond to MLPs with φ as activation function; here a finitely many-valued approximation of such a function is used. As concerns the complexity of the problem, a Π_2^p upper bound was given [19], but the exact complexity was unknown.

We briefly describe here two significant advancements with respect to the earlier work [19], from a theoretical point of view and on the practical side, for the φ-coherent entailment of typicality inclusions [1]. The complexity upper bound can be improved to $P^{NP[log]}$ by showing an algorithm running in polynomial time and performing *parallel* queries to an NP oracle ($P^{||NP}$). As $P^{||NP}$ is known to coincide with $P^{NP[log]}$ [9], while $\Pi_2^p = P^{NP[log]}$ is unlikely to hold (unless the polynomial hierarchy collapses to $P^{NP[log]}$), there was space for improving the proof-of-concept implementation.

Different ASP encodings have been considered. For the enforcement of φ-coherence (where grounding explosion should be avoided) the best results are obtained using weight constraints. Weak constraints are used to encode the preferential semantics, relying on an *order encoding* to stay in the $P^{NP[log]}$ class. Further improvements at an asymptotic level are unlikely, as the problem is shown to be $P^{NP[log]}$-complete by giving a polynomial-time reduction of the MAX SAT ODD problem [26], which amounts to determining whether the maximum number of jointly satisfiable clauses among a given set is an odd number.

The scalability of the ASP encodings has been assessed empirically on defeasible entailment queries over synthetic weighted DL KBs to show that the redesigned ASP encoding can go beyond the small KBs and search spaces processable by the earlier proof-of-concept implementation [1].

Acknowledgments

This work was partially supported by GNCS-INdAM. Mario Alviano was partially supported by Italian Ministry of Research (MUR) under PNRR project FAIR "Future AI Research", CUP H23C22000860006 and by the LAIA lab (part of the SILA labs).

References

[1] M. Alviano, L. Giordano & D. Theseider Dupré (2023): *Complexity and scalability of defeasible reasoning in many-valued weighted knowledge bases*. *CoRR* abs/2303.04534, doi:10.48550/arXiv.2303.04534.

[2] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi & P.F. Patel-Schneider (2007): *The Description Logic Handbook - Theory, Implementation, and Applications*. Cambridge, doi:10.1017/CBO9780511711787.

[3] F. Bobillo, M. Delgado, J. Gómez-Romero & U. Straccia (2012): *Joining Gödel and Zadeh Fuzzy Logics in Fuzzy Description Logics*. *Int. J. Uncertain. Fuzziness Knowl. Based Syst.* 20(4), pp. 475–508, doi:10.1142/S0218488512500249.

[4] S. Borgwardt & R. Peñaloza (2012): *Undecidability of Fuzzy Description Logics*. In Gerhard Brewka, Thomas Eiter & Sheila A. McIlraith, editors: *Proc. KR 2012, Rome, Italy, June 10-14, 2012.*

[5] S. Borgwardt & R. Peñaloza (2013): *The Complexity of Lattice-Based Fuzzy Description Logics*. J. Data Semant. 2(1), pp. 1–19, doi:10.1007/s13740-012-0013-x.

[6] G. Brewka (2004): *A Rank Based Description Language for Qualitative Preferences*. In: 6th Europ. Conf. on Artificial Intelligence, ECAI'2004, Valencia, Spain, August 22-27, 2004, pp. 303–307.

[7] G. Brewka, J. P. Delgrande, J. Romero & T. Schaub (2015): *asprin: Customizing Answer Set Preferences without a Headache*. In: Proc. AAAI 2015, pp. 1467–1474.

[8] K. Britz, J. Heidema & T. Meyer (2008): *Semantic Preferential Subsumption*. In G. Brewka & J. Lang, editors: KR 2008, AAAI Press, Sidney, Australia, pp. 476–484.

[9] S. R. Buss & L. Hay (1991): *On Truth-Table Reducibility to SAT*. Inf. Comput. 91(1), pp. 86–102, doi:10.1016/0890-5401(91)90075-D.

[10] G. Casini & U. Straccia (2010): *Rational Closure for Defeasible Description Logics*. In T. Janhunen & I. Niemelä, editors: JELIA 2010, LNCS 6341, Springer, pp. 77–90, doi:10.1007/978-3-642-15675-5_9.

[11] M. Cerami & U. Straccia (2011): *On the Undecidability of Fuzzy Description Logics with GCIs with Lukasiewicz t-norm*. CoRR abs/1107.4212. Available at `http://arxiv.org/abs/1107.4212`.

[12] A. García-Cerdaña, E. Armengol & F. Esteva (2010): *Fuzzy Description Logics and t-norm based fuzzy logics*. Int. J. Approx. Reason. 51(6), pp. 632–655,doi:10.1016/j.ijar.2010.01.001.

[13] L. Giordano (2021): *On the KLM properties of a fuzzy DL with Typicality*. In: Proc. ECSQARU 2021, Prague, Sept. 21-24, 2021, LNCS 12897, Springer, pp. 557–571, doi:10.1007/978-3-030-86772-0_40.

[14] L. Giordano, V. Gliozzi & D. Theseider Dupré (2022): *A conditional, a fuzzy and a probabilistic interpretation of self-organizing maps*. J. Log. Comput. 32(2), pp. 178–205, doi:10.1093/logcom/exab082.

[15] L. Giordano, V. Gliozzi, N. Olivetti & G. L. Pozzato (2009): *ALC+T: a Preferential Extension of Description Logics*. Fundamenta Informaticae 96, pp. 1–32, doi:10.3233/FI-2009-182.

[16] L. Giordano, V. Gliozzi, N. Olivetti & G. L. Pozzato (2015): *Semantic characterization of rational closure: From propositional logic to description logics*. Art. Int. 226, pp. 1–33, doi:10.1016/j.artint.2015.05.001.

[17] L. Giordano & D. Theseider Dupré (2020): *An ASP approach for reasoning in a concept-aware multipreferential lightweight DL*. Theory Pract. Log. Program. (TPLP) 10(5), pp. 751–766, doi:10.1017/S1471068420000381.

[18] L. Giordano & D. Theseider Dupré (2021): *Weighted defeasible knowledge bases and a multipreference semantics for a deep neural network model*. In: Proc. JELIA 2021, May 17-20, LNCS 12678, pp. 225–242, doi:10.1007/978-3-030-75775-5_16.

[19] L. Giordano & D. Theseider Dupré (2022): *An ASP approach for reasoning on neural networks under a finitely many-valued semantics for weighted conditional knowledge bases*. TPLP 22(4), pp. 589–605, doi:10.1017/S1471068422000163,.

[20] V. Gliozzi (2016): *Reasoning about Multiple Aspects in Rational Closure for DLs*. In: Proc. AI*IA 2016, pp. 392–405, doi:10.1007/978-3-319-49130-1_29.

[21] S. Haykin (1999): *Neural Networks - A Comprehensive Foundation*. Pearson.

[22] Pascal Hitzler, Markus Krötzsch & Sebastian Rudolph (2010): *Foundations of Semantic Web Technologies*. Chapman and Hall/CRC Press. Available at `http://www.semantic-web-book.org/`.

[23] T. Kohonen, M.R. Schroeder & T.S. Huang, editors (2001): *Self-Organizing Maps, Third Edition*. Springer Series in Information Sciences, Springer.

[24] S. Kraus, D. Lehmann & M. Magidor (1990): *Nonmonotonic Reasoning, Preferential Models and Cumulative Logics*. Artificial Intelligence 44(1-2), pp. 167–207, doi:10.1016/0004-3702(90)90101-5.

[25] D. Lehmann & M. Magidor (1992): *What does a conditional knowledge base entail?* Artificial Intelligence 55(1), pp. 1–60, doi:10.1016/0004-3702(92)90041-U.

[26] Klaus W. Wagner (1990): *Bounded Query Classes*. SIAM J. Comput. 19(5), pp. 833–846, doi:10.1137/0219058.

Benchmarking for Integrating Logic Rules with Everything Else

Yanhong A. Liu Scott D. Stoller Yi Tong K. Tuncay Tekle
Computer Science Department, Stony Brook University, Stony Brook, New York, USA
{liu,stoller,yittong,tuncay}@cs.stonybrook.edu

Integrating logic rules with other language features is increasingly sought after for advanced applications that require knowledge-base capabilities. To address this demand, increasingly more languages and extensions for such integration have been developed. How to evaluate such languages?

This paper describes a set of programming and performance benchmarks for evaluating languages supporting integrated use of rules and other features, and the results of evaluating such an integrated language together with logic languages and languages not supporting logic rules.

1 Introduction

As knowledge-base capabilities are increasingly needed for advanced applications, especially applications that require sophisticated reasoning, logic rules are increasingly needed together with other language features. This leads to increasingly more languages and extensions that provide such programming support, for using rules through libraries or directly as built-ins, in many different ways. To evaluate such language support for integrated uses of rules with other language features, a set of benchmarking problems is needed.

Many benchmarking suites have been developed for evaluating logic languages and rule engines, and even drastically more for imperative and other languages. In contrast, benchmarks and evaluation for languages that support integrated use of rules together with other language features have been lacking.

This paper describes a set of programming and performance benchmarks for such evaluation, and the uses of these benchmarks in evaluating an integrated language together with logic languages and languages not supporting logic rules. The benchmarks are developed and organized into three sets:

OpenRuleBench—for problems focused on using rules and evaluating rule systems, including all problems from OpenRuleBench [15], but modified and organized to use more appropriate language features for the benchmarking itself, especially imperative scripts for running and taking measurements.

RBAC—for problems that require (1) frequent imperative updates interleaved with expensive queries and (2) objects and classes with inheritance to organize system components, subsuming full Role-Based Access Control (RBAC) [1, 17], with queries expressed in different ways of using rules and not using rules.

PA—for problems expressed using rules together with set queries and aggregate queries and recursive functions, from problems in program analysis (PA), because aggregate queries are particularly important for many database, data mining, and machine learning applications, but are not in OpenRuleBench [15].

The input data and workload for these benchmarks are designed for runs with different problem scales—(1) large input data for queries using rules, (2) large query results from inference using rules, (3) large number of rules, (4) frequent switches between using rules and using other features, and (5) frequent invocations of inference using rules.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 12–26, doi:10.4204/EPTCS.385.3

The evaluation is for benchmark problems written in (1) Alda, (2) XSB, (3) Python, and (4) DistAlgo. XSB [28, 32] is a top logic rule engine for queries using rules, selected for reasons stated in Section 3.1. Python [27] is most used for its rich features including high-level queries but lacks logic rules. DistAlgo [20, 16] extends Python for distributed programming with higher-level and faster queries but still lacks logic rules. Alda [21] is an integrated language that supports logic rules as built-ins by building on DistAlgo and employing XSB. The evaluation yields the following main results:

- Queries using rules in Alda and XSB are similar, and are (1) simpler than using `while` loops in Python and DistAlgo, even when these loops use high-level set queries, and (2) drastically, asymptotically faster than `while` loops with high-level set queries in Python and DistAlgo.
- Benchmarking using more generic imperative scripts in Alda is drastically simpler than using rules in XSB in OpenRuleBench; and integrated use of rules with other features for RBAC and PA problems yields simpler programs than not using rules or only using rules.
- The performance of Alda programs is the accumulated performance of queries in XSB, other features in Python and DistAlgo, and interface between XSB and Python. Thus it is expected to increase accordingly when any part used is improved, including reducing the current interface overhead to 1% of it.
- Even with the current interface overhead, Alda programs are faster or even drastically faster than half or more of the rule engines tested in OpenRuleBench [15] for all but one benchmark, because the overhead is only linear in sizes of data and results and number of queries, and thanks to the XSB query performance.

Our analysis supports that similar results can be obtained by using other efficient rule engines, including ASP systems, and other languages not supporting logic rules, including lower-level languages. Our implementation has been available by request and will be made public pending improved documentation.

2 Language

XSB and Python are well established, and DistAlgo is subsumed by Alda. So we first introduce Alda, an integrated language that supports rules with all of sets, functions, updates, and objects, all as built-ins, seamlessly integrated without extra interface code. Figure 1 shows an example program in Alda.

Rules are defined using rule sets (e.g. lines 15–17, with rule set name `trans_rs`), and queried using calls to an inference function, `infer` (e.g. on line 19, using rule set `trans_rs`). The two rules in `trans_rs` (lines 16–17) define predicate `path` using predicate `edge`, where the first rule is the base case, and the second rule is the recursive case. The call to `infer` (line 19) returns the result of querying `path`, i.e., the set of pairs for which `path` holds, given that `edge` equals `RH`, i.e., `edge` holds for the set of pairs in `RH`.

In a rule set, predicates not in any conclusion are called base predicates; the other predicates are called derived predicates. The key idea is that predicates are simply set-valued variables and vice versa. So predicates can be defined and used directly as other variables, in any scope—global, object field, or local to a rule set. The only exception is that derived predicates can only be updated by `infer`, and are automatically maintained by an implicit call to `infer` when any non-local base predicates are updated. The exception ensures the declarative semantics of rules.

All other features—class, inheritance, method (function and procedure), update, and set query—are the same as in object-oriented languages. We mostly use Python syntax except for a few conventions from Java (`extends` for inheritance, `new` for object creation, and omission of `self` when there is no ambiguity) for ease of reading, and a few ideal syntax (`:=` for assignment, `{}` for empty set, and `+` for set union), and pattern matching in queries (in examples explained later).

For the example in Figure 1 but without using rules, computing the transitive closure (i.e., result of `infer` in function `transRH`) in a variable `T`, by adding a parameter `E` to function `transRH` and passing the value of `RH` to `E` when calling `transRH`, can use a `while` loop in Python, as used for [17]:

```
class CoreRBAC:                  # class for Core RBAC component/object
  def setup():                   # method to set up the object, with no arguments
    self.USERS, self.ROLES, self.UR := {},{},{}
                                 # set users, roles, user-role pairs to empty sets
  def AddRole(role):             # method to add a role
    ROLES.add(role)              # add the role to ROLES
  def AssignedUsers(role):       # method to return assigned users of a role
    return {u: u in USERS | (u.role) in UR}  # return set of users having the role
  ...
class HierRBAC extends CoreRBAC: # Hierarchical RBAC extending Core RBAC
  def setup():
    super().setup()              # call setup of CoreRBAC, to set sets as in there
    self.RH := {}                # set ascendant-descendant role pairs to empty set
  def AddInheritance(a,d):       # to add inherit. of an ascendant by a descendant
    RH.add((a,d))                # add pair (a,d) to RH
  rules trans_rs:                # rule set defining transitive closure
    path(x,y) if edge(x,y)       # path holds for (x,y) if edge holds for (x,y)
    path(x,y) if edge(x,z), path(z,y)  # ... if edge holds for (x,z) and for (z,y)
  def transRH():                 # to return transitive RH and reflexive role pairs
    return infer(path, edge=RH, rules=trans_rs) + {(r,r): r in ROLES}
  def AuthorizedUsers(role):     # to return users having a role transitively
    return {u: u in USERS. r in ROLES | (u.r) in UR and (r.role) in transRH()}
  ...
h = new(HierRBAC, [])            # create HierRBAC object h, with no args to setup
h.AddRole('chair')               # call AddRole of h with role 'chair'
...
h.AuthorizedUsers('chair')       # call AuthorizedUsers of h with role ‘chair’
...
```

Figure 1: An example program in Alda, for Role-Based Access Control (RBAC), demonstrating logic rules used with sets, functions, updates, and objects.

```
T = E.copy()
W = {(x,y) for (x,z) in T for (z2,y) in E if z2==z} - T
while W:
  T.add(W.pop())
  W = {(x,y) for (x,z) in T for (z2,y) in E if z2==z} - T
```

or a `while` loop with higher-level set queries in DistAlgo:

```
T := E.copy()
while some (x,z) in T, (z,y) in E | (x,y) not in T:
  T.add((x,y))
```

Using rules is clearly simpler, both conceptually and in the amount of code.

Alda is implemented by extending the DistAlgo compiler [20, 16] and invoking XSB [28, 32] for inference. DistAlgo is compiled to Python. Function `infer` translates data from Python to Prolog facts, translates results back, and invokes XSB in between using command line and file passing. The obvious overhead of this external interface can be removed with an in-memory interface between Python to XSB, which is actively being developed by the XSB team.[1] However, this has not affected Alda programs from having generally good performance, thanks to the XSB query performance.

3 Programming and performance benchmarks

---

[1] A version working for Unix, not yet Windows, has been released, and passing lists of length 100 million in memory took about 30 nanoseconds per element [32, release notes]. So even for the largest data in our experiments, of size a few millions, it would take 0.1-0.2 seconds to pass in memory, instead of 10-20 seconds with the current external interface.

Name	Description	Prog size	Data size
Join1	non-recursive tree of binary joins as inference rules	225	*
Join2	join from IRIS system producing large intermediate result	41	*
JoinDup	join of separate results of five copies of Join1	163	*
LUBM	university database adapted from LUBM benchmark	377	*
Mondial	geographical database derived from CIA Factbook	36	59,733
DBLP	well-known bibliography database on the Web	20	2,437,867
TC	classical transitive closure of a binary relation	75	*
SG	well-known same-generation siblings problem	90	*
WordNet	natural language processing queries based on WordNet	298	465,703
Wine	well-known OWL wine ontology as rules	1103	654
ModSG	modified SG to exclude ancestor-descendant relationships	38	*
Win	well-known win-not-win game with non-stratified negation	24	*
MagicSet	non-stratified rules from magic-set transformation	34	*

The three groups (of 6, 4, 3) in order are called large join tests, Datalog recursion, and default negation, respectively. Prog size is the XSB program size in lines of code without comments and empty lines. Data size is the input data size in number of facts; * means that scripts are used to generate input data of desired sizes.

Table 1: Benchmarks from OpenRuleBench.

This section presents three sets of benchmarks that we developed. They are written to express each given problem in the most direct way we can think of. This helps make the benchmarks clear and easy to read. This also avoids being unfair to any particular language or system being compared with.

3.1 OpenRuleBench—a wide variety of rule-based applications

OpenRuleBench [15] contains a wide variety of database, knowledge base, and semantic web application problems, written using rules in 11 well-known rule systems from 5 different categories, as well as large data sets and a large number of test scripts for running and measuring the performance. Among 14 benchmarks described in [15], we consider all except for one that tests interfaces of rule systems with databases (which is a non-issue for Python and its extensions DistAlgo and Alda, because Python has standard and widely-used database interfaces).

Table 1 summarizes the benchmarks. We compare with the benchmark programs in XSB, for three reasons: (1) XSB has been the most advanced rule system supporting well-founded semantics for non-stratified negation and tabling techniques for efficient query evaluation, and has been actively developed for over three decades, to this day; (2) among all systems reported in [15], XSB was one of the fastest, if not the fastest, and the single most consistent across all benchmarks; and (3) among all measurements reported, only XSB, OntoBroker, and DLV could run all benchmarks, but OntoBroker went bankrupt, and measurements for DLV were almost all slower, often by orders of magnitude.

We easily translated all 13 benchmarks into Alda, automatically for all except for three cases where the original rules used features beyond Datalog, which became two cases after we added support for negation. In all cases, it was straightforward to express the desired functionality in Alda, producing a program that is very similar to XSB or even simpler. Additionally, the code for reading data, running tests, timing, and writing results is drastically simpler in Alda than in XSB, because Alda extends Python. The three exception cases and additional findings are described below.

Result set. In most logic languages, including Prolog and many variants, a query returns only the first result that matches the query. To return the set of all results, some well-known tricks are used. The LUBM benchmark includes the following extra rules to return all answers of `query9_1`:

```
query9 :- query9_1(X,Y,Z), fail.
query9 :- writeln('========query9.======').
```

The first rule first queries `query9_1` to find an answer (a triple of values for `X,Y,Z`) and then uses `fail` to trick the inference into thinking that it failed to find an answer and so continuing to search for an answer; and it does this repeatedly, until `query9_1` does fail to find an answer after exhausting all answers. The second rule is necessary, even if with an empty right side, to trick the inference into thinking that it succeeded, because the first rule always ends in failing; this is so that the execution can continue to do the remaining work instead of stopping from failing.

In fact, this trick is used in all benchmarks, but other uses are buried inside the code for running, timing, etc., specialized for each benchmark, not as part of the rules for the application logic.

In Alda, such rules and tricks are never needed. A call to `infer` with query `query9_1` returns the set of all query results as desired.

Function symbols. Logic rules may use function symbols to form structured data that can be used as arguments to predicates. Uses of function symbols can be translated away. The Mondial benchmark uses a function symbol `prov` in several intermediate conclusions and hypotheses of the form `isa(prov(Y,X),provi)` or `att(prov(Y,X),number,A)`. They can simply be translated to `isa('prov',Y,X,provi)` and `att('prov',Y,X,number,A)`, respectively.

Negation. Logic languages may use negation applied to hypotheses in rules. Most logic languages only support stratified negation, where there is no negation involved in cyclic dependencies among predicates. Such negation can be done by set differences. The ModSG benchmark has such a negation, as follows, where `sg` is defined by the rules in the SG benchmark, and `nonsg` is defined by two new rules.

```
sg2(X,Y) :- sg(X,Y), not nonsg(X,Y).
```

In Alda, this can be written as

```
sg2 = infer(sg, rules=sg_rs) - infer(nonsg, rules=nonsg_rs)
```

where `sg_rs` and `nonsg_rs` are the rule sets defining `sg` and `nonsg`, respectively.

Alda also supports negation in rules. Its current implementation translates negation to tabled negation `tnot` in XSB. This handles even non-stratified negation by computing well-founded semantics using XSB [3], contrasting Prolog's negation as failure. The Win and MagicSet benchmarks have non-stratified negation. Both of them, as well as ModSG, can be expressed directly in Alda rule sets by using `not` for negation.

Benchmarking and organization. In OpenRuleBench benchmarks, even though the rules to be benchmarked are declarative and succinct, the benchmarking code for reading input, running tests, timing, and writing results are generally much larger. For example, the Join1 benchmark has 4 small rules and 9 small queries similar in size to those in the transitive closure example, plus a manually added tabling directive for optimization. However, for each of the 9 queries, 19 more lines for an import and two much larger rules are used to do the reading, running, timing, and writing.

In general, because benchmarking executes a bundle of commands, scripting those directly is simplest. Furthermore, organizing benchmarking code using procedures, objects, etc., allows easy reuse without duplicated code. These features are much better supported in languages like Python than rule systems, for both ease of programming and performance,

In fact, OpenRuleBench uses a large number of many different files, in several languages (language of the system being tested, XSB, shell script, Python, makefile) for such scripting. For example for Join1, the 4 rules, tabling directive, and benchmarking code are also duplicated in each of the 9 XSB files, one for each query; a 46-line shell script and a 9-line makefile are also used.

In contrast, our benchmarking code is in Alda, which uses Python functions for scripting. A single 45-line Alda program is used for timing any of the benchmarks, and for pickling (i.e., object serialization in Python, for fast data reading after the first reading) and timing of pickling.

Name	Features used for computing transitive role hierarchy
RBACnonloc	rule set `transRH_rs` with implicit `infer`, without `transRH()`, Sec. 3.2
RBACallloc	rule set `trans_role_rs` and `transRH()` that has only `infer`, Sec. 3.2
RBACunion	rule set `trans_rs` and `transRH()` that has `infer` and union, Sec. 2
RBACda	`while` loop and high-level set queries in DistAlgo, Sec. 2
RBACpy	`while` loop and high-level set comprehensions in Python, Sec. 2

Each benchmark performs a combination of updates to sets and relations `USERS`, `ROLES`, `UR`, and `RH` and queries with function `AuthorizedUsers(role)`, where the transitive role hierarchy is computed with a different way of using rules, or not using rules. In `AuthorizedUsers(role)` of all five programs, the call to `transRH()`, or reference to field `transRH`, is lifted out of the set query, by assigning its value to a local variable and using that variable in the query.

Table 2: Benchmarks for RBAC updates and queries.

Aggregation. Despite the wide variety of benchmarks in OpenRuleBench, no benchmark uses aggregate queries. Aggregate queries are essential for many database, data mining, and machine learning applications. We discuss them and compare with aggregate queries in a rule language like XSB in Section 3.3.

3.2 RBAC—rules with objects, updates, and set queries

The ANSI standard for Role-Based Access Control (RBAC [7, 1] involves many sets and query and update functions in a total of 9 components. To program the transitive role hierarchy at a high level, a complex and inefficient `while` loop was used before [17], because an efficient algorithm would be drastically even more complex.

With support for rules, the entire RBAC standard is easily written in Alda, similar as in Python [17], except with rules for computing the transitive role hierarchy, as shown in Figure 1, yielding a simpler yet more efficient program.

Below, we specify more ways of using rules to compute the transitive role hierarchy and function `AuthorizedUsers(role)` in Figure 1. All these ways are declarative and differ in size by only 1-2 lines. Table 2 summarizes the benchmarks for RBAC that include all RBAC classes with their inheritance relationships and perform update operations and these query functions in different ways.

In particular, in the first way below, a field, `transRH`, is used and maintained automatically; it avoids calling `transRH()` repeatedly, as desired in the RBAC standard, and it does so without the extra maintenance code in the RBAC standard for handling updates.

Rules with only non-local predicates. Using rules with only non-local predicates, one can add a field `transRH`, and use `transRH` in place of calls to `transRH()`, e.g., in function `AuthorizedUsers(role)`, and use the following rule set instead of `trans_rs` in class `HierRBAC`:[2]

```
rules transRH_rs:  # no need to use infer explicitly
  transRH(x,y) if RH(x,y)
  transRH(x,y) if RH(x,z), transRH(z,y)
  transRH(x,x) if ROLES(x)
```

Field `transRH` is automatically maintained at updates to `RH` and `ROLES` by implicit calls to `infer`; no explicit calls to `infer` are needed. This eliminates the need of function `transRH()` and repeated expensive calls to it even when its result is not changed most of the time. Overall, this simplifies the program, ensures correctness, and improves efficiency.

---

[2] The first rule could actually be omitted, because the second argument of `RH` is always in `ROLES` and thus the second rule when joining `RH` with reflexive pairs in `transRH` from the third rule subsumes the first rule.

Part	Analysis	Features used
1 Ext	classes, extension relation	rules (recursive if refined name analysis is used)
2 Stat	statistics, roots	aggregate and set queries
3 Hgt	max height, roots of max height	recursive functions, aggregate and set queries
4 Desc	max desc, roots of max desc	recursive rules, functions, aggregate and set queries

In Parts 1 and 4 that use rules, not using rules (esp. for recursive analysis, with tabling) would be drastically worse (i.e., harder to program and less efficient). In Parts 2-4 that use aggregate and set queries, using rules or recursive functions would be clearly worse. In Parts 3 and 4 that use functions, not using functions (with tabling, also called caching) would be much worse.

Table 3: Benchmark PA for program analysis, integrating different kinds.

Rules with only local predicates. Using rules with only local predicates, `infer` must be called explicitly. One can simply use the function `transRH()` in Figure 1, which calls `infer` using rule set `trans_rs` in the running example and then unions with reflexive role pairs. Alternatively, one can use the rules in `trans_rs` plus a rule that uses a local predicate `role` to infer reflexive role pairs:

```
rules trans_role_rs:  # as trans_rs plus the added last rule
  path(x,y) if edge(x,y)
  path(x,y) if edge(x,z), path(z,y)
  path(x,x) if role(x)
def transRH():        # use infer only, pass in also ROLES
  return infer(path, edge=RH, role=ROLES, rules=trans_role_rs)
```

Both ways show the ease of using rules by simply calling `infer`. Despite possible inefficiency in some cases, using only local predicates has the advantage of full reusability of rules and full control of calls to `infer`.

3.3 Program analysis—rules with aggregate queries and recursive functions

We designed a benchmark for program analysis (PA) problems, especially to show integrated use of rules with aggregate queries and recursive functions. Aggregate queries help quantify and characterize the analysis results, and recursive functions help do these on recursive structures. We describe programs written in Alda and then in XSB.

The benchmark is for analysis of class hierarchy of Python programs. It uses logic rules to extract class names and construct the class extension relation; aggregate queries and set queries to characterize the results and find special cases of interest; recursive functions as well as aggregate and set queries to analyze the special cases; and more logic rules, functions, and set and aggregate queries to further analyze the special cases.

Table 3 summarizes different parts of this benchmark, called PA. A variant, called PAopt, is the same as PA except that, in the recursive rule for defining transitive descendant relationship, the two hypotheses are reversed, following previously studied optimizations [18, 33].

Because the focus is on evaluating the integrated use of different features, each part that uses a single feature, such as rules, is designed to be small. Compared with making each part larger, which exercises individual features more, this design highlights the overhead of connecting different parts, in terms of both ease of use and efficiency of execution.

The benchmark program takes as input the abstract syntax tree (AST) of a Python program (a module or an entire package), represented as a set of facts. Each AST node of type T with k children corresponds to a fact for predicate T with k+1 arguments: id of the node, and ids of the k children. Lists are represented using `Member` facts. A `Member(`*lst*`,`*elem*`,`*i*`)` fact denotes that list *lst* has element *elem* at the *i*th position.

Part 1: Classes and class extension relation. This part examines all `ClassDef` nodes in the AST. A `ClassDef` node has 5 children: class name, list of base-class expressions, and three nodes not used for this analysis. The following rules can be used to find all defined class names and build a class extension relation using base-class expressions that are `Name` nodes. A `Name` node has two children: name and context.

```
rules class_extends_rs:
  defined(c) if ClassDef(_,c,_, _,_,_)
  extending(c,b) if ClassDef(_,c,baselist, _,_,_),
                    Member(baselist,base,_), Name(base,b,_)
```

For a dynamic language like Python, analysis involving names can be refined in many ways to give more precise results, e.g., [9]. We do not do those here, but Datalog rules are particularly good for such analysis of bindings and aliases for names, e.g., [30].

Part 2: Characterizing results and finding special cases. This part computes statistics for defined classes and the class extension relation and finds root classes (class with subclass but not super class). These use aggregate queries and set queries, where `(_,c)` and `(=c,_)` are tuple patterns, `_` matches anything, `c` is bound to the matched value, and `=c` matches the value of `c`.

```
num_defined := count(defined)
num_extending := count(extending)
avg_extending := num_extending/num_defined
roots := {c: (_,c) in extending, not some (=c,_) in extending}
```

Similar queries can compute many other statistics and cases: maximum number of classes that any class extends, leaf classes, histograms, etc.

Part 3: Analysis of special cases. This part computes the maximum height of the extension relation, which is the maximum height of the root classes, and finds root classes of the maximum height. These use a recursive function as well as aggregate and set queries.

```
def height(c):
  return 0 if not some (_,=c) in extending
         else 1 + max{height(d): (d,=c) in extending}
max_height := max{height(r): r in roots}
roots_max_height := {r: r in roots, height(r) = max_height}
```

For efficiency when a subclass can extend multiple base classes, caching of results of function calls is used. In Python, one can simply add `import functools` to import module `functools`, and add `@functools.cache` just above the definition of `height` to cache the results of that function.

Part 4: Further analysis of special cases. This part computes the maximum number of descendant classes following the extension relation from a root class, and finds root classes of the maximum number. Recursive functions and aggregate queries similar to finding maximum height do not suffice here, due to shared subclasses that may be at any depth. Instead, the following rules can infer all `desc(c,r)` facts where class `c` is a descendant following the extension relation from root class `r`, and aggregate and set queries with function `num_desc` then compute the desired results.

```
rules desc_rs:
  desc(c,r) if roots(r), extending(c,r)
  desc(c,r) if desc(b,r), extending(c,b)
def num_desc(r):
  return count{c: (c,=r) in desc}
max_desc := max{num_desc(r): r in roots}
roots_max_desc := {r: r in roots, num_desc(r) = max_desc}
```

For efficiency of the last query, caching is also used for function `num_desc`. If the last query is omitted, function `num_desc` can also be inlined in the `max_desc` query.

Comparing with aggregate queries and functions in rule languages. While rules in Alda correspond directly to rules in rule languages, expressing aggregate queries and functions using rules require translations that formulate computations as hypotheses and introduce additional variables to relate these hypotheses.

Aggregate queries are used extensively in database and machine learning applications, and are essential for analyzing large data or uncertain information. These queries are easy to express directly in database languages and scripting languages, but are less so in rule languages like Prolog; most rule languages also do not support general aggregation with recursion due to their subtle semantics [19]. For example, the simple query `num_defined := count(defined)` in Alda, when written in XSB, becomes:

```
num_defined(N) :- setof(C, defined(C), S), length(S, N).
```

Recursive functions are used extensively in list and tree processing and in solving divide-and-conquer problems. They are natural for computing certain information about parse trees, nested scopes, etc. However, in rule languages, they are expressed in a way that mixes function arguments and return values, and require sophisticated mode analysis to differentiate arguments from returns. For example, the `height` query, when written in XSB, becomes:

```
height(C,0) :- not(extending(_,C)).
height(C,H) :- findall(H1, (extending(D,C), height(D,H1)), L),
               max_list(L,H2), H is H2+1.
```

4 Experimental results

We present results about running times, program sizes, and data sizes. All measurements were taken on a machine with an Intel Xeon X5690 3.47 GHz CPU, 94 GB RAM, running 64-bit Ubuntu 16.04.7, Python 3.9.9, and XSB 4.0.0. For each experiment, the reported running times are CPU times averaged over 10 runs. Garbage collection in Python was disabled for smoother running times when calling XSB. Program sizes are numbers of lines excluding comments and empty lines. Data sizes are number of facts.

Compilation times and program sizes. Table 4 shows Alda compilation times and related XSB, Alda, DistAlgo, and Python program sizes before and after compilation. They are for all benchmarks described in Section 3, plus three variants of TC, explained below in the paragraph on performance of classical queries using rules. The Alda programs are 4–970 lines for OpenRuleBench benchmarks, 385–423 lines for RBAC benchmarks, and 33 lines for PA benchmarks. For each group of benchmarks, there is a single shared Alda file of benchmarking code, shown in the last row of each group.

The compilation times for all programs are 0.6 seconds or less, and for all but RBAC benchmarks and Wine in OpenRuleBench are about 0.1 seconds or less.

For Alda programs that have corresponding XSB programs (OpenRuleBench in Table 1 and PA), Alda programs are all much smaller, almost all by dozens or even hundreds of lines, and by an order of magnitude for Join1 and TC in OpenRuleBench, because we have all the benchmarking code in the shared benchmarking file.

Performance of classical queries using rules. We experimented with four programs for computing transitive closure: TC, the TC benchmark from OpenRuleBench, which is the same as `trans_rs` except with renamed predicates; TCrev, a well-known variant with the two predicates reversed in the recursive rule; and TCpy and TCda, which use the Python and DistAlgo `while` loops, respectively, in Section 2. For comparison, we also directly run the XSB program for TC from OpenRuleBench, and its corresponding

Benchmark name	Original XSB size	Alda size	Compilation time (ms)	Generated Python size	Generated XSB size
Join1	225	23	33.0	32	5
Join2	41	11	18.5	16	9
JoinDup	163	42	45.6	20	36
LUBM	377	125	116.4	29	110
Mondial	36	8	16.2	16	6
DBLP	20	4	16.3	16	2
TC	75	5	7.9	16	3
TCrev	*75	5	7.7	16	3
TCda	–	5	4.9	18	-
TCpy	–	7	6.5	11	-
SG	90	13	17.9	20	7
WordNet	298	58	76.2	44	28
Wine	1103	970	605.6	16	968
ModSG	38	14	14.8	16	12
Win	24	4	9.5	16	2
MagicSet	34	9	18.2	16	7
ORBtimer	–	45	35.2	56	-
RBACnonloc	–	423	346.4	538	4
RBACallloc	–	387	318.4	481	4
RBACunion	–	386	316.6	481	3
RBACda	–	385	312.8	483	-
RBACpy	–	387	314.6	476	-
RBACtimer	–	44	43.3	67	-
PA	*55	33	49.7	93	6
PAopt	*55	33	40.8	93	6
PAtimer	–	40	32.6	56	-

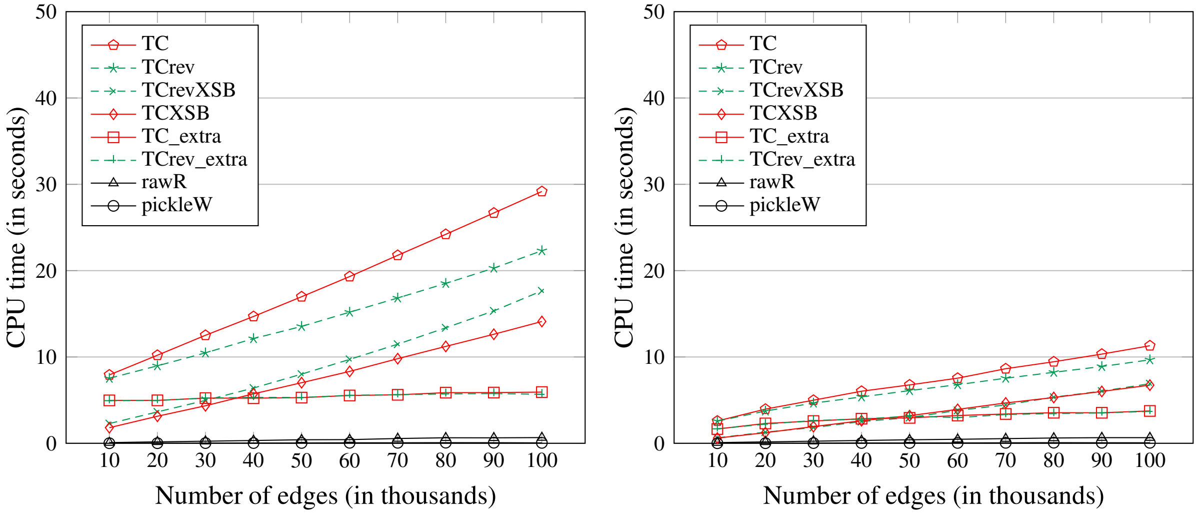
For Original XSB size, entries without * are from OpenRuleBench, as in Table 1; * indicates XSB programs we added; – means there is no corresponding XSB program. For Generated XSB size, - means no XSB code is generated.

Table 4: Compilation times and program sizes before and after compilation.

version for TCrev, except we change `load_dyn` to `load_dync`, for much faster reading of facts in XSB's canonical form; we call these programs TCXSB and TCrevXSB, respectively. The same data generation as OpenRuleBench is used to return large query results—almost complete graphs.

Figure 2 shows the running times of the TC benchmarks. RawR and PickleW are times for reading facts in XSB/Prolog form as used in OpenRuleBench and writing them in pickled form for use in Alda, respectively. Pickling is done only once; the pickled data is read in all repeated runs and all of TC, TCrev, TCda, and TCpy. The running time of Alda programs includes not only (1) reading data, (2) executing queries, and (3) returning results, but also (2pre) preparing data, queries, and commands and writing data to files for XSB before (2), and (2post) reading results from files written by XSB after (2). TC_extra and TCrev_extra are the part of TC and TCrev, respectively, for extra work interfacing with XSB, i.e., for 2pre and 2post and for XSB to read data (xsbRdata) and write results (xsbWres). A breakdown showing the time needed for each step in the extra work is in [22].

The results are as expected: the two Alda programs that use rules are asymptotically, drastically faster than Python and DistAlgo `while` loops, and they exhibit known notable performance differences [33, 34]. Most notably but as expected, passing the query results back from XSB has a high overhead, up to 5.9 seconds, out of 29.2 seconds total, for graphs of 100K edges, but this overhead is expected to be reduced



TCpy and TCda are not in the charts because they are asymptotically slower and took drastically longer: on 100 vertices, with cyclic data of 200 edges (2% of smallest data point in the charts), TCpy took 624.6 seconds, and TCda took 249.9 seconds; and with acyclic data of 600 edges, TCpy took 160.4 seconds, and TCda took 65.2 seconds.

Figure 2: Running times of TC benchmarks on cyclic (left) and acyclic (right) graphs.

to 1% of it when the in-memory Python-XSB interface is used. Note that reversing the two predicates in the recursive rule does give a linear-factor asymptotically different running time, but that barely shows because OpenRuleBench uses almost complete graphs.

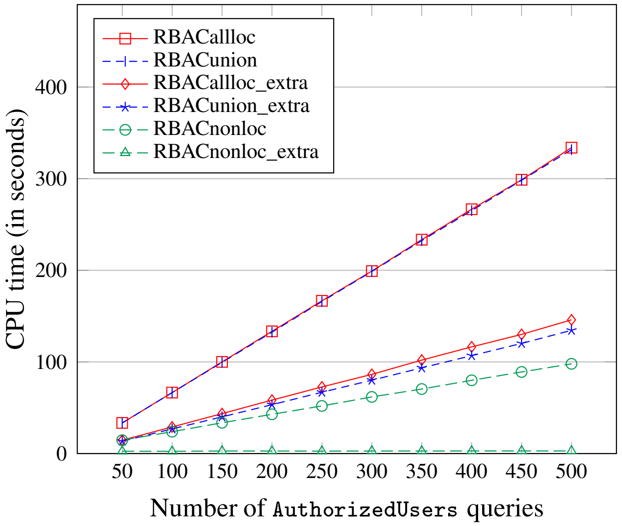
Integrating with objects, updates, and set queries. We use RBAC benchmarks in Table 2, Section 3.2, for this evaluation, especially with frequent queries and updates and intensively frequent restart of XSB for queries randomly mixed with updates of the queried data: 5000 users, 500 roles, 5000 UR relation, 550 RH relation, up to 500 queries, and 230 updates of various kinds.

Figure 3 shows the running times of the RBAC benchmarks, all scaling linearly in the number of queries, as expected. Labels with suffix _extra indicate the part of the running time of the corresponding program for extra work interfacing with XSB. A breakdown of the time for extra work is in [22].

The results are as expected as well: RBACunion and RBACalloc are very close, and are much slower than RBACnonloc—up to 331.7 and 333.9 seconds, respectively, vs. 97.9 seconds. Most notably but as expected, the overhead of repeated queries using XSB is high for RBACunion and RBACalloc, but low for RBACnonloc, up to 134.5 and 145.9 seconds, respectively, vs. 2.8 seconds. The highest overhead is from restarting XSB 500 times, which will be totally eliminated when in-memory interface is used.

Integrating with aggregate queries and recursive functions. We use PA and PAopt benchmarks and their corresponding programs in XSB, as described in Section 3.3, for this evaluation, and we focus on applying the analysis to large programs as input data. The programs analyzed include 9 widely-used open-source Python packages for a wide range of application domains: NumPy, SciPy, MatPlotLib, Pandas, SymPy, Blender, Django, Scikit-learn, and PyTorch—with 641K–5.1M input facts total and 252K–2.2M facts used by the analysis.

Table 5 shows data sizes, analysis results, and running times of the analysis. The columns are sorted by the total number of facts used. A breakdown of the running time into steps interfacing with XSB as well as the remainder of the running time is in [22].



RBACpy #queries	time (s)
50	688.7
100	1381.5
150	> 30 min

RBACda #queries	time (s)
50	384.6
100	768.1
150	1141.7
200	1517.5
250	> 30 min

RBACpy and RBACda are not in the chart because they are drastically slower.

Figure 3: Running times of RBAC benchmarks, for a workload of updates and queries over 5000 users, 500 roles, 5500 user-role assignments, and 550 role hierarchy pairs.

The results are not as expected: we found the corresponding XSB programs to be highly inefficient, being all slower and even drastically slower than Alda programs, even 120 times slower for PyTorch. Significant effort was spent on performance debugging and manual optimization, and we eventually created a version that is faster than Alda—5.1 seconds vs. 15.2 seconds on the largest input, SymPy—by using additional directives for targeted tabling that also subsumes some indexing.

As expected, the Alda programs here have a high overhead of passing the data to XSB, up to 13.1 seconds on SymPy, which again is expected to be reduced to 1% of it with in-memory Python-XSB interface. This means that the resulting Alda programs would be faster than even the manually optimized XSB, showing that computations not using rules, e.g., aggregations and functions, are not only simpler and easier in Alda/Python than in XSB but also faster.

Scaling with data and rules. We use the two largest benchmarks from OpenRuleBench: DBLP, with over 2.4 million facts, the largest real-world data set among all in OpenRuleBench; and Wine, with 961 rules, the largest rule set among all. Table 6 shows the running times for both benchmarks, for both the Alda programs and the XSB programs. _extra is the part of the total time on 2pre, 2post, xsbRdata, and xsbWres. OrigTotal is the Total time for the original program from OpenRuleBench, which uses `load_dyn` instead of `load_dync`.

The results are again as expected. For DBLP, XSB is more than three times as fast as Alda, 9.5 seconds vs. 30.6 seconds, but as for PA benchmarks, the overhead of passing the large data to XSB is large, here 26.9 seconds, and is expected to be reduced to 1% of that; note that the Alda program has faster reading from pickled data.

For Wine, XSB is more than eight times as fast as Alda, 3.8 seconds vs. 31.0 seconds. This is due to the use of `auto_table` in Alda generated code, which does variant tabling, whereas this program, through manual debugging and optimization, was found to need subsumptive tabling [34]. Optimizations [18, 33, 34] can be added to the Alda compiler to match this efficiency automatically. Note that this slowest Alda program is still faster than half of the systems tested in OpenRuleBench, which took up to 140 seconds and three systems gave errors, and where XSB was the fastest at 4.47 seconds [15].

Measure	Item/Name	numpy	django	sklearn	blender	pandas	matplot	scipy	pytorch	sympy
Data size	Total	640,715	815,551	862,031	909,600	942,315	1,064,859	1,092,466	5,142,905	5,115,105
	`ClassDef`	587	1,835	535	2,146	849	994	898	6,467	1,830
	`Name`	96,076	119,077	137,066	107,638	153,664	152,357	178,754	797,072	1,063,842
	`Member`	155,207	199,416	210,410	242,531	227,766	268,736	260,848	1,270,917	1,112,296
	Total used	251,870	320,328	348,011	352,315	382,279	422,087	440,500	2,074,456	2,177,968
Ratio	Used/Total	39.3%	39.3%	40.4%	38.7%	40.6%	39.6%	40.3%	40.3%	42.6%
Result size	`#defined`	519	1610	533	2118	804	935	882	4323	1786
	`#extending`	419	1457	710	2951	407	610	719	2207	1816
	`#roots`	79	225	51	133	88	104	60	137	92
	`max_height`	8	7	5	4	7	5	3	5	12
	`#roots_max_h`	1	2	2	1	2	1	4	1	1
	`#desc`	427	2329	822	4376	436	605	721	2174	2413
	`max_desc`	84	309	256	1,638	65	47	353	1,045	1,078
	`#roots_max_d`	1	1	1	1	1	1	1	1	1
Running time (in seconds)	PA	2.542	3.573	3.263	5.134	3.342	3.733	3.646	14.652	15.243
	PAopt	2.631	3.520	3.235	4.661	3.341	3.676	3.633	14.706	15.132
	PAXSB	6.297	112.091	10.795	243.765	6.378	14.400	22.221	969.228	65.382
	PAoptXSB	13.170	343.428	17.066	326.629	18.863	40.871	29.675	1773.374	181.961
	PAXSBopt	0.804	1.499	1.071	2.149	1.118	1.317	1.300	5.158	5.051
Ratio over PA	PAopt	103.5%	98.5%	99.1%	90.8%	100.0%	98.5%	99.6%	100.4%	99.3%
	PAXSB	247.7%	3137.2%	330.8%	4748.0%	190.8%	385.7%	609.5%	6615.0%	428.9%
	PAoptXSB	518.1%	9611.7%	523.0%	6362.1%	564.4%	1094.9%	813.9%	12103.3%	1193.7%
	PAXSBopt	31.6%	42.0%	32.8%	41.9%	33.5%	35.3%	35.6%	35.2%	33.1%

Total is the total number of facts about each package. Total used is the sum of numbers of `ClassDef`, `Name`, and `Member` facts.

Table 5: Data size, analysis results, and running times for program analysis benchmarks.

Name	Alda								XSB	
	RawR	PickleW	2pre	xsbRdata	xsbWres	2post	_extra	Total	Total	OrigTotal
DBLP	12.187	3.131	15.722	11.197	0.054	0.020	26.993	30.573	9.492	63.494
Wine	0.008	0.000	0.037	0.219	0.000	0.001	0.257	30.960	3.754	3.826

Table 6: Running times (in seconds) of DBLP and Wine benchmarks.

5 Related work and conclusion

Many benchmarking suites have been developed for evaluating the performance of queries in logic languages and rule engines, including the carefully constructed OpenRuleBench for comprehensively evaluating a diverse set of problems in a wide range of systems [15]. In particular, Prolog has had benchmarks for comparing performance of different implementations [5, 4]. Some are focused on a special class of problems, e.g., interpreters written in Prolog [13]. Some are for evaluating the performance of a particular implementation, e.g., SWI Prolog [31]. There are also works that evaluate queries in more general rule engines and database systems, e.g., the LUBM benchmark [11] and its extensions [24, 29] for OWL on a range of systems, and evaluations including SQL with rules [12, 2]. There are also works focused on evaluating interfaces, e.g., Java Prolog Interface [23]. These works study only queries.

Even drastically more benchmarking suites were developed for evaluating other systems. Works range from systematic studies, e.g., [14, 10, 25], to a large variety of specific benchmarks, e.g., the SPEC benchmarks for computer systems in general [25], the TPC benchmarks for transaction processing [26], the LINPACK benchmarks [6] on solving linear equations, and many more. These benchmarks exercise updates and many other language features but not logic rules.

In contrast, our work develops benchmarks that exercise integrated use of rules together with other language features. We improve OpenRuleBench benchmarks with significantly simplified benchmarking code, and design new benchmarks that exercise tightly integrated uses of different features on problems that test different problem scales in different ways. We also compare different ways of using rules vs. not using rules.

In conclusion, this work presents a set of programming and performance benchmarks for evaluating languages supporting integrated use of rules and other features, and the results of using these benchmarks in an evaluation. Future work can perform evaluations with additional languages and systems, especially efficient ASP systems such as Clingo [8].

Acknowledgments. We thank Bo Lin for his robust DistAlgo compiler implementation, and his initial help in implementing the Alda compiler, and we thank David S. Warren for his initial help in implementing an interface to XSB. We also thank Thang Bui for additional applications in program analysis and optimization, and students in undergraduate and graduate courses for using Alda and its earlier versions, called DA-rules. This work was supported in part by NSF under grants CCF-1954837, CCF-1414078, and IIS-1447549 and ONR under grants N00014-21-1-2719, N00014-20-1-2751, and N00014-15-1-2208.

References

[1] ANSI INCITS (2004): *Role-Based Access Control*. ANSI INCITS 359-2004, American National Standards Institute, International Committee for Information Technology Standards.

[2] Stefan Brass & Mario Wenzel (2019): *Performance Analysis and Comparison of Deductive Systems and SQL Databases*. In: *Proceedings of the 3rd International Workshop on the Resurgence of Datalog in Academia and Industry*, CEUR-WS.org, pp. 27–38.

[3] Weidong Chen & David S. Warren (1996): *Tabled Evaluation with Delaying for General Logic Programs*. *Journal of the ACM* 43(1), pp. 20–74, doi:`10.1145/227595.227597`.

[4] (2001): *The CHINA Benchmark Suite*. `http://www.cs.unipr.it/China/Benchmarks`. Accessed Feb. 1, 2023.

[5] (1985): *Prolog Benchmarking Suites*. `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/code/bench/0.html`. Accessed Feb. 1, 2023.

[6] Jack J Dongarra, Piotr Luszczek & Antoine Petitet (2003): *The LINPACK benchmark: past, present and future*. *Concurrency and Computation: practice and experience* 15(9), pp. 803–820, doi:`10.1002/cpe.728`.

[7] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn & Ramaswamy Chandramouli (2001): *Proposed NIST Standard for Role-Based Access Control*. *ACM Transactions on Information and Systems Security* 4(3), pp. 224–274, doi:`10.1145/501978.501980`.

[8] Martin Gebser, Roland Kaminski, Benjamin Kaufmann & Torsten Schaub (2019): *Multi-shot ASP solving with clingo*. *Theory and Practice of Logic Programming* 19(1), pp. 27–82, doi:`10.1017/S1471068418000054`.

[9] Michael Gorbovitski, Yanhong A. Liu, Scott D. Stoller, Tom Rothamel & Tuncay Tekle (2010): *Alias Analysis for Optimization of Dynamic Languages*. In: *Proceedings of the 6th Symposium on Dynamic Languages*, ACM Press, pp. 27–42, doi:`10.1145/1869631.1869635`.

[10] Jim Gray, editor (1993): *The Benchmark Handbook for Database and Transaction Processing Systems*, 2nd edition. Morgan Kaufmann Publishers.

[11] Yuanbo Guo, Zhengxiang Pan & Jeff Heflin (2005): *LUBM: A benchmark for OWL knowledge base systems*. *Journal of Web Semantics* 3(2-3), pp. 158–182, doi:`10.1016/j.websem.2005.06.005`.

[12] Ulrich John, Petra Hofstedt, Armin Wolf et al. (2019): *A New Benchmark Database and An Analysis of Transitive Closure Runtimes*. *Deklarative Ansätze zur Künstlichen Intelligenz–punktuelle Beiträge*, p. 3.

[13] Philipp Körner, David Schneider & Michael Leuschel (2020): *On the performance of bytecode interpreters in Prolog*. In: *International Workshop on Functional and Constraint Logic Programming*, Springer, pp. 41–56, doi:`10.1007/978-3-030-75333-7_3`.

[14] Byron C. Lewis & Albert E. Crews (1985): *The evolution of benchmarking as a computer performance evaluation technique*. *MIS Quarterly*, pp. 7–16, doi:`10.2307/249270`.

[15] Senlin Liang, Paul Fodor, Hui Wan & Michael Kifer (2009): *OpenRuleBench: An Analysis of the Performance of Rule Engines*. In: *Proceedings of the 18th International Conference on World Wide Web*, ACM Press, pp. 601–610, doi:`10.1145/1526709.1526790`.

[16] Bo Lin & Yanhong A. Liu (2014 (Latest update January 30, 2022)): *DistAlgo: A Language for Distributed Algorithms*. `http://github.com/DistAlgo`. Accessed May 25, 2023.

[17] Yanhong A. Liu & Scott D. Stoller (2007): *Role-Based Access Control: A Corrected and Simplified Specification*. In: *Department of Defense Sponsored Information Security Research: New Methods for Protecting Against Cyber Threats*, Wiley, pp. 425–439.

[18] Yanhong A. Liu & Scott D. Stoller (2009): *From Datalog Rules to Efficient Programs with Time and Space Guarantees*. *ACM Transactions on Programming Languages and Systems* 31(6), pp. 1–38, doi:`10.1145/1552309.1552311`.

[19] Yanhong A. Liu & Scott D. Stoller (2022): *Recursive Rules with Aggregation: A Simple Unified Semantics*. *Journal of Logic and Computation* 32(8), pp. 1659–1693, doi:`10.1093/logcom/exac072`. Also `http://arxiv.org/abs/2007.13053`.

[20] Yanhong A. Liu, Scott D. Stoller & Bo Lin (2017): *From Clarity to Efficiency for Distributed Algorithms*. *ACM Transactions on Programming Languages and Systems* 39(3), pp. 12:1–12:41, doi:`10.1145/2994595`. Also `http://arxiv.org/abs/1412.8461`.

[21] Yanhong A. Liu, Scott D. Stoller, Yi Tong & Bo Lin (2023): *Integrating logic rules with everything else, seamlessly*. *Theory and Practice of Logic Programming*. Special issue for selected papers from ICLP 2023. To appear. Also `http://arXiv.org/abs/2305.19202`.

[22] Yanhong A. Liu, Scott D. Stoller, Yi Tong, Bo Lin & K. Tuncay Tekle (2022): *Programming with Rules and Everything Else, Seamlessly*. *Computing Research Repository* arXiv:2205.15204 [cs.PL], doi:`10.48550/arXiv.2205.15204`.

[23] Jose E Zalacain Llanes (2022): *Java Prolog Interface*. *arXiv preprint arXiv:2203.17134*, doi:`10.48550/arXiv.2203.17134`.

[24] Li Ma, Yang Yang, Zhaoming Qiu, Guotong Xie, Yue Pan & Shengping Liu (2006): *Towards a complete OWL ontology benchmark*. In: *European Semantic Web Conference*, Springer, pp. 125–139, doi:`10.1007/11762256_12`.

[25] Raghunath Nambiar & Meikel Poess, editors (2009): *Performance Evaluation and Benchmarking*. Springer, doi:`10.1007/978-3-642-10424-4`.

[26] Raghunath Nambiar, Nicholas Wakou, Forrest Carman & Michael Majdalany (2010): *Transaction Processing Performance Council (TPC): State of the council 2010*. In: *Technology Conference on Performance Evaluation and Benchmarking*, Springer, pp. 1–9, doi:`10.1007/978-3-642-18206-8_1`.

[27] Python Software Foundation (2023): *Python*. `http://python.org/`. Accessed Jun 7, 2023.

[28] Konstantinos Sagonas, Terrance Swift & David S. Warren (1994): *XSB as an Efficient Deductive Database Engine*. In: *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, ACM Press, pp. 442–453, doi:`10.1145/191839.191927`.

[29] Gunjan Singh, Sumit Bhatia & Raghava Mutharaju (2020): *OWL2Bench: a benchmark for OWL 2 reasoners*. In: *International semantic web conference*, Springer, pp. 81–96, doi:`10.1007/978-3-030-62466-8_6`.

[30] Yannis Smaragdakis & George Balatsouras (2015): *Pointer Analysis*. *Foundations and Trends in Programming Languages* 2(1), pp. 1–69, doi:`10.1561/2500000014`.

[31] (2022): *SWI-Prolog benchmark suite*. `https://github.com/SWI-Prolog/bench`. Accessed Feb. 1, 2023.

[32] Theresa Swift, David S. Warren, Konstantinos Sagonas, Juliana Freire, Prasad Rao, Baoqiu Cui, Ernie Johnson, Luis de Castro, Rui F. Marques, Diptikalyan Saha, Steve Dawson & Michael Kifer (2022): *The XSB System Version 5.0,x*. `http://xsb.sourceforge.net`. Latest release May 12, 2022.

[33] K. Tuncay Tekle & Yanhong A. Liu (2010): *Precise Complexity Analysis for Efficient Datalog Queries*. In: *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pp. 35–44, doi:`10.1145/1836089.1836094`.

[34] K. Tuncay Tekle & Yanhong A. Liu (2011): *More Efficient Datalog Queries: Subsumptive Tabling Beats Magic Sets*. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pp. 661–672, doi:`10.1145/1989323.1989393`.

Explanations for Answer Set Programming

Mario Alviano
DEMACS, University of Calabria,
Via Bucci 30/B, 87036 Rende (CS), Italy
mario.alviano@unical.it

Ly Ly Trieu & Tran Cao Son
New Mexico State Universty
NM, USA
lytrieu|stran@nmsu.edu

Marcello Balduccini
Saint Joseph's University
PA , USA
mbalducc@sju.edu

The paper presents an enhancement of `xASP`, a system that generates explanation graphs for Answer Set Programming (ASP). Different from `xASP`, the new system, `xASP2` , supports different `clingo` constructs like the choice rules, the constraints, and the aggregates such as *#sum*, *#min*. This work formalizes and presents an explainable artificial intelligence system for a broad fragment of ASP, capable of shrinking as much as possible the set of assumptions and presenting explanations in terms of directed acyclic graphs.

1 Introduction

Recently, many modern artificial intelligence systems are increasingly capable of tackling complex problems. However, their lack of transparency can create a new issue: users may not comprehend why a solution was obtained, making it difficult to trust the results. Moreover, with the *right to an explanation* law extensively discussed in the USA, EU, and UK, and partly enacted in some countries, explainable artificial intelligence (XAI) has experienced a substantial increase in interest. Thus, the focus of this paper is on the development of an XAI system for Answer Set Programming (ASP) [10, 13]. Answer Set Programming (ASP) [10, 13] is a well-known paradigm for problem-solving using logic programs under answer set semantics [7] in knowledge representation and reasoning (KR&R) and an extension of Datalog with a strong connection with well-founded semantics [14]. A variety of applications such as planning, diagnosis, etc, have been successfully implemented using it. In this paper, our goal is to provide an answer to the question "*given an answer set A of a program* Π *and an atom* α*, why an atom* α *is true (or false) in A?*".

The emergence of XAI has brought significant attention from researchers in ASP community, resulting in numerous proposed systems aimed at addressing this issue such as xclingo [4], `DiscASP` [9], `xASP` [19], $\mathtt{exp}(\mathtt{ASP}^{\mathtt{c}})$ [18]. However, these systems are incapable of handling one or more of the following scenarios: (*i*) false atoms can be explained by the system, (*ii*) the ability to support certain advanced language features. In this paper, we proposed an improvement system, called `xASP2` , that takes inspiration from the approach used in `xASP` [19] and [16]. `xASP2` are able to substantially increase scalability and breadth of supported language features while producing explanation graphs with more immediately and consistently useful to users. The improvement presented here deals with two main issues in explaining the assignment of α in A: (*i*) how to compute a minimum cardinality set of atoms that is assumed to be false such that it is capable of explaining the assignment of α in A; and (*ii*) how to support sophisticated linguistic constructs such as choice rules and aggregates, which can be involved to explain the falsity of some atoms in easily understandable terms.

Our main contributions are the following:

- A notion of explanation in terms of directed acyclic graphs explains why an atom is (or is not) in an answer set in terms of easy-to-understand inferences originating from a hopefully minimum set

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 27–40, doi:10.4204/EPTCS.385.4

of assumed false atoms (Section 3). Note that the explanation graph of an atom is restricted to the atoms involved in the relevant rules for the explaining atoms.

- A proof of existence for the explanations according to the given definition that guarantees the correctness of our implementation (Section 4).
- The implementation of an enhancement system, `xASP2` , for producing explanations powered by ASP and its empirical evaluation(Sections 5–6). `xASP2` tackles logic programs with different `clingo` constructs such as aggregates and constraints.

The supported fragment of ASP includes uninterpreted function symbols, common aggregation functions, comparison expressions, strong negation, constraints, normal rules, and choice rules. Aggregates are expected to be stratified, to not involve default negation, and to have a single atomic condition. Choice rules are expected to be unconditional, or otherwise to have exactly one conditional atom with a self-explanatory condition (as for example a range expression or an extensional predicate). Additionally, to ease the presentation, in Section 2 we only consider *sum* aggregates, and completely omit uninterpreted function symbols, comparison expressions, strong negation, and conditions in choice rules. To the best of our knowledge, this is the first explanation generation system that supports different `clingo` constructs such as aggregates and constraints.

2 Background

All sets and sequences considered in this paper are finite. Let **P**, **C**, **V** be fixed nonempty sets of *predicate names*, *constants* and *variables*. Predicates are associated with an *arity*, a non-negative integer. A *term* is any element in $\mathbf{C} \cup \mathbf{V}$. An *atom* is of the form $p(\bar{t})$, where $p \in \mathbf{P}$, and $\bar{t}$ is a possibly empty sequence of terms. A *literal* is an atom possibly preceded by the default negation symbol *not*; they are referred to as positive and negative literals.

An *aggregate* is of the form

$$sum\{t_a, \overline{t'} : p(\bar{t})\} \odot t_g \tag{1}$$

where $\odot$ is a binary comparison operator, $p \in \mathbf{P}$, $\bar{t}$ and $\overline{t'}$ are possibly empty sequences of terms, and t_a and t_g are terms.

A *choice* is of the form

$$t_1 \leq \{atoms\} \leq t_2 \tag{2}$$

where *atoms* is a possibly empty sequence of atoms, and t_1, t_2 are terms. Let $\perp$ be syntactic sugar for $1 \leq \{\} \leq 1$.

A *rule* is of the form

$$head \leftarrow body \tag{3}$$

where *head* is an atom or a choice, and *body* is a possibly empty sequence of literals and aggregates. For a rule r, let $H(r)$ denote the atom or choice in the head of r; let $B^\Sigma(r)$, $B^+(r)$ and $B^-(r)$ denote the sets of aggregates, positive and negative literals in the body of r; let $B(r)$ denote the set $B^\Sigma(r) \cup B^+(r) \cup B^-(r)$.

A variable X occurring in $B^+(r)$ is a *global variable*. Other variables occurring among the terms $\bar{t}$ of some aggregate in $B^\Sigma(r)$ of the form (1) are *local variables*. And any other variable occurring in r is an *unsafe variable*. A *safe rule* is a rule with no *unsafe variables*. A *program* Π is a set of safe rules. Additionally, we assume that aggregates are stratified, that is, the *dependency graph* $\mathscr{G}_\Pi$ having a vertex for each predicate occurring in Π and an edge pq whenever there is $r \in \Pi$ with p occurring in $H(r)$ and q occurring in $B^+(r)$ or $B^\Sigma(r)$ is acyclic.

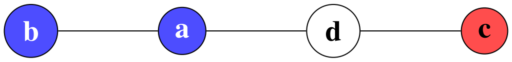


Figure 1: The undirected graph used as running example. Source vertices in blue, sink vertex in red.

Example 1. Given a connected undirected graph G encoded by predicate $edge/2$, source and sink nodes encoded by predicates $source/1$ and $sink/1$, the following program assigns a direction to each edge so that source nodes can still reach all sink nodes:

$$1 \leq \{arc(X,Y);\ arc(Y,X)\} \leq 1 \leftarrow edge(X,Y) \tag{4}$$
$$reach(X,X) \leftarrow source(X) \tag{5}$$
$$reach(X,Y) \leftarrow reach(X,Z),\ arc(Z,Y) \tag{6}$$
$$\perp \leftarrow source(X),\ sink(Y),\ not\ reach(X,Y) \tag{7}$$

If failures on the reachability condition are permitted up to a given threshold encoded by predicate $threshold/1$, the program comprising rules (4)–(6) and

$$fail(X,Y) \leftarrow source(X),\ sink(Y),\ not\ reach(X,Y) \tag{8}$$
$$\perp \leftarrow threshold(T), sum\{1,X,Y : fail(X,Y)\} > T \tag{9}$$

can be used. Note that X and Y are local variables in rule (9), and all other variables are global. ■

A substitution σ is a partial function from variables to constants; the application of σ to an expression E is denoted by $E\sigma$. Let $instantiate(\Pi)$ be the program obtained from rules of Π by substituting global variables with constants in $\mathbf{C}$, in all possible ways; note that local variables are still present in $instantiate(\Pi)$. The Herbrand base of Π, denoted $base(\Pi)$, is the set of ground atoms (i.e., atoms with no variables) occurring in $instantiate(\Pi)$.

Example 2. Let Π_{run} comprise rules (4)–(6), (8)–(9) and the facts (i.e., rules with an empty body) $edge(a,b)$, $edge(a,d)$, $edge(d,c)$, $source(a)$, $source(b)$, $sink(c)$, and $threshold(0)$ (see Figure 1). Hence, $instantiate$
(Π_{run}) contains, among others, the rules

$$1 \leq \{arc(a,b);\ arc(b,a)\} \leq 1 \leftarrow edge(a,b)$$
$$\perp \leftarrow threshold(0), sum\{1,X,Y : fail(X,Y)\} > 0$$

and $base(\Pi_{run})$ contains $fail(a,c), fail(b,c)$, and so on. ■

A *(two-valued) interpretation* is a set of ground atoms. For a two-valued interpretation I, relation $I \models \cdot$ is defined as follows: for a ground atom $p(\overline{c})$, $I \models p(\overline{c})$ if $p(\overline{c}) \in I$, and $I \models not\ p(\overline{c})$ if $p(\overline{c}) \notin I$; for an aggregate α of the form (1), the aggregate set of α w.r.t. I, denoted $aggset(\alpha,I)$, is $\{\langle t_a, \overline{t'}\rangle\sigma \mid p(\overline{t})\sigma \in I$, for some substitution $\sigma\}$, and $I \models \alpha$ if $(\sum_{\langle c_a,\overline{c'}\rangle \in aggset(\alpha,I)} c_a) \odot t_g$ is a true expression over integers; for a choice α of the form (2), $I \models \alpha$ if $t_1 \leq |I \cap atoms| \leq t_2$ is a true expression over integers; for a rule r with no global variables, $I \models B(r)$ if $I \models \alpha$ for all $\alpha \in B(r)$, and $I \models r$ if $I \models H(r)$ whenever $I \models B(r)$; for a program Π, $I \models \Pi$ if $I \models r$ for all $r \in instantiate(\Pi)$.

For a rule r of the form (3) and an interpretation I, let $expand(r,I)$ be the set $\{p(\overline{c}) \leftarrow body \mid p(\overline{c}) \in I$ occurs in $H(r)\}$. The *reduct* of Π w.r.t. I is the program comprising the expanded rules of $instantiate(\Pi)$ whose body is true w.r.t. I, that is, $reduct(\Pi,I) := \bigcup_{r \in instantiate(Pi),\ I \models B(r)} expand(r,I)$. An *answer set* of Π is an interpretation A such that $A \models \Pi$ and no $I \subset A$ satisfies $I \models reduct(\Pi,A)$.

Example 3. The only answer set A_{run} of program Π_{run} contains, among others, the atoms $arc(b,a)$, $arc(a,d)$, $arc(d,c)$, no other instance of $arc/2$, and no instance of $fail/2$. Hence, $A_{run} \models 1 \leq \{arc(a,b); arc(b,a)\} \leq 1$ and $A_{run} \not\models sum\{1,X,Y : fail(X,Y)\} > 0$. ■

A *three-valued interpretation* is a pair (L,U), where L,U are sets of ground atoms such that $L \subseteq U$; sets L and U, also denoted $(L,U)_1$ and $(L,U)_2$, are the lower and upper bounds on the true atoms, so atoms in L are true, atoms in $U \setminus L$ are undefined, and all other atoms are false. The *evaluation function* $[\![\cdot]\!]_L^U$ associates literals and aggregates with a truth value among $\mathbf{u}$, $\mathbf{t}$ and $\mathbf{f}$ as follows: $[\![\alpha]\!]_L^U = \mathbf{u}$ if α is a literal whose atom is $p(\overline{c})$ and $p(\overline{c}) \in U \setminus L$, or α is an aggregate of the form (1) and $aggset(\alpha, U \setminus L) \neq \emptyset$, or α is a choice of the form (2) and $(U \setminus L) \cap atoms \neq \emptyset$; $[\![\alpha]\!]_L^U = \mathbf{t}$ if $[\![\alpha]\!]_L^U \neq \mathbf{u}$ and $L \models \alpha$; and $[\![\alpha]\!]_L^U = \mathbf{f}$ if $[\![\alpha]\!]_L^U \neq \mathbf{u}$ and $L \not\models \alpha$. The evaluation function extends to rule bodies as follows: $[\![B(r)]\!]_L^U = \mathbf{f}$ if there is $\alpha \in B(r)$ such that $[\![\alpha]\!]_L^U = \mathbf{f}$; $[\![B(r)]\!]_L^U = \mathbf{t}$ if $[\![\alpha]\!]_L^U = \mathbf{t}$ for all $\alpha \in B(r)$; otherwise $[\![B(r)]\!]_L^U = \mathbf{u}$.

Example 4. For α being $sum\{1,X,Y : fail(X,Y)\} > 0$, $[\![\alpha]\!]_{\emptyset}^{\{fail(a,c)\}} = \mathbf{u}$, $[\![\alpha]\!]_{\{fail(a,c)\}}^{\{fail(a,c)\}} = \mathbf{t}$, and $[\![\alpha]\!]_{\emptyset}^{\emptyset} = \mathbf{f}$. ■

Mainstream ASP systems compute answer sets of a given program Π by applying several inference rules on (a subset of) $instantiate(\Pi)$, the most relevant ones for this work summarized below. Let (L,U) be a three-valued interpretation, and $p(\overline{c})$ be a ground atom such that $[\![p(\overline{c})]\!]_L^U = \mathbf{u}$. Atom $p(\overline{c})$ in $H(r)$ is *inferred true by support* if $[\![B(r)]\!]_L^U = \mathbf{t}$. (Actually, if $H(r)$ is a choice of the form (2), inference by support additionally requires that $|atoms \cap U| = t_1$, that is, undefined atoms in $atoms \cap U$ are required to reach the bound t_1. Such extra condition is not relevant for our work, and will not be used, because our explanations aim at associating true atoms with rules with true bodies.) Atom $p(\overline{c})$ is *inferred false by lack of support* if each rule $r \in instantiate(\Pi)$ with $p(\overline{c})$ occurring in $H(r)$ is such that $[\![B(r)]\!]_L^U = \mathbf{f}$. Atom $p(\overline{c})$ is *inferred false by a constraint-like rule* $r \in instantiate(\Pi)$ if $p(\overline{c}) \in B^+(r)$, $[\![H(r)]\!]_L^U = \mathbf{f}$ and $[\![B(r) \setminus \{p(\overline{c})\}]\!]_L^U = \mathbf{t}$. Atom $p(\overline{c})$ is *inferred false by a choice rule* $r \in instantiate(\Pi)$ if $H(r)$ has the form (2), $p(\overline{c}) \in atoms$, $|atoms \cap L| \geq t_2$ and $[\![B(r)]\!]_L^U = \mathbf{t}$. Atom $p(\overline{c})$ is *inferred false by well-founded computation* if it belongs to some *unfounded set* X for Π w.r.t. (L,U), that is, a set X such that for all rules $r \in instantiate(\Pi)$ at least one of the following conditions holds: (i) no atom from X occurs in $H(r)$; (ii) $[\![B(r)]\!]_L^U = \mathbf{f}$; (iii) $B^+(r) \cap X \neq \emptyset$.

Example 5. Given the program $instantiate(\Pi_{run})$, and the three-valued interpretation $(\emptyset, base(\Pi_{run}))$, atom $edge(a,a)$ is inferred false by lack of support, atom $source(a)$ is inferred true by support, and the set $\{edge(a,a), arc(a,a)\}$ is unfounded. Given $(\{arc(d,c)\}, base(\Pi_{run}) \setminus \{reach(a,c)\})$, atom $reach(a,d)$ is inferred false by the constraint-like rule (6), and $arc(c,d)$ is inferred false by the choice rule (4). ■

3 Explanations

Let Π be a program, and A be one of its answer sets. A *well-founded derivation* for Π w.r.t. A, denoted $wf(\Pi,A)$, is obtained from the interpretation $(\emptyset, base(\Pi))$ by iteratively (i) adding to its lower bound atoms of A that are inferred true by support, and (ii) removing from its upper bound atoms belonging to some unfounded set. Note that $wf(\Pi,A)$ is computed as a preprocessing step.

Example 6. Given Π_{run} and A_{run} from Examples 2–3, the lower bound of $wf(\Pi_{run}, A_{run})$ contains head atoms in Example 2, $arc(b,a)$, $arc(a,d)$, $arc(d,c)$, $reach(a,a)$, $reach(b,b)$, $reach(a,d)$, $reach(a,c)$, $reach(b,a)$, $reach(b,c)$, and $reach(b,d)$. The upper bound additionally contains $arc(a,b)$, $arc(d,a)$, $arc(c,d)$, and several instances of $reach/2$ and $fail/2$. ■

An *explaining derivation* for Π and A from (L,U) is obtained by iteratively (i) adding to L atoms of A that are inferred true by support, and (ii) removing from U atoms that are inferred false by lack of

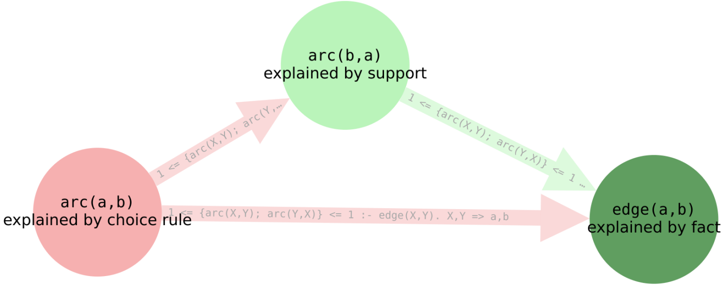


Figure 2: Induced DAG on the vertices reachable from $arc(a,b)$ for the minimal assumption set $\emptyset$ for Π_{run}.

support, constraint-like rules and choice rules. An *assumption set* for Π and A is a set $X \subseteq base(\Pi) \setminus A$ of ground atoms such that the explaining derivation for Π and A from $(\emptyset, wf(\Pi,A)_2 \setminus X)$ terminates with A (in words, A is reconstructed from the false atoms of the well-founded derivation extended with X). Let $AS(\Pi,A)$ be the set of assumption sets for Π and A. A *minimal assumption set* for Π, A and a ground atom α is a set $X \in AS(\Pi,A)$ such that $X' \subset X$ implies $X' \notin AS(\Pi,A)$, and $\alpha \in X$ implies $\alpha \in X'$ for all $X' \in AS(\Pi,A)$. (In other words, we prefer assumption sets not including the atom to explain. When all assumption sets include the atom to explain, we opt for the singleton comprising the atom to explain alone.) Let $MAS(\Pi,A,\alpha)$ be the set of minimal assumption sets for Π, A and α.

Example 7. Set $base(\Pi_{run}) \setminus A_{run}$ is an assumption set for Π_{run} and its answer set A_{run}. It can be checked that also $\emptyset \in AS(\Pi_{run}, A_{run}, \alpha)$, and it is indeed the only minimal assumption set in this case, for any atom in $base(\Pi_{run})$. ■

Given an assumption set X and an explaining derivation from $(\emptyset, wf(\Pi,A)_2 \setminus X)$, a directed acyclic graph (DAG) can be obtained as follows: The vertices of the graph are the atoms in $base(\Pi)$ and the aggregates occurring in $instantiate(\Pi)$. (The vertex $p(\overline{c})$ is also referred to as *not* $p(\overline{c})$.) Any aggregate of the form (1) is linked to instances of $p(\overline{t})$. Atoms inferred true by support due to a rule $r \in instantiate(\Pi)$ are linked to elements of $B(r)$. Any atom α inferred false by lack of support is linked to an element of $B(r)$ that is inferred false before α, for each rule $r \in instantiate(\Pi)$ such that α occurs in $H(r)$. Any atom α inferred false by a constraint-like rule $r \in instantiate(\Pi)$ is linked to the atoms occurring in $H(r)$ and the elements of $B(r) \setminus \{\alpha\}$. Any atom α inferred false by a choice rule $r \in instantiate(\Pi)$ is linked to the atoms occurring in $H(r)$ that are true in A, and to the elements of $B(r)$. A portion of an example DAG is reported in Figure 2.

4 Existence of Minimal Assumption Sets

This section is devoted to formally show that the existence of minimal assumption sets is guaranteed, and so are DAGs as defined in Section 3.

Theorem 1 (Main Theorem)**.** *Let Π be a program, A one of its answer sets, and α a ground atom in $base(\Pi)$. Set $MAS(\Pi,A,\alpha)$ is nonempty.*

To prove the above theorem, we introduce some additional notation and claims. Let Π be a program, and (L,U) be a three-valued interpretation. We denote by $\Pi, L, U \vdash \alpha$ the fact that $\alpha \in base(\Pi)$ is inferred true by support, which is the case when $[\![\alpha]\!]_L^U = \mathbf{u}$, and there is $r \in instantiate(\Pi)$ such that α occurs in $H(r)$ and $[\![B(r)]\!]_L^U = \mathbf{t}$, as defined in Section 2. Similarly, we denote by $\Pi, L, U \vdash not\ \alpha$ the fact that $\alpha \in base(\Pi)$ is inferred false by lack of support, constraint-like rules and choice rules, which is the case when $[\![\alpha]\!]_L^U = \mathbf{u}$, and one of the following conditions holds: each rule $r \in instantiate(\Pi)$ with α

occurring in $H(r)$ is such that $[\![B(r)]\!]_L^U = \mathbf{f}$; there is $r \in \mathit{instantiate}(\Pi)$ with $\alpha \in B^+(r)$, $[\![H(r)]\!]_L^U = \mathbf{f}$ and $[\![B(r) \setminus \{\alpha\}]\!]_L^U = \mathbf{t}$; there is $r \in \mathit{instantiate}(\Pi)$ with $H(r)$ of the form (2), $\alpha \in \mathit{atoms}$, $|\mathit{atoms} \cap L| \geq t_2$ and $[\![B(r)]\!]_L^U = \mathbf{t}$.

The *explaining derivation operator* D_Π is defined as

$$\begin{aligned} D_\Pi(L,U) := (&L \cup \{\alpha \in \mathit{base}(\Pi) \mid \Pi, L, U \vdash \alpha\},\\ &U \setminus \{\alpha \in \mathit{base}(\Pi) \mid \Pi, L, U \vdash \mathit{not}\ \alpha\}). \end{aligned}$$

Let $(L,U) \sqsubseteq (L',U')$ denote the fact that $L \subseteq L' \subseteq U' \subseteq U$, i.e., everything that is true w.r.t. (L,U) is true w.r.t. (L',U'), and everything that is false w.r.t. (L,U) is false w.r.t. (L',U').

Lemma 1. *Operator D_Π is monotonic w.r.t. $\sqsubseteq$.*

Proof. For $(L,U) \sqsubseteq (L',U')$, we shall show that $D_\Pi(L,U) \sqsubseteq D_\Pi(L',U')$ holds. For $\alpha \in D_\Pi(L,U)_1 \setminus L$ such that $\alpha \notin L'$, we have $\Pi, L, U \vdash \alpha$, that is, there is $r \in \mathit{instantiate}(\Pi)$ such that α occurs in $H(r)$ and $[\![B(r)]\!]_L^U = \mathbf{t}$. As $(L,U) \sqsubseteq (L',U')$, we have that $[\![B(r)]\!]_{L'}^{U'} = \mathbf{t}$, that is, $\Pi, L', U' \vdash \alpha$ holds, and therefore $\alpha \in D_\Pi(L',U')_1 \setminus L$.

For $\alpha \in U \setminus D_\Pi(L,U)_2$ such that $\alpha \in U'$, we have $\Pi, L, U \vdash \mathit{not}\ \alpha$, and therefore we have three cases:

1. Each rule $r \in \mathit{instantiate}(\Pi)$ with α occurring in $H(r)$ is such that $[\![B(r)]\!]_L^U = \mathbf{f}$. As $(L,U) \sqsubseteq (L',U')$, $[\![B(r)]\!]_{L'}^{U'} = \mathbf{f}$ holds.
2. There is $r \in \mathit{instantiate}(\Pi)$ with $\alpha \in B^+(r)$, $[\![H(r)]\!]_L^U = \mathbf{f}$ and $[\![B(r) \setminus \{\alpha\}]\!]_L^U = \mathbf{t}$. As $(L,U) \sqsubseteq (L',U')$, $[\![H(r)]\!]_{L'}^{U'} = \mathbf{f}$ and $[\![B(r) \setminus \{\alpha\}]\!]_{L'}^{U'} = \mathbf{t}$.
3. There is $r \in \mathit{instantiate}(\Pi)$ with $H(r)$ of the form (2), $\alpha \in \mathit{atoms}$, $|\mathit{atoms} \cap L| \geq t_2$ and $[\![B(r)]\!]_L^U = \mathbf{t}$. As $(L,U) \sqsubseteq (L',U')$, $|\mathit{atoms} \cap L'| \geq t_2$ and $[\![B(r)]\!]_{L'}^{U'} = \mathbf{t}$.

In any case, $\Pi, L', U' \vdash \alpha$ holds, and therefore $\alpha \in U' \setminus D_\Pi(L',U')_2$. $\square$

Lemma 2. *$L \subseteq A \subseteq U$ implies $D_\Pi(L,U)_1 \subseteq A \subseteq D_\Pi(L,U)_2$.*

Proof. For $\alpha \in D_\Pi(L,U)_1 \setminus L$ we have $\Pi, L, U \vdash \alpha$, that is, there is $r \in \mathit{instantiate}(\Pi)$ such that $[\![B(r)]\!]_L^U = \mathbf{t}$. Hence, $A \models B(r)$, and therefore $\mathit{expand}(r,A) \subseteq \mathit{reduct}(\Pi,A)$. In particular, $\alpha \leftarrow B(r)$ belongs to the reduct, and therefore $\alpha \in A$.

For $\alpha \in U \setminus D_\Pi(L,U)_2$ we have $\Pi, L, U \vdash \mathit{not}\ \alpha$ and we have to show that $\alpha \notin A$. Three cases:

1. Each rule $r \in \mathit{instantiate}(\Pi)$ with α occurring in $H(r)$ is such that $[\![B(r)]\!]_L^U = \mathbf{f}$.
2. There is $r \in \mathit{instantiate}(\Pi)$ with $\alpha \in B^+(r)$, $[\![H(r)]\!]_L^U = \mathbf{f}$ and $[\![B(r) \setminus \{\alpha\}]\!]_L^U = \mathbf{t}$.
3. There is $r \in \mathit{instantiate}(\Pi)$ with $H(r)$ of the form (2), $\alpha \in \mathit{atoms}$, $|\mathit{atoms} \cap L| \geq t_2$ and $[\![B(r)]\!]_L^U = \mathbf{t}$.

In the first case, $A \setminus \{\alpha\} \models \mathit{reduct}(\Pi,A)$, and therefore $A \setminus \{\alpha\} = A$ because A is an answer set of Π. In the other two cases, $\alpha \notin A$ because $A \models \Pi$ by assumption. $\square$

The explaining derivation from (L,U) is obtained as the fix point of the sequence $(L_0,U_0) := (L,U)$, $(L_{i+1},U_{i+1}) := D_\Pi(L_i,U_i)$ for $i \geq 0$. Note that the fix point is reached in at most $|\mathit{base}(\Pi)|$ steps because of Lemma 1 and each application of D_Π reduces the undefined atoms (or is a fix point). Thus, the system eventually terminates in at most $|\mathit{base}(\Pi)|$ steps.

Lemma 3. *For any answer set A of Π, set $\mathit{base}(\Pi) \setminus A$ is an assumption set for Π and A.*

Proof. Let (L,U) be the explaining derivation from $(\emptyset, base(\Pi) \setminus A)$. Thanks to Lemma 2, it is sufficient to show that $p(\overline{c}) \in A$ implies $p(\overline{c}) \in L$. Let us consider a topological ordering $C_1, \ldots, C_n$ ($n \geq 1$) for the strongly connected components of $\mathscr{G}_\Pi$, and let $p \in C_i$. We use induction on i. Since $p(\overline{c}) \in A$, there must be $r \in reduct(\Pi, A)$ such that $H(r) = p(\overline{c})$ and $A \models B(r)$. Hence, $[\![B^-(r)]\!]_L^U = \mathbf{t}$. Moreover, $[\![B^\Sigma(r)]\!]_L^U = \mathbf{t}$, either because $i = 1$ and $B^\Sigma(r) = \emptyset$, or because of the induction hypothesis. Therefore, to have $\alpha \notin L$, it must be the case that $[\![B^+(r)]\!]_L^U \neq \mathbf{t}$ for all such rules, but in this case $L \models reduct(\Pi, A)$, a contradiction with the assumption that A is an answer set of Π. □

Given Lemma 3, the proof of Main Theorem is immediate by the definition of $MAS(\Pi, A, \alpha)$ as following:

Proof of Main Theorem. By definition, a minimal assumption set for Π, A and α is a set $X \in AS(\Pi, A)$ such that $X' \subset X$ implies $X' \notin AS(\Pi, A)$, and $\alpha \in X$ implies $\alpha \in X'$ for all $X' \in AS(\Pi, A)$. Lemma 3 guarantees the existence of an assumption set for Π and A. Existence of a minimal assumption set for Π, A and α is therefore guaranteed. □

5 Generation via Meta-Programming

By leveraging ASP systems, the concepts introduced in Section 3 can be computed. A meta-programming approach is presented in this section, where the full language of ASP is used, including constructs omitted in the previous sections, like weak constraints, uninterpreted functions, conditional literals and @-terms. The reader is referred to [5] for details. We will use the name *ASP programs* for encodings using the full language of ASP, in contrast to the name *program* that we use for encodings using the restricted syntax introduced in Section 2.

Program Π, answer set A and the atom to explain are encoded by a set of facts obtained by computing the unique answer set of the ASP program $serialize(\Pi, A, \alpha)$, defined next. Each atom $p(\overline{c})$ in $base(\Pi)$ is encoded by a fact `atom(`$p(\overline{c})$`)`; moreover, the encoding includes a fact `true(`$p(\overline{c})$`)` if $p(\overline{c}) \in A$, and `false(`$p(\overline{c})$`)` otherwise; additionally, if $p(\overline{c})$ is false in $wf(\Pi, A)$, the encoding includes a fact `explained_by(`$p(\overline{c})$`, initial_well_founded)`. As for α, the encoding includes a fact `explain(`α`)`. Each rule r of $instantiate(\Pi)$ is encoded by

`rule(`$id(\overline{X})$`) :- atom(`$p_1(\overline{t_1})$`), ..., atom(`$p_n(\overline{t_n})$`).`

where id is an identifier for r, $\overline{X}$ are the global variables of r, and $B^+(r) = \{p_i(\overline{t_i}) \mid i = 1, \ldots, n\}$; moreover, the encoding includes

`head(`$id(\overline{X})$`,`$p(\overline{t})$`) :- rule(`$id(\overline{X})$`).`
`pos_body(`$id(\overline{X})$`,`$p'(\overline{t'})$`) :- rule(`$id(\overline{X})$`).`
`neg_body(`$id(\overline{X})$`,`$p''(\overline{t''})$`) :- rule(`$id(\overline{X})$`).`

for each $p(\overline{t})$ occurring in $H(r)$, $p'(\overline{t'}) \in B^+(r)$ and $p''(\overline{t''}) \in B^{(}r)$; additionally, for each aggregate α of the form (1) in $B^\Sigma(r)$, the encoding includes

`pos_body(`$id(\overline{X})$`,`$agg(\overline{X})$`) :- rule(`$id(\overline{X})$`).`
`aggregate(`$agg(\overline{X})$`) :- rule(`$id(\overline{X})$`).`
`true(`$agg(\overline{X})$`) :- rule(`$id(\overline{X})$`),`
    `#sum{`t_a`,`$\overline{t'}$` : true(`$p(\overline{t})$`)}`$\odot t_g$`.`
`false(`$agg(\overline{X})$`):- rule(`$id(\overline{X})$`), not true(`$agg(\overline{X})$`).`

`rule(`$agg(\overline{X})$`):- aggregate(`$agg(\overline{X})$`),true(`$agg(\overline{X})$`).`

head($agg(\overline{X})$,$agg(\overline{X})$) :- rule($agg(\overline{X})$).
pos_body($agg(\overline{X})$,$p(\overline{t})$):- rule($agg(\overline{X})$),true($p(\overline{t})$).
neg_body($agg(\overline{X})$,$p(\overline{t})$):-rule($agg(\overline{X})$),false($p(\overline{t})$).

rule(($agg(\overline{X})$,$p(\overline{t})$)) :- aggregate($agg(\overline{X})$), false($agg(\overline{X})$), atom($p(\overline{t})$).
head(($agg(\overline{X})$,$p(\overline{t})$),$agg(\overline{X})$):- rule(($agg(\overline{X})$,$p(\overline{t})$)).
pos_body(($agg(\overline{X})$,$p(\overline{t})$),$p(\overline{t})$) :- rule(($agg(\overline{X})$,$p(\overline{t})$)), false($p(\overline{t})$).
neg_body(($agg(\overline{X})$,$p(\overline{t})$),$p(\overline{t})$) :- rule(($agg(\overline{X})$,$p(\overline{t})$)), true($p(\overline{t})$).

where agg is an identifier for α; finally, if $H(r)$ is a choice of the form (2), the encoding includes

choice($id(\overline{X})$,t_1,t_2) :- rule($id(\overline{X})$).

Note that a true ground aggregate of the form (1) identified by $agg(\overline{c})$ is associated with a single rule whose body becomes true after all instances of $p(\overline{t})$ are assigned the truth value they have in the answer set A; on the other hand, a false aggregate is associated with one rule for each instance of $p(\overline{t})$, whose bodies becomes false when instances of $p(\overline{t})$ are assigned the truth value they have in the answer set A.

Example 8. Recall Π_{run} and A_{run} from Examples 2–3. The ASP program $serialize(\Pi_{run}, A, arc(a,b))$ includes

```
atom(edge(a,b)).  atom(arc(b,a)).  atom (arc(a,b)).  explain(arc(a,b)).
true(edge(a,b)).  true(arc(b,a)).  false(arc(a,b)).

rule(r4(X,Y)) :- atom(edge(X,Y)).
choice(r4(X,Y),1,1) :- rule(r4(X,Y)).
head(r4(X,Y), arc(X,Y)) :- rule(r4(X,Y)).
head(r4(X,Y), arc(Y,X)) :- rule(r4(X,Y)).
pos_body(r4(X,Y), edge(X,Y)):- rule(r4(X,Y)).

aggregate(agg1(T)) :- rule(r9(T)).
true(agg1(T)) :- rule(r9(T)), #sum{1,X,Y : true(fail(X,Y))} > T.
```

and several other rules. The answer set of $serialize(\Pi_{run}, A, arc(a,b))$ includes, among other atoms, `aggregate(agg1(0))` and `false(agg1(0))`. ■

The ASP program Π_{MAS} reported in Figure 3, coupled with a fact for each atom in the answer set of $serialize(\blacksquare, A, \alpha)$, has optimal answer sets corresponding to cardinality-minimal elements in $MAS(\Pi, A, \alpha)$. Intuitively, line 1 guesses the assumption set, line 2–3 minimizes the size of the assumption set (preferring to not assume the falsity of the atom to explain), and lines 4–5 impose that each atom must have exactly one explanation. The other rules encode the explaining derivation for Π and A from $wf(\Pi, A) \setminus X$, where X is the guessed assumption set.

Given a minimal assumption set encoded by predicate `assume_false/1`, an explaining derivation can be computed by removing lines 1–3 from the ASP program Π_{MAS}. Let Π_{EXP} be such an ASP program. Finally, given an explaining derivation encoded by `explained_by(Index,Atom,Reason)`, with the additional `Index` argument encoding the order in the sequence, a DAG linking atoms according to the derivation can be computed by the ASP program Π_{DAG} reported in Figure 4.

Example 9. Let Π_S have a fact for each atom in the answer set of $serialize(\Pi_{run}, A_{run}, arc(a,b))$. $\Pi_{MAS} \cup \Pi_S$ generates the empty assumption set. $\Pi_{EXP} \cup \Pi_S \cup \emptyset$ generates an explaining derivation, for example one including $explained\_by(edge(a,b), (support, r6))$, $explained\_by(arc(b,a), (support, r1(a,b)))$ and $explained\_by(arc(a,b), (choice\_rule, r1(a,b)))$. Let Π_E have a fact for each instance of $explained\_by/3$

```
{assume_false(Atom)} :- false(Atom), not aggregate(Atom).
:~ false(Atom), assume_false(Atom), not explain(Atom). [1@1, Atom]
:~ false(Atom), assume_false(Atom), explain(Atom). [1@2, Atom]

has_explanation(Atom) :- explained_by(Atom,_).
:- atom(X), #count{Reason: explained_by(Atom,Reason)} != 1.

explained_by(Atom, assumption) :- assume_false(Atom).

{explained_by(Atom, (support, Rule))} :- head(Rule,Atom), true(Atom);
   true (BAtom) : pos_body(Rule,BAtom); has_explanation(BAtom) : pos_body(Rule,BAtom);
   false(BAtom) : neg_body(Rule,BAtom); has_explanation(BAtom) : neg_body(Rule,BAtom).

{explained_by(Atom, lack_of_support)} :- false(Atom); false_body(Rule) : head(Rule,Atom).
false_body(Rule) :- rule(Rule); pos_body(Rule,BAtom), false(BAtom), has_explanation(BAtom).
false_body(Rule) :- rule(Rule); neg_body(Rule,BAtom), true(BAtom), has_explanation(BAtom).

{explained_by(Atom, (required_to_falsify_body, Rule))} :- false(Atom), not aggregate(Atom);
   pos_body(Rule,Atom), false_head(Rule); true(BAtom) : pos_body(Rule,BAtom), BAtom != Atom;
   has_explanation(BAtom) : pos_body(Rule,BAtom), BAtom != Atom;
   false(BAtom) : neg_body(Rule,BAtom); has_explanation(BAtom) : neg_body(Rule,BAtom).
explained_head(Rule) :- rule(Rule); has_explanation(HAtom) : head(Rule,HAtom).
false_head(Rule) :- explained_head(Rule), not choice(Rule,_,_);
   false(HAtom) : head(Rule,HAtom).
false_head(Rule) :- explained_head(Rule), choice(Rule, LowerBound, UpperBound);
   not LowerBound <= #count{HAtom' : head(Rule,HAtom'), true(HAtom')} <= UpperBound.

{explained_by(Atom, (choice_rule, Rule))} :- false(Atom);
   head(Rule,Atom), choice(Rule, LowerBound, UpperBound);
   true(BAtom) : pos_body(Rule,BAtom); has_explanation(BAtom) : pos_body(Rule,BAtom);
   false(BAtom) : neg_body(Rule,BAtom); has_explanation(BAtom) : neg_body(Rule,BAtom);
   #count{HAtom : head(Rule, HAtom), true(HAtom), has_explanation(HAtom)} = UpperBound.
```

Figure 3: ASP program Π_{MAS} for computing a minimal assumption set

in the explaining derivation. $\Pi_{DAG} \cup \Pi_S \cup \Pi_E$ generates a DAG, for example one including $link(arc(b,a), edge(a,b))$, $link(arc(a,b), arc(b,a))$ and $link(arc(a,b), edge(a,b))$. ■

6 Implementation and Experiment

We deployed an XAI system for ASP named `xASP2` , which is powered by the `clingo python api` [8]. By taking an ASP program Π, one of its answer sets A, and an atom α as input, `xASP2` is capable of producing minimal assumption sets, explaining derivations, and DAGs as output to assist the user in determining the assignment of α. The source code is available at https://github.com/alviano/xasp and an example DAG is given at https://xasp-navigator.netlify.app/.

The pipeline implemented by `xASP2` starts with the serialization of the input data, which is obtained by means of an ASP program crafted from the abstract syntax tree of Π and whose answer set identifies the relevant portion of $instantiate(\Pi)$ and $base(\Pi)$. In a nutshell, ground atoms provided by the user, $A \cup \{\alpha\}$, are part of $base(\Pi)$ and used to instantiate rules of Π (by matching positive body literals), which in turn may extend $base(\Pi)$ with other ground atoms occurring in the instantiated rules; possibly, some atoms of $base(\Pi)$ of particular interest can be explicitly provided by the user. Aggregates are also processed automatically by means of an ASP program, and so is the computation of false atoms in the well-founded derivation $wf(\Pi, A)$.

Obtained $serialize(\Pi, A, \alpha)$, `xASP2` proceeds essentially as described in Section 5, by computing a minimal assumption set, an explaining derivation and an explanation DAG. As an additional opti-

```
link(Atom, BAtom) :- explained_by(_, Atom, (support, Rule)); pos_body(Rule, BAtom).
link(Atom, BAtom) :- explained_by(_, Atom, (support, Rule)); neg_body(Rule, BAtom).

{link(Atom, A) : pos_body(Rule,A), false(A), explained_by(I,A,_), I < Index;
 link(Atom, A) : neg_body(Rule,A), true (A), explained_by(I,A,_), I < Index} = 1 :-
   explained_by(_, Atom, lack_of_support); head(Rule, Atom).

link(Atom,A) :- explained_by(_, Atom, (required_to_falsify_body, Rule)); head(Rule,A).
link(At,A) :- explained_by(_,At,(required_to_falsify_body, Rule)); pos_body(Rule,A), A!=At.
link(Atom,A) :- explained_by(_,Atom,(required_to_falsify_body, Rule)); neg_body(Rule,A).

link(Atom, HAtom) :- explained_by(_,Atom,(choice_rule, Rule)); head(Rule,HAtom), true(HAtom).
link(Atom, BAtom) :- explained_by(_, Atom, (choice_rule, Rule)); pos_body(Rule, BAtom).
link(Atom, BAtom) :- explained_by(_, Atom, (choice_rule, Rule)); neg_body(Rule, BAtom).
```

Figure 4: ASP program Π_{DAG} for computing a directed acyclic graph associated with an explaining derivation

mization, the explaining derivation is shrunk to the atoms reachable from α, utilizing an ASP program. Finally, the user can opt for a few additional steps: obtain a graphical representation by means of the `igraph` network analysis package (https://igraph.org/); obtain an interactive representation in https://xasp-navigator.netlify.app/; ask for different minimal assumption sets, explaining derivations and DAGs.

We assessed `xASP2` empirically on the commercial application. The ASP program comprises 420 rules and 651 facts. After grounding, there are 4261 ground rules and 4468 ground atoms. The program was expected to have a unique answer set, but two answer sets were actually computed. Our experiment was run on an Intel Core i7-1165G7 @2.80 GHz and 16 GB of RAM. `xASP2` computed a DAG for the unexpected true atom, `behaves_inertially(testing_posTestNeg,121)`, in 14.85 seconds on average, over 10 executions. The DAG comprises 87 links, 45 internal nodes and 20 leaves, only one of which is explained by assumption; only 30 of the 420 symbolic rules and 11 of the 651 facts are involved in the DAG; at the ground level, only 48 of the 4261 ground rules and 65 of the 4468 ground atoms are involved. Additionally, we repeated the experiment on 10 randomly selected atoms with respect to two different answer sets, repeating each test case 10 times. We measured an average runtime of 14.79 seconds, with a variance of 0.004 seconds.

Table 1: The action preconditions and effects in Blockworlds problem

Action	**Precondition**	**Effects**
$stack(X,Y)$ - stack block X is on block Y	Block Y is clear The agent holds the block X	X is clear X is on Y Y is no longer clear The agent does not hold anything
$unstack(X,Y)$ - unstack block X is on block Y	X is clear X is on Y The agent does not hold anything	The agent holds the block X Y becomes clear X is not clear
$pickup(X)$ - pickup block X from the table	X is clear X is on the table the agent does not hold anything	The agent holds the block X X is no longer on the table and is not clear
$putdown(X)$ - put down block X onto the table	The agent holds the block X	X is clear X is on the table the agent does not hold anything

`xASP2` also has the ability to handle explainable planning, meaning it can generate an explanation

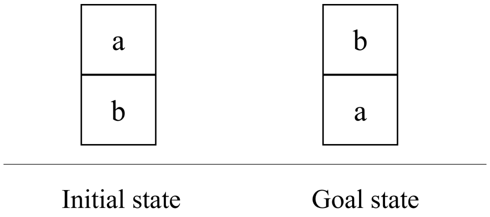


Figure 5: The initial and goal states of Blocksworld.

```
1 h(X,T+1) :- action(action(A)),occurs(A,T), postcondition(action(A),
     effect(unconditional),X,value(X,true)).
2 -h(X,T+1) :- action(action(A)),occurs(A,T), postcondition(action(A),
     effect(unconditional),X,value(X,false)).
3 h(X,T+1) :- h(X,T), not -h(X,T+1).
4 -h(X,T+1) :- -h(X,T), not h(X,T+1).
5 non_exec(A,T) :- action(action(A)), not h(X,T), precondition(action(A),X,value(X, true)).
6 non_exec(A,T) :- action(action(A)), not -h(X,T), precondition(action(A),X,value(X, false)).
7 :- action(action(A)),occurs(A,T), non_exec(A,T).
```

Figure 6: ASP program for reasoning about effects of actions [12]

graph showing why a particular action cannot take place at a certain time. To demonstrate this capability, we will use a popular problem known as Blocksworld. The initial state (left) and goal state (right) of the problem are shown in Figure 5. Five fluents are $on(X,Y)$ - block X is on block Y, $onTable(X)$ - block X is on the table, $clear(X)$ - block X is clear, $holding(X)$ - the agent holds the block X, and $handEmpty$ - the agent does not hold anything. Four differnt actions are $stack$, $unstack$, $pickup$ and $putdown$. The domain description of the problem is shown in Table 1 in which the predictions and effects of four actions are presented.

The rules for reasoning about effects of actions, action generation and goal enforcement [12] are utilized as programming input in `xASP2` . Figure 6 shows the ASP program for reasoning about the effects of actions in which an action occurs only when its preconditions are true and then its effects are true in the next time step. Specifically, lines 5 and 6 are used to define states in which an action cannot be executed, and constraint is employed to prevent non-executable actions from occurring (line 7).

For the problem described in Figure 5, executing the actions of $unstack(a,b)$, $putdow(a)$, $pickup(b)$ and $stack(b,a)$ at times 0, 1, 2, and 3 respectively constitutes the optimal plan (assuming time starts at 0). However, if users are in a rush and want to put down block a on the table at time 0, as represented by the atom $occurs((\text{''}putdown\text{''}, constant(\text{''}a\text{''})), 0)$, they will encounter a false occurrence of the action $putdown(a)$. Figure 6 shows that atom $occurs((\text{''}putdown\text{''}, constant(\text{''}a\text{''})), 0)$ is false because the constraint rule prevents its execution and the prediction of the action holding block a is invalid/false.

7 Related Work

Our work, as stated in the introduction, is situated within the realm of XAI and can be used as a debugging tool to provide explanations for an unexpected result. For instance, if an element α is false in all answer sets of a program Π despite user expectations, our system can help identify which rules are contributing to this anomalous behavior. Thus, in this section, we will explore both debugging tools for ASP and cutting-edge XAI systems designed for ASP.

In Table 2, a summary of the compared features is presented. The features include as follows:

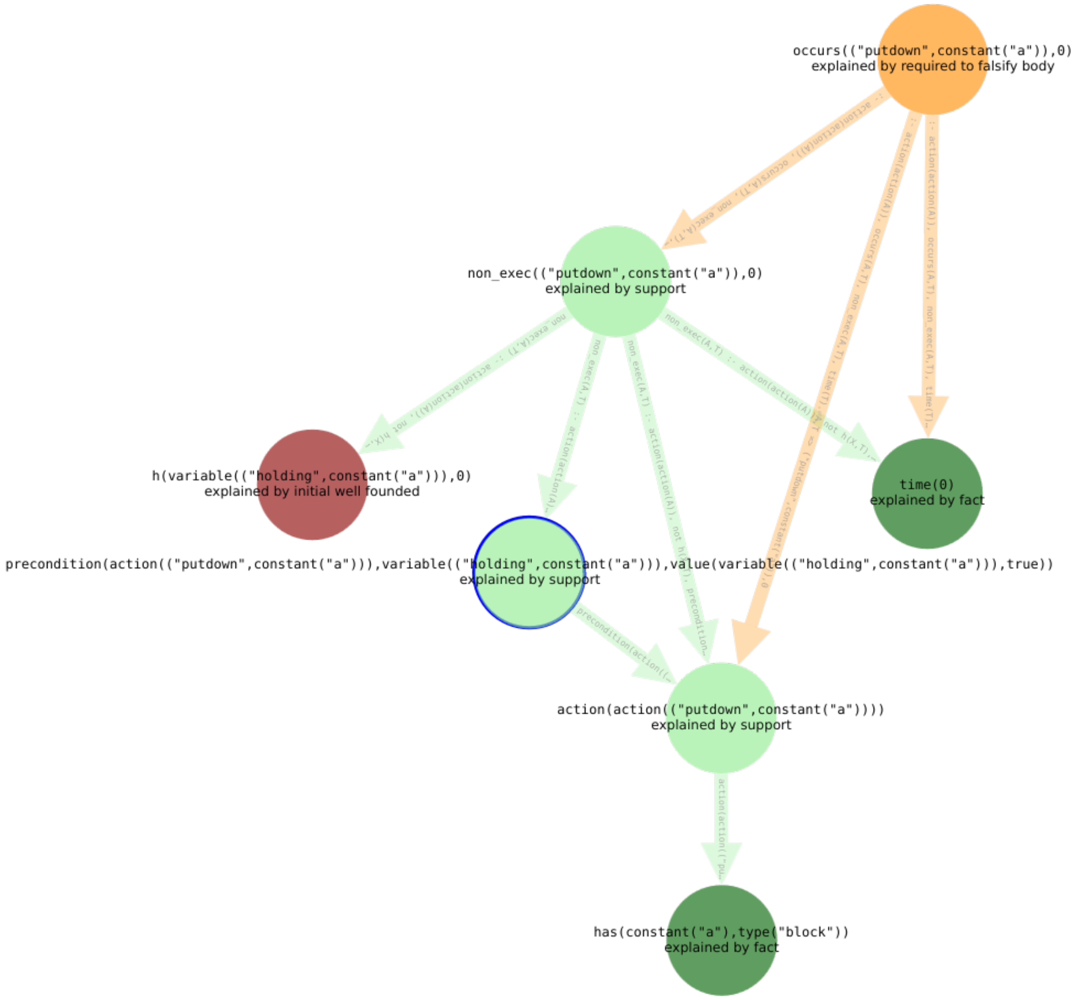


Figure 7: The DAG for atom $occurs((\text{"}putdown\text{"}, constant(\text{"}a\text{"})), 0)$.

whether the explanation is guaranteed to be acyclic; the capability to handle the input program with aggregates and constraints; the ability to provide an explanation when the query atom can be false in the answer set; and whether the system is available for experimentation. As can be seen from Table 2, our system is capable of providing explanations for false atoms and does not lead to cyclic argumentation in the explanation. `xASP2` is the only system that tackles a program that includes both aggregates and constraints. It is worth noting that the topic of aggregates is addressed in another approach [11], even though no system implementing this approach is mentioned or available.

8 Conclusion

We have developed and implemented `xASP2` , an XAI system that targets the ASP language and is powered by ASP engines. Our approach to explaining why an atom is true/false in an answer set involves deriving easy-to-understand inferences originating from a hopefully small set of atoms assumed false. `xASP2` has the ability to support different `clingo` constructs such as aggregates and constraints. It produces an explanation as a DAG with the atom to be explained as the root and takes a few seconds to compute the explanation in our test cases. Further investigation to include ASP's other linguistic

Table 2: Summary of compared features

System (if any) and reference	**Acyclic explanation**	**Linguistic extentions**	**Explanation for false atoms**	**System availability**
`s(CASP)` [1]	Yes	Constraints	Yes	Yes
`ASPeRiX` [2]	Yes	Constraints	Yes	Yes
`spock` [3]	Yes	Constraints	No	Yes
`xclingo` [4]	Yes	None	No	Yes
`DWASP` [6]	Yes	Constraints	No	Yes
[11]	No	Aggregates	Yes	No
`Visual-DLV` [15]	Yes	Constraints	No	Yes
[16]	No	None	Yes	No
`LABAS` [17]	No	None	Yes	Yes
exp(ASP^c) [18]	No	Constraints	Yes	Yes
[20]	Yes	None	Yes	No
`xASP2`	Yes	Aggregates and Constraints	Yes	Yes

constructs such as conditional literals, beyond those currently supported, present in the future work.

Acknowledgments

Portions of this publication and research effort are made possible through the help and support of NIST via cooperative agreement 70NANB21H167. Tran Cao Son was also partially supported by NSF awards #1757207 and #1914635. Mario Alviano was also partially supported by Italian Ministry of Research (MUR) under PNRR project FAIR "Future AI Research", CUP H23C22000860006, under PNRR project Tech4You "Technologies for climate change adaptation and quality of life improvement", CUP H23C22000370006, and under PNRR project SERICS "SEcurity and RIghts in the CyberSpace", CUP H73C22000880001; by Italian Ministry of Health (MSAL) under POS project RADIOAMICA, CUP H53C22000650006; by the LAIA lab (part of the SILA labs) and by GNCS-INdAM.

References

[1] Joaquín Arias, Manuel Carro, Zhuo Chen & Gopal Gupta (2020): *Justifications for Goal-Directed Constraint Answer Set Programming*. *Electronic Proceedings in Theoretical Computer Science* 325, p. 59–72, doi:10.4204/EPTCS.325.12.

[2] Christopher Béatrix, Claire Lefèvre, Laurent Garcia & Igor Stéphan (2016): *Justifications and blocking sets in a rule-based answer set computation*. In: *Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, pp. 6:1–6:15, doi:10.4230/OASIcs.ICLP.2016.6.

[3] Martin Brain, Martin Gebser, Jörg Pührer, Torsten Schaub, Hans Tompits & Stefan Woltran (2007): *That is illogical captain! The debugging support tool spock for answer-set programs: system description*. In: *Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07)*, pp. 71–85.

[4] Pedro Cabalar, Jorge Fandinno & Brais Muñiz (2020): *A System for Explainable Answer Set Programming*. *Electronic Proceedings in Theoretical Computer Science* 325, p. 124–136, doi:10.4204/EPTCS.325.19.

[5] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca & Torsten Schaub (2020): *ASP-Core-2 Input Language Format*. Theory Pract. Log. Program. 20(2), pp. 294–309, doi:10.1017/s1471068419000450.

[6] Carmine Dodaro, Philip Gasteiger, Kristian Reale, Francesco Ricca & Konstantin Schekotihin (2019): *Debugging non-ground ASP programs: Technique and graphical tools*. Theory and Practice of Logic Programming 19(2), pp. 290–316, doi:10.1017/S1471068418000492.

[7] M. Gelfond & V. Lifschitz (1990): *Logic programs with classical negation*. In D. Warren & Peter Szeredi, editors: Logic Programming: Proc. of the Seventh International Conference, pp. 579–597.

[8] Roland Kaminski, Javier Romero, Torsten Schaub & Philipp Wanko (2023): *How to Build Your Own ASP-based System?!* Theory and Practice of Logic Programming 23(1), p. 299–361, doi:10.1017/S1471068421000508.

[9] Fang Li, Huaduo Wang, Kinjal Basu, Elmer Salazar & Gopal Gupta (2021): *DiscASP: A Graph-based ASP System for Finding Relevant Consistent Concepts with Applications to Conversational Socialbots*. In: Proceedings 37th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2021, Porto (virtual event), 20-27th September 2021, EPTCS 345, pp. 205–218, doi:10.4204/EPTCS.345.35.

[10] V. Marek & M. Truszczyński (1999): *Stable models and an alternative logic programming paradigm*. In: The Logic Programming Paradigm: a 25-year Perspective, pp. 375–398, doi:10.1007/978-3-642-60085-2_17.

[11] Simon Marynissen, Jesse Heyninck, Bart Bogaerts & Marc Denecker (2022): *On nested justification systems (full version)*. CoRR abs/2205.04541, doi:10.48550/arXiv.2205.04541.

[12] Van Nguyen, Stylianos Loukas Vasileiou, Tran Cao Son & William Yeoh (2020): *Explainable planning using answer set programming*. In: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning, pp. 662–666, doi:10.24963/kr.2020/66.

[13] I. Niemelä (1999): *Logic programming with stable model semantics as a constraint programming paradigm*. Annals of Mathematics and Artificial Intelligence 25(3,4), pp. 241–273, doi:10.1023/A:1018930122475.

[14] Nikolay Pelov, Marc Denecker & Maurice Bruynooghe (2007): *Well-founded and stable semantics of logic programs with aggregates*. Theory Pract. Log. Program. 7(3), pp. 301–353, doi:10.1017/S1471068406002973.

[15] Simona Perri, Francesco Ricca, Giorgio Terracina, D Cianni & P Veltri (2007): *An integrated graphic tool for developing and testing DLV programs*. In: Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07), pp. 86–100.

[16] Enrico Pontelli, Tran Cao Son & Omar Elkhatib (2009): *Justifications for logic programs under answer set semantics*. Theory and Practice of Logic Programming 9(1), pp. 1–56, doi:10.1017/S1471068408003633.

[17] Claudia Schulz & Francesca Toni (2016): *Justifying answer sets using argumentation*. Theory and Practice of Logic Programming 16(1), pp. 59–110, doi:10.1017/S1471068414000702.

[18] Ly Ly Trieu, Tran Cao Son & Marcello Balduccini (2021): *exp(aspc): explaining asp programs with choice atoms and constraint rules*. Electronic Proceedings in Theoretical Computer Science, doi:10.4204/EPTCS.345.28.

[19] Ly Ly Trieu, Tran Cao Son & Marcello Balduccini (2022): *xASP: An Explanation Generation System for Answer Set Programming*. In: International Conference on Logic Programming and Nonmonotonic Reasoning, Springer, pp. 363–369, doi:10.1007/978-3-031-15707-3_28.

[20] Carlos Viegas Damásio, Anastasia Analyti & Grigoris Antoniou (2013): *Justifications for logic programming*. In: Logic Programming and Nonmonotonic Reasoning: 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proc. 12, Springer, pp. 530–542, doi:10.1007/978-3-642-40564-8_53.

A Dataflow Analysis for Comparing and Reordering Predicate Arguments

Gonzague Yernaux
Namur Digital Institute
Faculty of Computer Science
University of Namur, Belgium
`gonague.yernaux@unamur.be`

Wim Vanhoof
Namur Digital Institute
Faculty of Computer Science
University of Namur, Belgium
`wim.vanhoof@unamur.be`

In this work, which is done in the context of a (moded) logic programming language, we devise a data-flow analysis dedicated to computing what we call argument profiles. Such a profile essentially describes, for each argument of a predicate, its functionality, i.e. the operations in which the argument can be involved during an evaluation of the predicate, as well as how the argument contributes to the consumption and/or construction of data values. While the computed argument profiles can be useful for applications in the context of program understanding (as each profile essentially provides a way to better understand the role of the argument), they more importantly provide a way to discern between arguments in a manner that is more fine-grained than what can be done with other abstract characterizations such as types and modes. This is important for applications where one needs to identify correspondences between the arguments of two or more different predicates that need to be compared, such as during clone detection. Moreover, since a total order can be defined on the abstract domain of profiles, our analysis can be used for rearranging predicate arguments and order them according to their functionality, constituting as such an essential ingredient for predicate normalization techniques.

1 Introduction

When writing code, subroutines (be it methods, procedures, functions or predicates) and their arguments play an important role, as they constitute the main mechanism by which the programmer can make his or her code modular and general and thus applicable in different contexts. While this is true in any language, it is even more so in declarative languages where modularity is often more fine-grained, resulting in lots of small functions and predicates, and where the lack of iterative control structures makes induction-based control (which itself heavily relies on argument manipulation) the rule rather than the exception [11]. In this work we consider logic programming and thus predicates as the program's main building blocks.

Understanding the source code of a predicate requires thus understanding the role of the arguments involved, and the data flow relations expressed within the code. If one pursues debugging purposes for instance, statically inferring upon which potential instructions (or, in a logic programming context, atoms) each argument does or does not have influence is crucial to better understand the program at hand [14, 24]. While dataflow analysis is a well-known and indispensable ingredient in applications such as code comprehension [13], compiler optimization [3] and automatic parallelization [17], its potential has, to the best of our knowledge, been less explored in applications such as code normalization, anti-unification and clone detection [18, 19] which is the prime motivation for the current work.

Indeed, when comparing predicate definitions during clone detection or anti-unification, one wants to detect as many (dis)similarities as possible [27]. It is then often important to consider the right matching

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 41–54, doi:10.4204/EPTCS.385.5

between the respective arguments, as the following somewhat contrived example shows. Consider the traditional definition of the `append/3` predicate and another predicate, `concat/3`:

```
append([],L,L).
append([X|Xs],Y,[X|Zs]):- append(Xs,Y,Zs).

concat(L,[],L).
concat([E|Zs],[E|Es],Y):- concat(Zs,Es,Y).
```

Intuitively it is clear that the two predicates define essentially the same ternary relation, where one argument is the concatenation of the two others. The code of the two predicates differs not only in the names of the variables used, but also in the role played by the arguments. Indeed, for an atom `append`(t_1,t_2,t_3) to succeed, t_3 must be the concatenation of t_1 and t_2 whereas for `concat`(t_1,t_2,t_3) to succeed, it is t_1 that must be the concatenation of t_2 and t_3. For an analysis to detect that one of these predicates is a "clone" – a textual variant (renaming) of the other *modulo a permutation of the arguments*, it needs to consider potentially all possible argument permutations which adds a non-negligible factor to the complexity of the detection process. In fact, the search for a so-called argument mapping (designating the pairing of corresponding arguments in two predicates) that maximizes the outlined similarity of the involved definitions is one of the key factors rendering a search-based clone detection procedure or, more broadly, the computation of so-called *predicative anti-unification* intractable [28]. This is especially true when the predicates to be compared are composed of more than a few clauses, since for each suitable argument mapping, there might exist a large number of potential clause mappings that should be explored to find a functional link between the predicates to be compared.

It is not hard to see that the problem of finding a suitable argument mapping can be alleviated by taking adequate abstractions into account. Type- and mode information, for instance, can substantially reduce the number of argument mappings to consider, at least if a sufficient number of arguments are of different type and/or mode. In the example above type information does not really help (as all arguments are supposed to be of the same list type), but using mode information allows to limit the search for corresponding arguments to the subset of input, respectively output arguments of each predicate.

In a more general setting, the question is related to the problem of reordering the arguments in a standard (and preferably unique) way such that arguments playing a similar role (in different predicates) are positioned in similar positions. Ordering arguments is an important aspect of *code normalization*, a process that, generally speaking, aims at restructuring and simplifying code fragments or programs into some kind of *normal* or *canonical* form [5, 2] Again, while type and mode information can be used to classify arguments, it is generally not sufficient to sort all of the arguments in a unique way.

In this work, we introduce the notion of an argument profile being an abstract characterization of how that argument is used within the predicate and we devise an analysis capable of computing such profiles. Our approach encompasses, to some extent, type and mode information, but goes further by incorporating into the abstract domain the operations in which the argument participates. While the result of our analysis is not guaranteed to identify each and every argument by a unique value, examples show that it is capable of distinguishing between arguments much more precisely than approaches using only type and mode information.

2 Basic Concepts and Notations

In this paper we consider a simple logic language $\mathscr{L}$ where predicates, clauses, atoms and terms are used and defined in a style similar to that of Prolog. The language is however moded and represents, as such, certain similarities with (a subset of) Mercury [12]. We assume given a finite set of variables $\mathscr{V}$, a finite set of functor symbols $\mathscr{F}$ and a finite set of predicate symbols $\mathscr{P}$. As usual variables in $\mathscr{V}$ are strings starting with an uppercase letter while functors and predicates from $\mathscr{F}$, respectively $\mathscr{P}$ are written p/n where p is a string starting with a lowercase letter or symbol called the name of the functor (resp. predicate) and $n \in \mathbb{N}$ its arity, i.e. its number of arguments. We will ease notation by supposing that if a predicate (or functor) p/n exists in the program, then no predicate (or functor) p/m with $m \neq n$ can exist, so that a predicate (or functor) p/m will sometimes simply be referred to as p. The set of terms constructed from $\mathscr{V}$ and $\mathscr{F}$ is denoted $\mathscr{T}$. A term $t \in \mathscr{T}$ is said to be ground if it contains no variables.

A program is defined as a set of predicate definitions, where each predicate is defined by a set of clauses. For simplicity, we will consider only definite clauses, that is each clause is of the form $H \leftarrow B_1,\ldots,B_n$ where H is an atom denoted the head of the clause, and $B_1,\ldots,B_n$ a conjunction of atoms denoting its body. We furthermore assume that the head of a clause contains only variables as arguments (all unifications are made explicit in the body) and that all clauses defining a predicate share the same head. For a predicate p we will use $def(p)$ to denote the set of clauses in its definition and $args(p)$ to denote the sequence of its formal argument variables. With a slight abuse of notation we denote by $args(p)_i$ the ith formal argument of p (i being a number between 1 and the arity of p). For any given program construction c, be it a predicate, a clause, an atom or a clause head, we denote by $vars(c)$ the set of variables occurring in c. We will suppose that each atom in the program is uniquely identified by a natural number from $\mathbb{N}$ that will be referred to as the atom's program point in the program.

We will restrict ourselves to programs that are *directly recursive* to ease the analysis formulation and obtain concrete and efficient results [7]. Without loss of generality, we will also assume that clause bodies are in some standard, flattened, form in which each atom is either a predicate call having only variables as arguments, or a unification between variables and/or terms in which each term has only an outermost functor (its arguments being variables). We consider our language to be *moded*: each argument appearing in a clause's head is characterized as being either input or output. The argument modes restrict the usage of the predicate in the sense that any call to the predicate must provide a fully instantiated (ground) value for the input arguments, whereas each output argument will be a free variable that is guaranteed to be bound to a ground value upon success of the call. Likewise, unifications are moded as well.

Definition 1. *A moded unification is an atom in one of the following forms.*

- $V \Rightarrow f(X_1,\ldots,X_n)$*, called deconstruction, where* V *is supposed to be input and* $X_1,\ldots,X_n$ *output. It succeeds if the value bound to* V *has* f/n *as an outermost functor in which case it binds* $X_1,\ldots,X_n$ *to the values figuring in the arguments of* f/n*.*
- $V \Leftarrow f(X_1,\ldots,X_n)$*, called construction, where* V *is supposed to be output and* $X_1,\ldots,X_n$ *input. The construction succeeds if during evaluation* $f(X_1,\ldots,X_n)$ *is a ground value that can be bound to the free variable* V*.*
- $V \leftrightarrow W$*, called test, where both* V *and* W *are supposed to be input. The test succeeds if both* V *and* W *are bound to identical ground values.*
- $V := W$*, called assignment, where* V *is supposed to be output, and* W *input. The assignment succeeds if* W *is bound to a ground value that can be assigned to the free variable* V*.*

Given these constructions and the moded context, our predicates do to some extent resemble what are called *procedures* in Mercury [12].

Example 1. *If we represent lists in the usual way, by a functor nil representing the empty list and a functor cons/2 for list construction, the predicate* `app/3` *below, to be used in a mode* `(input, input, output)` *realizes the classical ground list concatenation operation in $\mathscr{L}$. The first two arguments are thus supposed to be input, the third one output. The subscript numbers represent the atoms' program points.*

$$\begin{array}{lll} app(X,Y,Z) & \leftarrow & X \Rightarrow_1 nil, Z :=_2 Y. \\ app(X,Y,Z) & \leftarrow & X \Rightarrow_3 cons(E,Es), app_4(Es,Y,Zs), Z \Leftarrow_5 cons(E,Zs). \end{array}$$

In the remainder of the paper, we will use $\mathscr{A}$ to represent the set of atoms (predicate calls and unifications) as they can occur in the program text, i.e. in the flat form defined above. For an atom $A \in \mathscr{A}$, we denote by $in(A)$ the input arguments of A and by $out(A)$ its output arguments. Note that this only concerns variables, i.e. for any $A \in \mathscr{A}$ we have $in(A) \subseteq vars(A)$ and $out(A) \subseteq vars(A)$. As usual, a substitution is a mapping from variables to terms and applying a substitution θ to a syntactical construct e, written $e\theta$, denotes the construct obtained by simultaneously replacing in e all variables from the domain of θ, denoted $dom(\theta)$, with their corresponding value. Given substitutions θ and σ, their composition $\theta \circ \sigma$ is also written as $\theta\sigma$. A *renaming* $\rho : \mathscr{V} \mapsto \mathscr{V}$ is a special kind of substitution as it is an injective (and idempotent) mapping between variables.

We suppose that programs, when executed, behave in a mode-correct way, meaning that if an instance of an atom (be it a unification or a predicate call) is selected for resolution, the arguments in the atom's input positions are bound to ground values, whereas the arguments in the output positions are unbound variables. To formalize the semantics of our language, we thus introduce the notion of a mode-correct instance.

Definition 2. *Let $A \in \mathscr{A}$ be an atom (predicate call or unification). We say that A' is a mode-correct instance of A if and only if there exists a substitution θ such that $A' = A\theta$ and*

(1) $\forall X \in in(A) : \theta(X)$ is a ground term;

(2) $\forall X \in out(A) : \theta(X)$ is a free variable if $X \in dom(\theta)$.

The semantics of the moded unifications defined above can easily be defined as follows:

Definition 3. *Let $U \in \mathscr{A}$ denote a unification and $U\theta$ (for some substitution θ) a mode-correct instance. Then we say that $U\theta$* succeeds with answer *θ' if and only if the following holds:*

- *If U is of the form $X \Rightarrow f(Y_1,\ldots,Y_n)$ it holds that $\theta(X) = f(t_1,\ldots,t_n)$ and $\theta' = \{Y_1/t_1,\ldots,Y_n/t_n\}$.*
- *If U is of the form $X \Leftarrow f(Y_1,\ldots,Y_n)$ it holds that $\theta' = \{X/f(\theta(Y_1),\ldots,\theta(Y_n))\}$.*
- *If U is of the form $X \leftrightarrow Y$ it holds that $\theta(X) = \theta(Y)$ and $\theta' = \emptyset$.*
- *If U is of the form $X := Y$ it holds that $\theta' = \{X/\theta(Y)\}$.*

The operational semantics of a program is defined in function of a query as usual.

Definition 4. *Given a program P, let Q be a query of the form $\leftarrow A_1,\ldots,A_n$. We say that a query Q'* is derived from Q with answer *θ if and only if one of the following conditions holds:*

1. *A_1 is a mode-correct instance of a unification that succeeds with answer θ, and Q' is the query $\leftarrow (A_2,\ldots,A_n)\theta$.*
2. *A_1 is a mode-correct instance $p(t_1,\ldots,t_n)$ of the head $H = p(X_1,\ldots,X_n)$ of a (renamed apart) clause $H \leftarrow B_1,\ldots,B_k \in P$ and Q' is the query $\leftarrow (B_1,\ldots,B_k,A_2,\ldots,A_n)\theta$ where it holds that $\theta = \{X_1/t_1,\ldots,X_n/t_n\}$.*

The above definition is basically equivalent to a traditional SLD-resolution step (with a leftmost selection rule) except for the explicit handling of the (moded) unifications and the limitation to resolving mode-correct instances of atoms only. Next, we can define the notion of a derivation as a sequence of individual derivation steps.

Definition 5. *Given a program P and query Q_0. A* derivation *for Q in P is a sequence of queries and substitutions $Q_0 \overset{\theta_0}{\to} Q_1 \overset{\theta_1}{\to} \ldots \overset{\theta_{n-1}}{\to} Q_n$ such that Q_i is derived from Q_{i-1} with answer θ_{i-1} for each $1 \leq i \leq n$. If Q_n is the empty query $\diamond$ then we say that the derivation is* successful *and has associated computed answer substitution $\theta_0\theta_1 \ldots \theta_{n-1}$.*

Again, our notion of a derivation is essentially equivalent to an SLD-derivation with a left-to-right selection rule. However, as a consequence of the simple mode system, all computed answers are ground substitutions.

3 Argument and Predicate Profiles

The analysis described in the next section essentially interprets a well-moded logic program and registers the encountered operations into special sets called *interaction sets* that will in the end allow to define a so-called *profile* for each of the predicate's arguments. The key idea of this section is to formalize the values that will be computed and manipulated by our analysis.

First, let us abstract n-ary computations by the dataflow relations that are exhibited between the arguments of a predicate, each dataflow relation being annotated by the set of operations that participate in the relation. Among the operations of interest are the basic unification operators defined by the set B as follows:

$$B = \{:=, \leftrightarrow\} \cup \bigcup_{f \in \mathscr{F}} \{\Leftarrow_f, \Rightarrow_f\}$$

For a given argument, we will represent a single dataflow relation it participates in by means of an *o-set*, the latter being essentially a tuple (o, j) in which o represents a subset of operations (from a given set of admissible operations, like B above) and j a natural number representing the position of one of the (other) arguments. More formally:

Definition 6. *Given a set of operations S, we define the set of* o-sets over *S as*

$$OS(S) = \{(o, j) \mid o \in \mathbb{P}(S) \text{ and } j \in \mathbb{N}\}$$

In general, an argument participates in more than one dataflow relation, relating it to several other arguments (each time by means of a set of operations). To represent such a *set* of dataflow relations, we introduce the notion of an *argument profile*. Intuitively, an argument profile for the i'th argument of p/n denotes a set of dataflow dependencies with some of the other arguments of p, where each dependency is represented – through an o-set – by the set of operations linking both arguments. Formally, we define the notion of an argument profile for an n-ary operation as follows:

Definition 7. *Given a set of operations S and $n \in \mathbb{N}$, we define an argument profile for an n-ary operation with respect to S as a set $A \subseteq OS(S)$ where for each $(o, j) \in A$ we have that $j \in \{1, \ldots, n\}$. We will use $AP_n(S)$ to represent the set of all possible argument profiles for an n-ary operation with respect to S.*

Example 2. *The following is an argument profile: $\{(\{\Rightarrow_{cons}, :=\}, 2), (\{\Leftarrow_{cons}\}, 3)\}$. It represents the fact that the concerned argument is involved through a deconstruction in a list, and an assignment, with the value of the argument in position 2. It also helps building the argument in position 3 through a list construction atom.*

The above definitions are fine as long as we restrict ourselves to using operations from a fixed set of operations such as B. However, it is worthwhile to include among the allowed operations also those operations defined (by means of predicates) in the program itself. We will not include the predicates as such in the set of admissible operations as it would make the domain too dependent on the names chosen for the predicates at hand. Rather, we will use abstractions of these predicates – notably those abstractions our analysis aims to compute. As such, the basic idea is to represent an n-ary operation (or predicate) by means of a term $\psi(\alpha_1,\ldots,\alpha_n)$ where the α are argument profiles. A special term $\psi_\perp$ is introduced in order to represent an operation for which no argument profiles are known; in the analysis it will be used to represent direct recursive calls. Since these ψ-based terms use argument profiles that themselves can contain ψ-based terms, we define the set of all possible abstract operations as the least fixed point of the following operator R:

Definition 8. *Given a set of operations S, we define*

$$R(S) = B \cup \{\psi_\perp\} \cup \bigcup_{n\in\mathbb{N}_0} \{\psi(\alpha_1,\ldots,\alpha_n) \mid \alpha_i \in AP_n(S)\}$$

While $lfp(R)$ contains some infinite terms, all terms created by our analysis will be of finite size, as will be made clear further down. In the following we use AP_n to refer to the set of all possible argument profiles for an n-ary operation with respect to $OS(lfp(R))$. We will refer to the elements of $lfp(R)$ in which a ψ appears as *ψ-based operations*.

In order to obtain argument profiles, the analysis will compute data flow relations within a predicate, annotated with the operations that are encountered upon establishing the relation. We thus define an *interaction* as being the association of an input variable and an output variable with a set of operations and the program points these operations are occurring at. Formally:

Definition 9. *Let p be a predicate in a program P. An* interaction in *p is a mapping* $vars(p) \times vars(p) \mapsto \mathbb{P}(lfp(R) \times \mathbb{N})$. *Notation-wise, we will typically write* $V \overset{O}{\rightsquigarrow} \hat{V}$ *to represent an interaction between a variable V and another variable* $\hat{V}$ *through a set* $O \subset lfp(R) \times \mathbb{N}$.

In order not to overload our notation, when writing interactions, we will usually drop the program points and consider the sets of operations in an interaction to be a multiset $O \subset lfp(R)$. We will thus allow doubles in the set, assuming they are operations implemented by atoms located at different program points. We will only occasionally include program points explicitly when needed in order to explicitly distinguish between identical operations coming from different atoms.

An important characteristic of the set of interactions describing a predicate is that for each pair of variables, there is at most a single interaction between these variables present in the set. Another characteristic is the fact that for any interaction $V \overset{O}{\rightsquigarrow} \hat{V}$ it holds that $\hat{V}$ cannot be an input argument, since mode-correct input arguments cannot be constructed by computations in a predicate's body. V does not have such a limitation, as long as V and $\hat{V}$ are distinct. More formally:

Definition 10. *For a predicate p, we call a well-defined interaction set for p a set* ϕ *of interactions in p such that for all* $V,\hat{V} \in vars(p)$ *it holds that if there exists* $V \overset{O}{\rightsquigarrow} \hat{V} \in \phi$ *for some O, then the following conditions all hold:*

1. $V \neq \hat{V}$;
2. $\nexists V \overset{O'}{\rightsquigarrow} \hat{V} \in \phi : O' \neq O$;
3. $\hat{V} \in args(p) \Rightarrow \hat{V}$ *is an output argument.*

We will use $ISet_p$ to denote the set of all well-defined interaction sets for a given predicate p. In case p is clear from the context, we will use the shorter notation $ISet$. Now we define the following quasi-order allowing to organize $ISet_p$ in a lattice.

Definition 11. *Let p be a predicate. For $\phi_1, \phi_2 \in ISet_p$ we say that ϕ_1 is more precise than ϕ_2, denoted $\phi_1 \sqsubseteq \phi_2$, if and only if $\forall V \overset{O}{\rightsquigarrow} \hat{V} \in \phi_1 : \exists V \overset{O'}{\rightsquigarrow} \hat{V} \in \phi_2$ such that $O \subseteq O'$.*

That is, $\phi_1 \sqsubseteq \phi_2$ when each interaction appearing in ϕ_1 labeled by an operation set O is matched by an interaction in ϕ_2 that is labeled by an operation set being a superset of O, and ϕ_2 may contain interactions involving pairs of variables that are not linked by an interaction in ϕ_1. We now define the following operator.

Definition 12. *For a predicate p, let $\phi \in ISet_p$ and let $V \overset{O}{\rightsquigarrow} \hat{V}$ be an interaction for p. Then we define*

$$(V \overset{O}{\rightsquigarrow} \hat{V}) \sqcup \phi = \begin{cases} \{V \overset{O}{\rightsquigarrow} \hat{V}\} \cup \phi & \text{if } \nexists (V \overset{O'}{\rightsquigarrow} \hat{V}) \in \phi \text{ for some } O' \\ (\phi \setminus \{V \overset{O'}{\rightsquigarrow} \hat{V}\}) \cup \{V \overset{O \cup O'}{\rightsquigarrow} \hat{V}\} & \text{otherwise} \end{cases}$$

Note that adding an interaction to a well-defined interaction set results in a well-defined interaction set. It can also be easily seen that when constructing a well-defined interaction set, the order in which the individual interactions are added has no influence on the final result. Consequently, we can extend the $\sqcup$ operator such that it merges two well-defined interaction sets:

Definition 13. *Let ϕ and ϕ' be well-defined interaction sets for a predicate p. Then we define $\phi \sqcup \phi'$ as the following well-defined interaction set: $\phi \sqcup \phi' = \bigsqcup_{V \overset{O}{\rightsquigarrow} \hat{V} \in \phi} (V \overset{O}{\rightsquigarrow} \hat{V}) \sqcup \phi'$.*

Proposition 1. *For a predicate p, $(ISet_p, \sqcup)$ is a join semi-lattice.*

Proof. We need to prove that for a predicate p, the $\sqcup : ISet_p \times ISet_p \mapsto ISet_p$ operation is idempotent, associative and commutative. This follows directly from the definition of $\sqcup$ (being essentially a union operation on sets of interactions and possibly on sets of operations) and the fact that the union operator on sets is itself idempotent, associative, and commutative. ☐

The induced partial order, namely $\sqsubseteq$, is such that $\phi \sqsubseteq \phi'$ if and only if $\phi \sqcup \phi' = \phi'$, so that we get a partially ordered set $(ISet_p, \sqsubseteq)$ in which each subset $\{\phi_1, \dots, \phi_n\}$ has a least upper bound, namely $\sqcup\{\phi_1, \dots, \phi_n\}$. The partially ordered set has a minimal element, namely the empty set $\{\}$ which we will refer to by $\bot$ as it is a unit for the join operator: $\forall \phi \in ISet_p : \bot \sqcup \phi = \phi \sqcup \bot = \phi$. The maximal element $\top_p \in ISet_p$ is the set containing all possible interactions between each argument and all the (other) output arguments.

The goal of our analysis is to compute, for each predicate p in a given program P, a well-defined interaction set for p. This element of $ISet_p$ will be such that it only reflects the interactions between variables $V, \hat{V}$ such that $V, \hat{V} \in args(p)$. Such an element is what we will call a *predicate profile*.

Definition 14. *Given a program P and a predicate p defined therein. A* predicate profile *for p is a well-defined interaction set ϕ of interactions in p such that for all $V \overset{O'}{\rightsquigarrow} \hat{V} \in \phi$ we have that V and $\hat{V}$ are formal arguments of p, that is $\{V, \hat{V}\} \subseteq args(p)$.*

We can "decompose" a predicate profile into individual argument profiles as follows:

Definition 15. *Given a program P, a predicate p in P, and a predicate profile ϕ for p, we define the argument profile of the i'th argument of p with respect to ϕ as the following set of o-sets:*

$$\alpha_i = \{(O, j) \mid V_i \overset{O}{\rightsquigarrow} V_j \in \phi\}$$

where $V_i = args(p)_i$ and $V_j = args(p)_j$. Moreover, we define the computed argument profile *of p with respect to ϕ as the sequence $\langle \alpha_1, \dots, \alpha_n \rangle$.*

Recall that, based on such computed argument profiles, our objective is to *reorder* the predicate arguments, preferably in a unique way. As a first observation, note that it is not hard to define *a* total order on AP as the following example illustrates.

Example 3. *For an argument profile* $\alpha \in AP$*, let us define the* features of α *as the vector* $(\#\alpha, o, m, s, r, c, d)$ *with* o *the total number of operations contained in* α*,* r *the number of* ψ*-based operations in it,* c*,* a*,* d *its number of constructions, assignments and deconstructions respectively. Denoting by* (0) *a vector filled with zeroes, we define the total order* $\leq$ *as the operator such that for any two argument profiles* α_1 *and* α_2 *with respective features* t_1 *and* t_2*, the following holds:*

$$\alpha_1 \leq \alpha_2 \Leftrightarrow t_1 - t_2 = (0) \vee \textit{the first non-zero dimension in } t_1 - t_2 \textit{ is positive}$$

While the order of Example 3 is somewhat arbitrary and not necessarily capable of producing a *unique* order, its definition is independent of the analyzed program. In the following section, we construct our analysis that takes a total order $\leq$ on AP as a parameter. Given such an order $\leq$, for a predicate p with some profile ϕ, we will use $opr(\phi)$ to represent a profile of p ordered by $\leq$ with respect to ϕ.

Definition 16. *Given a predicate* p/n*, a profile* ϕ *and a total order* $\leq$*. Let* $\langle \alpha_1, \ldots, \alpha_n \rangle$ *be the argument profile of* p *with respect to* ϕ*. Then we define the* ordered profile *of* p *with respect to* ϕ *as a permutation* $\langle \alpha'_1, \ldots, \alpha'_n \rangle$ *of* $\langle \alpha_1, \ldots, \alpha_n \rangle$ *such that* $\alpha_i \leq \alpha_{i+1}$ *for all* $1 \leq i < n$*.*

4 A Dataflow Analysis Computing Argument Profiles

The analysis will basically compute what we call an *environment* which is a mapping from predicates to well-defined interaction sets that represent the already computed interactions between the predicate's formal arguments. We will use the symbol $\Phi : \mathscr{P} \mapsto ISet$ to represent such an environment. The analysis is defined by induction on the syntactic structure of the program's predicates. We start by defining the analysis of an individual atom. It basically incorporates the operations of interest into interactions involving local variables as well as arguments. The analysis is parametrized by the current environment Φ and a total order $\leq$ capable of ordering a predicate profile ϕ into $opr(\phi)$.

Definition 17. *Let* P *be a program of interest. The atomic analysis function* $\mathbb{A} : \mathscr{A} \mapsto (\mathscr{P} \mapsto ISet) \mapsto ISet$ *is defined as the function that returns, given an atom* A *and an environment* Φ*, a set of interactions composed by those operations from* $lfp(R)$ *that are found occurring in* A*:*

$$\mathbb{A}[\![V \Rightarrow f(Y_1, \ldots, Y_n)]\!]\Phi = \bigsqcup_{i \in 1..n} \{V \overset{\{\Rightarrow_f\}}{\rightsquigarrow} Y_i\}$$

$$\mathbb{A}[\![V \Leftarrow f(Y_1, \ldots, Y_n)]\!]\Phi = \bigsqcup_{i \in 1..n} \{Y_i \overset{\{\Leftarrow_f\}}{\rightsquigarrow} V\}$$

$$\mathbb{A}[\![V := W]\!]\Phi = \{W \overset{\{:=\}}{\rightsquigarrow} V\}$$

$$\mathbb{A}[\![V \leftrightarrow W]\!]\Phi = \{\}$$

$$\mathbb{A}[\![q(Y_1, \ldots, Y_m)]\!]\Phi = \Phi(q)\rho \sqcup \phi_q$$

$$\textit{where } \rho = \{args(q)_1/Y_1, \ldots, args(q)_m/Y_m\}$$

$$\textit{and } \phi_q = \left\{Y_i \overset{o}{\rightsquigarrow} Y_j \mid Y_i \in in(q(Y_1, \ldots, Y_m)), Y_j \in out(q(Y_1, \ldots, Y_m))\right\}$$

$$\textit{in which } o = \begin{cases} \psi_\perp & \textit{if it is a directly recursive call} \\ \psi(opr(\Phi(p))) & \textit{otherwise} \end{cases}$$

In the definition, we apply a renaming ρ to a set of interactions $\Phi(q)$, which consists in replacing each variable V from $dom(\rho)$ occurring in $\Phi(q)$ by $\rho(V)$. Using $opr(\Phi(p))$ allows the ψ-based operations occurring in an argument profile to describe atoms based on similar operations by means of normalized values. For instance, as will be made clear later on, whether a predicate makes a call to $app/3$ or to a variant of it where some arguments are swapped, the resulting ψ-based operation will be the same.

Example 4. *The following are applications of our function* $\mathbb{A}$ *on atoms that appear in the predicate app from Example 1. We consider given an environment* Φ_0 *that maps app on* $\bot$*.*

$$\mathbb{A}[\![X \Rightarrow cons(E,Es)]\!]\Phi_0 = \{X \overset{\{\Rightarrow_{cons}\}}{\rightsquigarrow} E, X \overset{\{\Rightarrow_{cons}\}}{\rightsquigarrow} Es\}$$

$$\mathbb{A}[\![Z \Leftarrow cons(E,Zs)]\!]\Phi_0 = \{E \overset{\{\Leftarrow_{cons}\}}{\rightsquigarrow} Z, Zs \overset{\{\Leftarrow_{cons}\}}{\rightsquigarrow} Z\}$$

$$\mathbb{A}[\![app(Es,Y,Zs)]\!]\Phi_0 = \{Es \overset{\{\psi_\bot\}}{\rightsquigarrow} Zs, Y \overset{\{\psi_\bot\}}{\rightsquigarrow} Zs\}$$

Extending the analysis function to clauses is relatively straightforward as it suffices to analyze each of the body atoms, joining the results using $\sqcup$. However, we need to include a transitive closure operator that allows to *combine* the interactions resulting from the analysis of the individual atoms such that the resulting interactions represent – where possible – data flow between arguments rather than involving local variables.

Definition 18. *Let* $p \in \mathscr{P}$ *and* $\phi \in ISet_p$*. Let* $T : ISet \mapsto ISet$ *denote the following operator*

$$T(\phi) = \{X \overset{O \cup O'}{\rightsquigarrow} Z \,|\, X \overset{O}{\rightsquigarrow} Y, Y \overset{O'}{\rightsquigarrow} Z \in \phi \textit{ for some distinct } X,Y,Z \in \mathscr{V}\}$$

and let $cl_T(\phi)$ *denote the transitive closure of* T *on* ϕ*, that is the smallest relation on* ϕ *that contains* T *and is transitive. Then the projection of* ϕ *onto the arguments of* p *is denoted by* $\pi_p(\phi)$ *and defined as*

$$\pi_p(\phi) = \{X \overset{O}{\rightsquigarrow} Y \in cl_T(\phi) \,|\, X,Y \in args(p)\}.$$

For a given $\phi \in ISet$, the transitive closure $cl_T(\phi)$ can always be computed by merging into ϕ those interactions that can be seen as *transitive interactions*, i.e. interactions that concern three different variables X,Y,Z in the way described in the Definition above. The number of these transitive interactions is inevitably finite, being proportional to the number of combinations among a finite number of variables.

The analysis of a complete program consists in repeatedly analyzing each and every clause of the program with respect to the current environment, computing as such an updated environment that incorporates the results of the current analysis round.

Definition 19. *Let* P *be a program and* $p \in P$ *a predicate of interest. The predicate analysis function* $\mathbb{S} : \mathscr{P} \mapsto (\mathscr{P} \mapsto ISet) \mapsto ISet$ *is defined as the function that returns, given a predicate* p *and an environment* Φ*, a well-defined interaction set for* p*:*

$$\mathbb{S}[\![p]\!]\Phi = \bigsqcup_{h \leftarrow a_1,\ldots,a_n \in def(p)} \pi_p\Big(\bigsqcup_{i \in 1\ldots n} \mathbb{A}[\![a_i]\!]\Phi\Big)$$

Note the effect of the different join operations. First, the interaction sets resulting from the analysis of the individual atoms in a clause body are combined (using the innermost join). The outermost join combines the interaction sets resulting from the different clauses, after projection, into a single interaction set. The projection onto the arguments of the predicate is important, as it avoids the construction of spurious interactions caused by the same local variable that might be used in different clauses. The fact that local variables are ignored in the result of the formula above is no limitation, since the operator $\mathbb{S}$ is used below to compute the successive environments, and our analysis uses the environment solely for exploiting the interactions among arguments.

Algorithm 1 Analyzing a program P

```
$PS \leftarrow P, i \leftarrow 0, \Phi_0 \leftarrow \bigcup_{p \in P}\{\,(p,\bot)\,\}$
while $leafs(PS) \neq \emptyset$ do
    select $p \in leafs(PS)$
    while $(\mathbb{S}[\![p]\!]\Phi_i)(p) \neq \Phi_i(p)$ do
        $\Phi_{i+1} \leftarrow \mathbb{S}[\![p]\!]\Phi_i$
        $PS \leftarrow PS \setminus \{p\}$
        $i \leftarrow i+1$
```

Example 5. *Let us consider again the predicate app from Example 1. A round of our analysis for app is partially depicted in Example 4, its result being* $\mathbb{S}[\![app]\!]\Phi_0 = \{Y \overset{\{:=,\psi_\bot\}}{\rightsquigarrow} Z, X \overset{\{\Rightarrow_{cons},\psi_\bot\Leftarrow_{cons}\}}{\rightsquigarrow} Z\}$*, which corresponds to the projection on* X*,* Y *and* Z *of the following computed interactions:*

$$\{Y \overset{\{:=\}}{\rightsquigarrow} Z, X \overset{\{\Rightarrow_{cons}\}}{\rightsquigarrow} E, X \overset{\{\Rightarrow_{cons}\}}{\rightsquigarrow} Es, X \overset{\{\psi_\bot\}}{\rightsquigarrow} Z, Y \overset{\{\psi_\bot\}}{\rightsquigarrow} Z, E \overset{\{\Leftarrow_{cons}\}}{\rightsquigarrow} Z, Zs \overset{\{\Leftarrow_{cons}\}}{\rightsquigarrow} Z\}$$

Now, to analyze a program from scratch, we start from an initial environment Φ_0 in which each predicate is associated to an initial interaction set $\bot$. The predicates are subsequently analyzed according to their position in the program's call graph in a bottom-up manner, that is prioritizing those predicates that contain no calls to predicates except maybe themselves or predicates that have previously been analyzed. We will denote by $leafs(P)$ the set of such *eligible* predicates in a program P. Each time a predicate's analysis reaches a fixpoint, the analysis proceeds to the next eligible predicate. The process is repeated until every predicate has been considered. It is depicted in Algorithm 1.

Example 6. *Let us resume the analysis of* $app/3$ *started in Examples 4 and 5, where we obtained an environment value, say* Φ_1*, after one analysis round. A second round of the analysis will only differ in the handling of the atom* $app(Es,Y,Zs)$*:*

$$\mathbb{A}[\![app(Es,Y,Zs)]\!]\Phi_1 = \{Y \overset{\{:=,\psi_\bot\}}{\rightsquigarrow} Zs, Es \overset{\{\Rightarrow_{cons},\Leftarrow_{cons},\psi_\bot\}}{\rightsquigarrow} Z\}$$

After merging and projection on the arguments, we obtain Φ_2 *such that*

$$\Phi_2(app) = \{X \overset{\{\Rightarrow_{cons},\Leftarrow_{cons},\psi_\bot\}}{\rightsquigarrow} Z, Y \overset{\{\Leftarrow_{cons},:=,\psi_\bot\}}{\rightsquigarrow} Z\}$$

where the $\Leftarrow_{cons}$ *operation linking* Y *to* Z *is obtained by the fact that we have both* $Y \overset{\{:=\}}{\rightsquigarrow} Zs$ *and* $Zs \overset{\Leftarrow_{cons}}{\rightsquigarrow} Z$ *in the computed interactions set. Any subsequent analysis round would not alter this environment, so that the analysis is finished for* `app`*.*

Let us now consider that our program is also constituted of a moded version of the $concat/3$ *predicate introduced in Section 1:*

$$\begin{array}{lcl} concat(A,B,C) & \leftarrow & B \Rightarrow_6 nil, A :=_7 C. \\ concat(A,B,C) & \leftarrow & B \Rightarrow_8 cons(I,Is), concat_9(As,Is,C), A \Leftarrow_{10} cons(I,As). \end{array}$$

Analyzing concat yields the interactions $\{B \overset{\{\Rightarrow_{cons},\Leftarrow_{cons},\psi_\bot\}}{\rightsquigarrow} A, C \overset{\{\Leftarrow_{cons},:=,\psi_\bot\}}{\rightsquigarrow} A\}$*. Now using* $\leq$*, the ordered profiles of both predicates are one and the same, namely*

$$\langle\{(\{\Rightarrow_{cons},\Leftarrow_{cons},\psi_\bot\},2)\},\{(\{:=,\psi_\bot,\Leftarrow_{cons}\},2)\}\rangle$$

which corresponds to the respective profiles of X/B*,* Y/C *and* Z/A*. In other words, reordering the arguments according to* $\leq$ *leaves app untouched but transforms* $concat(A,B,C)$ *into* $concat(B,C,A)$*.*

The predicate calls in the example above being recursive calls, we introduce the following example to illustrate the case where a predicate makes calls to other predicates.

Example 7. *Let us extend Example 6 with the double append operation embodied by* $dapp/4$*:*

$$dapp(L1,L2,L3,L4) \leftarrow app_{11}(L1,L2,L12), concat_{12}(L4,L12,L3).$$

The analysis finds the following final interaction set for dapp:

$$\left\{ \begin{array}{l} L1 \overset{\{\Rightarrow_{cons}(3),\Leftarrow_{cons}(5),\psi_\perp(4),\psi_a(11),\Rightarrow_{cons}(8),\Leftarrow_{cons}(10),\psi_\perp(9),\psi_a(12)\}}{\rightsquigarrow} L4, \\ L2 \overset{\{:=(2),\psi_\perp(4),\Leftarrow_{cons}(5),\psi_a(11),\Rightarrow_{cons}(8),\Leftarrow_{cons}(10),\psi_\perp(9),\psi_a(12)\}}{\rightsquigarrow} L4, \\ L3 \overset{\{:=(7),\psi_\perp(9),\Leftarrow_{cons}(10),\psi_a(12)\}}{\rightsquigarrow} L4 \end{array} \right\}$$

where $\psi_a = \psi(\{(\{\Rightarrow_{cons}, \Leftarrow_{cons}, \psi_\perp\}, 2)\}, \{(\{:=, \psi_\perp, \Leftarrow_{cons}\}, 2)\})$ *and where the program points have been made explicit when applicable.*

The example shows that our analysis allows to entirely distinguish the four arguments of $dapp/4$, whereas type- and mode information alone would not have made a distinction among the first three arguments. Having these profiles for different arguments allows to order these by using an appropriate $\leq$ operator and, hence, to match $dapp/4$ with predicates that implement the same functionality differently.

A prototype implementation of the analysis, taking into account more elaborate examples, has been implemented. It is available online as an open source project[1]. The tool is capable of reordering predicate arguments and displaying the computed profiles for a directly recursive CLP program given as input.

We conclude this section with two important observations on the analysis described above. First, we show that the algorithm terminates. Next, we give an upper bound of its computational complexity.

Proposition 2. *The sequence* (Φ_n) *as defined by Algorithm 1 is convergent.*

Proof. First note that by construction, the sequence of computed environments $\Phi_0, \Phi_1, \ldots$ is such that $\forall i \in \mathbb{N}_0$, either $\Phi_i = \Phi_{i-1}$ and then Φ_i is the fixpoint of the sequence, or there exists $p \in \mathscr{P}$ such that $\Phi_i(p) \neq \Phi_{i-1}(p)$. In that case, the only possibilities are that

- $\Phi_i(p) \supset \Phi_{i-1}(p)$, due to a new interaction being discovered during the iteration, and/or
- $\exists V \overset{O_1}{\rightsquigarrow} \hat{V} \in \Phi_i(p), V \overset{O_2}{\rightsquigarrow} \hat{V} \in \Phi_{i-1}(p) : O_1 \neq O_2$. This can only happen if a new operation is added to an existing interaction, or if a ψ-based operation is replaced by a different ψ-based operation.

Now, for a predicate p/n, the number of interactions in $\Phi_i(p)$ (for any i) is limited by the number of pairs of (possibly interacting) arguments, which is of the order $O(n^2)$. Likewise, the set of operations labeling an interaction is necessarily finite, as its size is limited by the number of program points. What remains to be shown is that for an operation (a predicate call, say to some predicate q/m) at a given program point, there is no infinite succession of different ψ-based operations representing this operation. Now, this could only happen if the called predicate q/m was itself re-analyzed between analysis rounds of p. This is excluded, as we restricted programs to direct-recursive programs only, and our analysis analyses predicates bottom-up in the call-graph such that when a predicate is analyzed that is calling q/m, the analysis results for q/m are definitely known and hence the ψ-based operation representing this call will always be the same (some abstract profile $\psi(\alpha_1, \ldots, \alpha_m)$ or $\psi_\perp$ in case the call is recursive). □

Proposition 3. *Let P be a program containing* ℓ_P *predicates, with a total of* ℓ_a *program points. Let* $\ell_{io} = \max\{(j + (l-1)) \times l \mid p/n \in P, p/n \text{ has } j \text{ input arguments and } l \text{ output arguments}\}$*. Then the running time of the analysis is of worst-case complexity* $\mathscr{O}(\ell_P \times \ell_{io} \times \ell_a \times \ell_R)$ *with* ℓ_R *a finite natural proportional to the number of potential operations to be registered in the predicates.*

---

[1] The artifact code is available as a GitHub repository located at `https://github.com/Gounzy/PredArgs`.

Proof. Let us consider the analysis of a given predicate $p_k/n (k \in 1..\ell_P)$. The required lattice for the abstract value associated to the predicate has $\perp$, i.e. $\{\}$, as minimal set of interactions. The maximal element, $\top_p$, is the set containing an interaction $V_i \overset{\mathbb{O}_k}{\rightsquigarrow} V_o$ for each pair of variables $V_i, V_o \in args(p_k)$ such that $i \neq o$ and V_o is output. The elements in-between in the lattice are the sets of "incomplete" interactions, i.e. where all variables and/or operations are not present.

The number of combination of arguments in potential interactions of p_k is $(j + (l-1)) \times l$, with j, resp. l, the number of input, resp. output arguments of p_k, since each input argument can have exactly one interaction with each output argument, and each output argument can also contribute to the construction of the $(l-1)$ other output arguments. This quantity is majored by $n - 1 \times n$.

We still need to prove that a finite number of (also finite) operations from $lfp(R)$ suffices to populate the potential interactions and thereby restrict the lattice's height. First, observe that the number of operations in an interaction is majored by the number of program point in P which is finite. Now concerning the ψ-based operations, only a finite amount of these is treated by the analysis as stated earlier. We will denote by ℓ_R the number of operations that the analysis could possibly compute for a predicate given a program's call graph. For p_k, this quantity is proportional to both the number of program points in its body and, recursively, the number of potential operations of the predicates it makes calls to. These ψ-based operations evolve as they are recomputed by successive analysis rounds; ℓ_R represents the number of such steps that can occur before a computed ψ-operation converges.

So the height of the lattice, that is the maximal number of steps from $\perp$ to $\top_p$, is majored by $\ell_R \times (n-1) \times n$ (this corresponds to adding, at each step up the lattice, an operation to one of the existing interactions, or creating an interaction decorated by one operation). As the analysis climbs up in the lattice until reaching a fixpoint, this gives a realistic upper bound for the number of analysis iterations for p_k. The analysis might have to run up the lattice of each of the ℓ_P predicates in P, and at each iteration it needs to crawl through ℓ_a program points and compute ℓ_P projections, hence the result. ☐

5 Conclusions and Future Work

This work aims to develop a tractable process for profiling predicate arguments and normalizing their order of apparition in a prototypical Mercury-like language. Our analysis essentially computes a high-level abstraction of program derivations, called *interactions*. Although a normalization procedure already existed for Mercury [8], it focused on normalizing clause bodies and did not address predicate arguments.

Our approach to code normalization revolves around the search of an ordering among predicate arguments. Central to this technique is the research for an *ideal* ordering of the arguments, i.e. a total order $\leq$ that allows to sort arguments in a non-ambiguous, unique way, at least in the context of a single program. While we have introduced a first working, but rather arbitrary, example of such an order based on argument profiles metrics, it is our belief that more precise or application-tailored orderings could be found to enhance the analysis output in concrete situations. In particular, identifying the situations in which an order is to be preferred over other incarnations, is left as future work.

Having a normal form for programs is recognized as an important step in several applications, one of interest being a clone detection scheme, where recognizing a couple of similar predicates implies finding a mapping of clauses and a mapping of arguments among the predicates such that two clauses, or arguments, in the mapping play similar roles in the predicate's definition. The problem, which is intractable in general, becomes radically more manageable if a quadratic approximation is found for one of the two interleaved matching problems [28]. We intend to explore the use of our analysis for computing a matching of arguments in this context.

Program comprehension is a rising research field in which all aspects of dataflow information constitute useful pieces of information. Program slicing, for example, is a way of extracting the computations in which a given (set of) argument(s) plays a prominent role [24]. Interestingly, what we achieve by computing argument profiles resembles the extraction of such program slices. In existing program slicing techniques however, the computed slices are actual parts of the considered program [24], whereas our profiles rather constitute abstract representations of data flow information. Moreover, while an argument profile typically exhibits the details of the operations (be it unifications or calls to predicates) that involve the argument, the program portions obtained by means of slicing do not carry any *interpretation* of the program, as the slices' purpose is to represent the part of the program that might be of interest [20]. As an example, consider a predicate in which all of the arguments are somehow participating in every single atom but in different manners. The slices for the different arguments then systematically come down to the whole predicate definition. In contrast, our argument profiles contain finer-grained distinctions, allowing to identify which operations involve which arguments, as well as specific links between input and output arguments – but abstracting from the order in which the involved atoms are executed. We therefore believe our approach to be complementary to program slicing and to constitute a new step towards better understanding links between arguments and, hence, deriving useful information about the operations hidden in a predicate definition.

Other analyses addressing program comprehension or security concerns by studying interactions among variables could benefit from our method, some examples being feature analysis, trace analysis and taint analysis [9, 4].

References

[1] Magnus Ågren, Tamás Szeredi, Nicolas Beldiceanu & Mats Carlsson (2002): *Tracing and Explaining Execution of CLP(FD) Programs*. In Alexandre Tessier, editor: *Proceedings of the 12th International Workshop on Logic Programming Environments*, pp. 1–16, doi:10.48550/arXiv.cs/0207047.

[2] Danilo Bruschi, Lorenzo Martignoni & Mattia Monga (2007): *Code Normalization for Self-Mutating Malware*. *IEEE Security & Privacy* 5(2), pp. 46–54, doi:10.1109/MSP.2007.31.

[3] Keith D. Cooper, Timothy J. Harvey & Ken Kennedy (2006): *An Empirical Study of Iterative Data-Flow Analysis*. In: *2006 15th International Conference on Computing*, pp. 266–276, doi:10.1109/CIC.2006.22.

[4] Bas Cornelissen, Andy Zaidman, Arie Deursen, Leon Moonen & Rainer Koschke (2009): *A Systematic Survey of Program Comprehension through Dynamic Analysis*. *Software Engineering, IEEE Transactions on* 35, pp. 684 – 702, doi:10.1109/TSE.2009.28.

[5] Stefania Costantini & Alessandro Provetti (2005): *Normal forms for Answer Sets Programming*. *Theory and Practice of Logic Programming* 5, doi:10.1017/S1471068404002339.

[6] Céline Dandois & Wim Vanhoof (2012): *Semantic Code Clones in Logic Programs*. In E. Albert, editor: *Proc. of the 22nd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'12)*, *LNCS* 7844, Springer, pp. 35–50, doi:10.1007/978-3-642-38197-3_4.

[7] Saumya K. Debray (1992): *Efficient Dataflow Analysis of Logic Programs*. *J. ACM* 39(4), p. 949–984, doi:10.1145/146585.146624.

[8] François Degrave & Wim Vanhoof (2008): *Towards a Normal Form for Mercury Programs*. In Andy King, editor: *Logic-Based Program Synthesis and Transformation*, Springer, pp. 43–58, doi:10.1007/978-3-540-78769-3_4.

[9] Thomas Eisenbarth, Rainer Koschke & Daniel Simon (2001): *Aiding Program Comprehension by Static and Dynamic Feature Analysis*. doi:10.1109/ICSM.2001.972777.

[10] Rebecca Faust, Katherine Isaacs, William Z. Bernstein, Michael Sharp & Carlos Scheidegger (2019): *Anteater: Interactive Visualization for Program Understanding*. *CoRR*, doi:10.48550/arXiv.1907.02872.

[11] Melvin Fitting (2002): *Fixpoint semantics for logic programming a survey*. *Theoretical Computer Science* 278(1), pp. 25 – 51, doi:10.1016/S0304-3975(00)00330-3. Mathematical Foundations of Programming Semantics 1996.

[12] Fergus Henderson, Thomas Conway, Zoltan Somogyi, Peter Schachte, Simon Taylor & Chris Speirs (1999): *The Mercury Language Reference Manual*.

[13] Ulf Kargén & Nahid Shahmehri (2012): *InputTracer: A Data-Flow Analysis Tool for Manual Program Comprehension of x86 Binaries*. In: *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pp. 138–143, doi:10.1109/SCAM.2012.16.

[14] Ludovic Langevine, Pierre Deransart, Mireille Ducassé & Erwan Jahier (2001): *Prototyping CLP(FD) tracers: a trace model and an experimental validation environment*. In Anthony J. Kusalik, editor: *Proceedings of the Eleventh Workshop on Logic Programming Environments*, doi:10.48550/arXiv.cs/0111043.

[15] Florian D. Loch, Martin Johns, Martin Hecker, Martin Mohr & Gregor Snelting (2020): *Hybrid Taint Analysis for Java EE*. In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, Association for Computing Machinery, New York, NY, USA, p. 1716–1725, doi:10.1145/3341105.3373887.

[16] Anne Mulkers, William Winsborough & Maurice Bruynooghe (1994): *Live-Structure Dataflow Analysis for Prolog*. *ACM Trans. Program. Lang. Syst.* 16(2), p. 205–258, doi:10.1145/174662.174664.

[17] Kalyan Muthukumar, Francisco Bueno, Maria Jose García de la Banda & Manuel Hermenegildo (1999): *Automatic compile-time parallelization of logic programs for restricted, goal level, independent and parallelism*. *The Journal of Logic Programming* 38(2), pp. 165–218, doi:10.1016/S0743-1066(98)10022-5.

[18] Davide Pizzolotto & Katsuro Inoue (2020): *Blanker: A Refactor-Oriented Cloned Source Code Normalizer*. In: *14th International Workshop on Software Clones*, pp. 22–25, doi:10.1109/IWSC50091.2020.9047634.

[19] Dhavleesh Rattan, Rajesh Bhatia & Maninder Singh (2013): *Software clone detection: A systematic review*. *Information and Software Technology* 55(7), pp. 1165–1199, doi:10.1016/j.infsof.2013.01.008.

[20] Gyöngyi Szilágyi, Tibor Gyimóthy & Jan Maluszynski (2002): *Static and Dynamic Slicing of Constraint Logic Programs*. *Automated Software Engineering* 9, pp. 41–65, doi:10.1023/A:1013280119003.

[21] Jichang Tan & I-Peng Lin (1992): *Compiling Dataflow Analysis of Logic Programs*. *SIGPLAN Not.* 27(7), p. 106–115, doi:10.1145/143103.143123.

[22] Wim Vanhoof (2000): *Binding-Time Analysis by Constraint Solving*. In Michel Parigot & Andrei Voronkov, editors: *Logic for Programming and Automated Reasoning*, Springer, Berlin, Heidelberg, pp. 399–416, doi:10.1007/3-540-44404-1_25.

[23] Wim Vanhoof & Gonzague Yernaux (2020): *Generalization-Driven Semantic Clone Detection in CLP*. In Maurizio Gabbrielli, editor: *Logic-Based Program Synthesis and Transformation*, Springer International Publishing, Cham, pp. 228–242, doi:10.1007/978-3-030-45260-5_14.

[24] Martin Ward & Hussein Zedan (2007): *Slicing as a program transformation*. *ACM Trans. Program. Lang. Syst.* 29, doi:10.1145/1216374.1216375.

[25] William Hale Winsborough & Charles N. Fischer (1988): *Automatic, Transparent Parallelization of Logic Programs at Compile Time*. Ph.D. thesis, The University of Wisconsin - Madison.

[26] Mengmei Ye, Jonathan Sherman, Witawas Srisa-an & Sheng Wei (2018): *TZSlicer: Security-aware dynamic program slicing for hardware isolation*. In: *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 17–24, doi:10.1109/HST.2018.8383886.

[27] Gonzague Yernaux & Wim Vanhoof (2019): *Anti-unification in Constraint Logic Programming*. *Theory and Practice of Logic Programming* 19(5-6), p. 773–789, doi:10.1017/S1471068419000188.

[28] Gonzague Yernaux & Wim Vanhoof (2022): *On Detecting Semantic Clones in Constraint Logic Programs*. In: *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, pp. 32–38, doi:10.1109/IWSC55060.2022.00014.

Towards a Rule-based Approach for Deriving Abstract Domains (Extended Abstract) [*] [†]

[1,2]Daniel Jurjo, [1,2]Jose F. Morales, [3,2]Pedro López-García, and [1,2]Manuel V. Hermenegildo

[1]Universidad Politécnica de Madrid (UPM) Madrid, Spain

[2]IMDEA Software Institute, Madrid, Spain

[3]Spanish Council for Scientific Research

{daniel.jurjo,josef.morales,pedro.lopez,manuel.hermenegildo}@imdea.org

Abstract interpretation [3] allows constructing sound program analysis tools which can extract properties of a program by safely approximating its semantics. Static analysis tools are a crucial component of the development environments for many programming languages. Abstract interpretation proved practical and effective in the context of (Constraint) Logic Programming ((C)LP) [15, 12, 14, 13, 1, 10, 9] which was one of its first application areas (see [6]), and the techniques developed in this context have also been applied to the analysis and verification of other programming paradigms by using semantic translation to Horn Clauses (see the recent survey [4]). Unfortunately, the implementation of (sound, precise, efficient) abstract domains usually requires coding from scratch a large number of domain-related operations. Moreover, due to undecidability, a loss of precision is inevitable, which makes the design (and implementation) of more domains, as well as their combinations, eventually necessary to successfully prove arbitrary program properties. In this paper we focus on the latter problem by proposing a rule-based methodology for the design and rapid prototyping of new domains for logic programs, as well as composing and combining existing ones. Our techniques are inspired by those used in logic-based languages for implementing constraint domains at different abstraction levels.

Proposal. The construction of analyses based on abstract interpretation requires the defintion of some basic domain operations ($\sqsubseteq, \sqcap, \sqcup$ and, optionally, the widening ∇ operator); the abstract semantics of the primitive constraints (representing the *built-ins*, or basic operations of the source language) via *transfer functions* (f^{α}); and possibly some other additional instrumental operations over abstract substitutions. In addition, the classical top-down analysis approach requires a number of additional definitions of derived operations used by the analysis framework to implement procedure call, return, recursion, etc. Detailed descriptions of all these operations can be found in [12, 11, 2, 7, 5]. We propose a rule language inspired in rewriting to derive, script, and combine abstract domains. The objective is to reduce the time and effort required to write new abstract domains, both from scratch and as combinations of other domains

The proposed rule-based language. Given $s+1$ sets of constraints, $\mathscr{L}, \mathscr{C}_1, \ldots, \mathscr{C}_s$, we define $AND(\mathscr{L}, \mathscr{C}_1, \ldots, \mathscr{C}_s)$ as the set of rules of the form $l_1, \ldots, l_n \mid g_1, \ldots, g_l \Rightarrow r_1, \ldots, r_m \# label$, where s, n, m, and l are arbitrary positive integers, and the rule meets the following condition:

$$\forall i, j, k \text{ s.t. } i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\} \text{ and } k \in \{1, \ldots, l\} : (l_i, r_j \in \mathscr{L} \text{ and } \exists u \in \{1, \ldots, s\} \text{ s.t. } g_k \in \mathscr{C}_u)$$

The elements $l_1, \ldots, l_n$ constitute the *left side* of the rule; $r_1, \ldots, r_m$ the *right side*; and $g_1, \ldots, g_l$ the *guards*. Given $t+s$ sets of constraints $\mathscr{L}_1, \ldots, \mathscr{L}_t, \mathscr{C}_1, \ldots, \mathscr{C}_s$ such that $\forall v \in \{1, \ldots, t\} : \mathscr{L}_v \subseteq \mathscr{L}$, we

---

[*] An extended version of this work can be found in [8].

[†] This work has been partially supported by MICINN projects PID2019-108528RB-C21 *ProCode*, TED2021-132464B-I00 *PRODIGY*, and FJC2021-047102-I, the European Union NextGenerationEU/PRTR, and the Tezos foundation.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 55–57, doi:10.4204/EPTCS.385.6

```
inf(X, top)  | X = []            ==> inf(X, 0.Inf).  # empty
inf(L, X) | L = [H|T]            ==> inf(T, X).      # list_const1
inf(T, X) | L = [H|T], H =< X  ==> inf(L, H).      # list_const2
inf(L, X) | L = S                ==> inf(S, X).      # unif_prop
inf(L, X) | Y =< X               ==> inf(L, Y).      # reduction
inf(X, A) ; inf(X, B) | A =< B ==> inf(X, A).      # lub_1
inf(X, A) ; inf(X, B) | A >= B ==> inf(X, B).      # lub_2
```

Figure 2: A subset of the inf-domain rules.

define $OR(\mathscr{L}, \mathscr{C}_1, \ldots, \mathscr{C}_n)$ as the set of rules of the form $l_1; \ldots; l_n \mid g_1, \ldots, g_l \Rightarrow r_1, \ldots, r_m$ # *label*, where s, t, n, m, and l are arbitrary positive integers, and the rule meets the following condition:

$$\forall i, j, k \quad \text{s.t. } i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\} \text{ and } k \in \{1, \ldots, l\} :$$
$$\exists v \in \{1, \ldots, t\} \, \exists u \in \{1, \ldots, s\} \text{ s.t. } (l_i \in \mathscr{L}_v, r_j \in \mathscr{L} \text{ and } g_k \in \mathscr{C}_u)$$

Notice that while in *AND*−rules all the elements l_i belong to the same set of constraints $\mathscr{L}$, in the *OR*-rules they belong to (possibly) different subsets of a set of constraints $\mathscr{L}$. The operational meaning of *AND*−rules is similar to that of rewriting rules. If the left side holds in the set where the rule is being applied to, and the guards also hold, then the left-side elements are replaced by the right-side elements. The operational meaning of *OR*−rules is similar, but instead of rewriting over the "same" set the right-side elements are written in a "new" set. When no more rules can be applied, different strategies can be followed. In general we can rewrite the pending elements to a given element or simply delete them.

In the context of abstract interpretation the sets of constraints that we have mentioned have to be seen as abstract domains being the rules applied then over abstract substitutions/constraints. *AND-rules* are intended to capture the behaviour of operations over one abstract substitution with the knowledge that can be inferred from other substitutions that meet the guards. This is useful for example when defining the *greatest lower bound*. Moreover, these rules are also useful for refining abstract substitutions, performing abstractions, combining different abstract domains, etc. On the other hand, *OR-rules* are intended to capture the behaviour of operations applied over multiple abstract substitutions of an abstract domain, such as the *least upper bound* or the *widening*.

```
partition([], _, [], []).
partition([E|R], C, Left,
    [E|Right1]) :-
    E >= C,
    partition(R, C, Left, Right1).
partition([E|R], C, [E|Left1],
    Right) :-
    E < C,
    partition(R, C, Left1, Right).
```

Figure 1: A Prolog program.

An example. Fig. 1 shows a classic Prolog predicate for partitioning a list. A call `partition(L, X, L1, L2)` is expected to satisfy some properties; for example, that $\forall v \in L2, X \leq v$, which we can express as `inf(L2, X)`. With the help of two auxiliary domains to deal with structures containing variables and with constraints (resp. *depth* − *k* and *polyhedra*) we can derive an abstract domain for the `inf/2` property. A subset of the rules can be seen in Fig. 2. These rules allow, when connected with the abstract domain operations, to exploit the information gathered from the previous domains and use it to infer `inf(L2, X)`. Similarly, we can also capture the equivalent `sup(L1, X)`, or *multiset* properties capturing that $L \subseteq L1 \cup L2$ and $L1 \cup L2 \subseteq L$. Moreover, we can infer the *sortedness* property for the classical *quicksort* implementation.

Conclusions & future work. We have presented a framework for simplifying the development of abstract domains for logic programs in the context of abstract interpretation frameworks, and concretely that of `CiaoPP`. While some domains are easier to specify with a rule-based language, keeping a constraint-based representation for abstract substitutions may not be efficient compared with specialized representations and operations. In this respect, we plan to explore the use of rules both as an input language for *abstract domain compilation* and as a specification language for debugging or verifying properties of hand-written domains. In our experience so far, the proposed approach seems promising for prototyping and experimenting with new domains, enhancing the precision for particular programs, and adding domain combination rules, without the need for understanding the analysis framework internals.

References

[1] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens & W. Simoens (1996): *Global Analysis of Constraint Logic Programs*. ACM Transactions on Programming Languages and Systems 18(5), pp. 564–615.

[2] M. Bruynooghe (1991): *A Practical Framework for the Abstract Interpretation of Logic Programs*. Journal of Logic Programming 10, pp. 91–124.

[3] P. Cousot & R. Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In: ACM Symposium on Principles of Programming Languages (POPL'77), ACM Press, pp. 238–252, doi:10.1145/512950.512973.

[4] Emanuele De Angelis, Fabio Fioravanti, John P. Gallagher, Manuel V. Hermenegildo, Alberto Pettorossi & Maurizio Proietti (2021): *Analysis and Transformation of Constrained Horn Clauses for Program Verification*. Theory and Practice of Logic Programming FirstView, pp. 1–69, doi:10.1017/S1471068421000211. Available at `http://arxiv.org/abs/2108.00739`.

[5] I. Garcia-Contreras, J. F. Morales & M. V. Hermenegildo (2021): *Incremental and Modular Context-sensitive Analysis*. Theory and Practice of Logic Programming 21(2), pp. 196–243, doi:10.1017/S1471068420000496. Available at `https://arxiv.org/abs/1804.01839`.

[6] R. Giacobazzi & F. Ranzato (2022): *History of Abstract Interpretation*. IEEE Ann. Hist. Comput. 44(2), pp. 33–43. Available at `https://doi.org/10.1109/MAHC.2021.3133136`.

[7] M. V. Hermenegildo, G. Puebla, K. Marriott & P. Stuckey (2000): *Incremental Analysis of Constraint Logic Programs*. ACM Transactions on Programming Languages and Systems 22(2), pp. 187–223.

[8] D. Jurjo, J. F. Morales, P. Lopez-Garcia & M.V. Hermenegildo (2022): *A Rule-based Approach for Designing and Composing Abstract Domains*. Technical Report, CLIP Lab, IMDEA Software Institute.

[9] A. Kelly, K. Marriott, H. Søndergaard & P.J. Stuckey (1998): *A Practical Object-Oriented Analysis Engine for CLP*. Software: Practice and Experience 28(2), pp. 188–224.

[10] B. Le Charlier & P. Van Hentenryck (1994): *Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog*. ACM Transactions on Programming Languages and Systems 16(1), pp. 35–101.

[11] K. Muthukumar & M. Hermenegildo (1989): *Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation*. In: 1989 North American Conference on Logic Programming, MIT Press, pp. 166–189.

[12] K. Muthukumar & M. Hermenegildo (1990): *Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs*. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759. Available at `http://cliplab.org/papers/mcctr-fixpt.pdf`.

[13] K. Muthukumar & M. Hermenegildo (1992): *Compile-time Derivation of Variable Dependency Using Abstract Interpretation*. Journal of Logic Programming 13(2/3), pp. 315–347.

[14] P. Van Roy & A. M. Despain (1990): *The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler*. In: North American Conference on Logic Programming, MIT Press, pp. 501–515.

[15] R. Warren, M. Hermenegildo & S. K. Debray (1988): *On the Practicality of Global Flow Analysis of Logic Programs*. In: Fifth International Conference and Symposium on Logic Programming, MIT Press, pp. 684–699.

“Would life be more interesting if I were in AI?” Answering Counterfactuals based on Probabilistic Inductive Logic Programming

Kilian Rückschloß
Ludwig-Maximilians-Universität München
Oettingenstraße 67, 80538 München, Germany
`kilian.rueckschloss@lmu.de`

Felix Weitkämper
Ludwig-Maximilians-Universität München
Oettingenstraße 67, 80538 München, Germany
`felix.weitkaemper@lmu.de`

Probabilistic logic programs are logic programs where some facts hold with a specified probability. Here, we investigate these programs with a causal framework that allows counterfactual queries. Learning the program structure from observational data is usually done through heuristic search relying on statistical tests. However, these statistical tests lack information about the causal mechanism generating the data, which makes it unfeasible to use the resulting programs for counterfactual reasoning. To address this, we propose a language fragment that allows reconstructing a program from its induced distribution. This further enables us to learn programs supporting counterfactual queries.

1 Introduction

While only observing the world, humans are used to drawing counterfactual conclusions, i.e. they reason about how events would have unfolded under different circumstances. This leads us to judgements like: “I would have published more papers, if I were in AI.” without actually experiencing the alternative reality in which we work in AI. Note that this capability allows us to make sense of the past, to plan courses of actions, to make emotional and social judgments as well as to adapt our behaviour [2]. Hence, in artificial intelligence one also wants to infer a model of the world that supports counterfactual reasoning.

Currently, the WHATIF-solver [3] establishes a counterfactual reasoning for ProbLog programs [1], i.e. logic programs in which each clause holds with a specified probability. However, is the counterfactual reasoning provided by a ProbLog program uniquely determined by its distribution semantics [7]?

Assume for instance that a patient is treated, denoted *treatment*, with a probability of 0.5. If we treat a patient, we expect him to recover, denoted *recovery*, with a probability of 0.7, otherwise he recovers with a probability of 0.5. The resulting distribution can be encoded with the following two programs $\mathbf{P}_{1/2}$.

$$\mathbf{P}_1: \quad 0.5 :: treatment \quad 0.5 :: recovery \quad 0.4 :: recovery \leftarrow treatment$$

$$\mathbf{P}_2: \quad 0.5 :: treatment \quad 0.5 :: recovery \leftarrow \neg treatment \quad 0.7 :: recovery \leftarrow treatment$$

Assume further that the patient recovers while he has not been treated. What is the probability that he would have recovered under treatment?

In both programs $\mathbf{P}_{1/2}$, we conclude from our observations that the patient recovers because of the second clause, i.e. we conclude that the second clause holds in the world we observe. If we had additionally treated the patient, under program $\mathbf{P}_1$, he would still have recovered as the second clause in $\mathbf{P}_1$ is still applicable under treatment. Hence, we obtain a probability of one for the patient to recover under treatment. Whereas, in program $\mathbf{P}_2$, the second clause is not applicable under treatment. Hence, in this

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D’Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 58–64, doi:10.4204/EPTCS.385.7

case, if we had treated the patient here, he can only recover because of the third clause and we obtain a probability of 0.7 for the patient to recover under treatment.

As we see, in general, the classical distribution semantics [7] does not uniquely determine the outcome of a counterfactual query. Further, note that ProbLog programs are usually learned from observations sampled from a distribution of interest. Hence, even if we assume perfect learning, we can only ensure to obtain a program representing the correct distribution. In particular, a structure learning algorithm is not able to distinguish the programs $\mathbf{P}_{1/2}$, i.e. we cannot ensure that a learned program answers counterfactual queries correctly.

In this contribution, we present a fragment in which each program is uniquely determined by its class dependency graph and the corresponding distribution. We further argue that this yields a setting for the available structure learning methods which supports counterfactual reasoning.

2 Foundations

Here, we introduce ProbLog programs [1] and we recall how counterfactual queries are processed on them [3]. Finally, we quickly explain the design of the currently available structure learning algorithms for these programs.

As the semantics of non-ground ProbLog programs is usually defined by grounding, we restrict ourselves to the propositional case. Hence, we construct our programs from a **propositional alphabet** that is given by a finite set of propositions $\mathfrak{P}$ together with a subset $\mathfrak{E}(\mathfrak{P}) \subseteq \mathfrak{P}$ of **external proposition**. In this context, we call $\mathfrak{I}(\mathfrak{P}) := \mathfrak{P} \setminus \mathfrak{E}(\mathfrak{P})$ the set of **internal propositions**.

A **literal** l is an expression p or $\neg p$ for a proposition $p \in \mathfrak{P}$. We call l a **positive** literal if it is of the form p and a **negative** literal if it is of the form $\neg p$. Further, we call l an **external** or **internal** literal if $p \in \mathfrak{E}(\mathfrak{P})$ or $p \in \mathfrak{I}(\mathfrak{P})$ respectively. A **(logical) clause** LC is an expression $h \leftarrow b_1,...,b_n$ where $h \in \mathfrak{I}(\mathfrak{P})$ is an internal proposition called the **head** and where $\{b_1,...,b_n\}$ is a finite set of literals called the **body** of LC. Finally, a **random fact** RF is an expression $\pi(RF) :: u(RF)$ where $\pi(RF) \in [0,1]$ is the **probability** and where $u(RF) \in \mathfrak{E}(\mathfrak{P})$ is an external proposition called the **proposition** of RF.

Example 1. *Consider the alphabet* $\mathfrak{P} := \{treatment,\ recovery, u_1, u_2, u_3\}$ *with external literals* $\mathfrak{E}(\mathfrak{P})$ *given by* $\{u_1,u_2,u_3\}$. *We have that* $treatment$ *is a positive literal, whereas* $\neg treatment$ *is a negative literal. Further,* $recovery \leftarrow treatment, u_3$ *is a clause and* $0.4 :: u_3$ *is a random fact.*

Now a **logic program** is a finite set of clauses and a **ProbLog program P** is given by a logic program LP(**P**) and a set Facts(**P**) consisting of a unique random fact for every external proposition. In this case, we call LP(**P**) the **underlying logic program** of **P**. Finally, we define ($\mathfrak{P}$-)formulas ϕ and ($\mathfrak{P}$-)structures $\mathscr{M} : \mathfrak{P} \to \{True, False\}$ as usual in propositional logic. Whether a given $\mathfrak{P}$-structure $\mathscr{M}$ satisfies a formula ϕ, written $\mathscr{M} \models \phi$, is also defined as usual in propositional logic.

Example 2. *In the alphabet* $\mathfrak{P}$ *of Example 1 we can write the following ProbLog program.*

$$0.5 :: u_1 \qquad 0.5 :: u_2 \qquad 0.4 :: u_3 \qquad treatment \leftarrow u_1 \qquad recovery \leftarrow u_2 \qquad recovery \leftarrow treatment, u_3$$

As the semantics of ProbLog programs we choose the FCM-semantics [6], which supports counterfactual reasoning: For a ProbLog program **P** we define the **functional causal models semantics** or **FCM-semantics** to be the system of Boolean equations

$$\mathrm{FCM}(\mathbf{P}) := \left\{ p^{\mathrm{FCM}} := \bigvee_{\substack{LC \in \mathrm{LP}(\mathbf{P}) \\ \mathrm{head}(LC)=p}} \left(\bigwedge_{\substack{l \in \mathrm{body}(LC) \\ l \text{ internal literal}}} l^{\mathrm{FCM}} \wedge \bigwedge_{\substack{u(RF) \in \mathrm{body}(LC) \\ RF \in \mathrm{Facts}(\mathbf{P})}} u(RF)^{\mathrm{FCM}} \right) \right\}_{p \in \mathfrak{I}(\mathfrak{P})} .$$

Here, we find that the $u(RF)^{\mathrm{FCM}}$ are mutually independent Boolean random variables for every random fact $RF \in \mathrm{Facts}(\mathbf{P})$, each holding hold with probability $\pi(RF)$.

Example 3. *The FCM-semantics of the program* $\mathbf{P}$ *in Example 2 is given by*

$$treatment^{\mathrm{FCM}} := u_1^{\mathrm{FCM}} \qquad recovery^{\mathrm{FCM}} := u_2^{\mathrm{FCM}} \vee (treatment^{\mathrm{FCM}} \wedge u_3^{\mathrm{FCM}}),$$

where u_1^{FCM}, u_2^{FCM} *and* u_3^{FCM} *are mutually independent Boolean random variables holding true with a probability of* 0.5, 0.5 *and* 0.4 *respectively.*

For the rest of this section, we fix a ProbLog program $\mathbf{P}$ with an acyclic underlying logic program. Note that the FCM-semantics $\mathrm{FCM}(\mathbf{P})$ yields a unique solution for every internal proposition $p \in \mathfrak{I}(\mathfrak{P})$ in terms of the mutually independent Boolean random variables $u(RF)^{\mathrm{FCM}}$. In this way, it defines a distribution on the $\mathfrak{P}$-structures $\mathscr{M} : \mathfrak{P} \to \{True, False\}$ which coincides with the distribution semantics [7] according to Rückschloß and Weitkämper [6]. Finally, we define the probability of a formula ϕ to hold as $\pi(\phi) := \sum_{\substack{\mathscr{M}\ \mathfrak{P}\text{-structure} \\ \mathscr{M} \models \phi}} \pi(\mathscr{M})$.

However, the FCM-semantics does not only support queries about conditional and unconditional probabilities. It allows us to answer two more general causal query types, namely determining the effect of external interventions and counterfactuals [6]. Assume for instance we want to **intervene** and set a subset $\mathbf{X} \subseteq \mathfrak{I}(\mathfrak{P})$ of internal propositions to truth values specified by an assignment $\mathbf{x}$. In this case, we build a modified program $\mathbf{P}^{\mathrm{do}(\mathbf{x})}$ by erasing all clauses $LC \in \mathrm{LP}(\mathbf{P})$ with head in $\mathbf{X}$ and by adding a fact $p \leftarrow$ if $p \in \mathbf{X}$ is set to true by $\mathbf{x}$. If we now ask for the probability $\pi(\phi | \mathrm{do}(\mathbf{x}))$ of a formula ϕ to hold after setting $\mathbf{X}$ to the values $\mathbf{x}$, we query the program $\mathbf{P}^{\mathrm{do}(\mathbf{x})}$ for the probability of ϕ.

Example 4. *Assume we treat the patient in the program* $\mathbf{P}$ *of Example 2. In this case, we obtain*

$$0.5 :: u_1 \quad 0.5 :: u_2 \quad 0.4 :: u_3 \qquad treatment \leftarrow \quad recovery \leftarrow u_2 \quad recovery \leftarrow treatment, u_3$$

for the modified program $\mathbf{P}^{\mathrm{do}(treatment)}$. *This means we obtain a probability of* 0.7 *for recovery if we are the doctor and decide to treat our patient.*

Further, we do not only want to either observe or intervene, we also want to know what the probability of an event would have been if we had intervened before observing some evidence. This is especially interesting in the **counterfactual** case where our evidence contradicts the given intervention.

Example 5. *Consider the query in the introduction and observe that the evidence* $\{\neg treatment, recovery\}$ *contradicts the intervention* $\mathrm{do}(treatment)$, *i.e. this is a counterfactual query.*

Hence, fix another subset of internal propositions $\mathbf{E} \subseteq \mathfrak{I}(\mathfrak{P})$ and assume we observe the evidence that the propositions in $\mathbf{E}$ take values according to the assignment $\mathbf{e}$. We now ask for the probability $\pi(\phi | \mathbf{e}, \mathrm{do}(\mathbf{x}))$ of the formula ϕ to hold if we had set the propositions in $\mathbf{X}$ to the values specified by $\mathbf{x}$ before observing our evidence $\mathbf{e}$. To answer queries like that we proceed as Kiesel et al. [3]:

First we generate two copies $\mathfrak{I}(\mathfrak{P})^{e/i}$ of the set of internal propositions – one to handle the evidence and the other to handle the interventions. Further, we set $u^{e/i} := u$ for every external proposition $u \in \mathfrak{E}(\mathfrak{P})$. Note that this yields maps $\_^{e/i}$ of literals, clauses, programs etc. We define the **counterfactual semantics** of $\mathbf{P}$ to be the ProbLog program $\mathbf{P}^K$ which consists of the logic program $\mathbf{L}(\mathbf{P})^e \cup \mathbf{L}(\mathbf{P})^i$ and the random facts $\mathrm{Facts}(\mathbf{P})$. Now we intervene in $\mathbf{P}^K$ and set the proposition in $\mathbf{X}^i$ to the truth values specified by $\mathbf{x}$ to obtain the program $\mathbf{P}^{K,\mathrm{do}(\mathbf{x})}$. Finally, we query the program $\mathbf{P}^{K,\mathrm{do}(\mathbf{x})}$ for the probability $\pi(\phi^i|\mathbf{E}^e = \mathbf{e})$ to obtain the desired result for $\pi(\phi|\mathbf{e},\mathrm{do}(\mathbf{x}))$.

Example 6. *Assume we did not treat the patient in the program* $\mathbf{P}$ *of Example 2 and he recovered. What is the probability* $\pi(recovery|\neg treatment, recovery, \mathrm{do}(treatment))$ *that he would have recovered, if he had been treated? To answer this question we query the program* $\mathbf{P}^{K,\mathrm{do}(treatment)}$

$$\begin{array}{lll} 0.5 :: u_1 & 0.5 :: u_2 & 0.4 :: u_3 \\ treatment^i \leftarrow & recovery^i \leftarrow u_2 & recovery^i \leftarrow treatment^i, u_3 \\ treatment^e \leftarrow u_1 & recovery^e \leftarrow u_2 & recovery^e \leftarrow treatment^e, u_3 \end{array}$$

for the probability $\pi(recovery^i|recovery^e, \neg treatment^e) = 1$.

This procedure automates Pearl's counterfactual reasoning [4] and is implemented in the WHATIF-solver of Kiesel et al. [3].

In this contribution, we restrict ourselves to ProbLog programs, which can be represented with ProbLog clauses. A **ProbLog clause** RC is an expression $\pi(RC) :: \mathrm{effect}(RC) \leftarrow causes(RC)$, where $\mathrm{effect}(RC) \in \mathfrak{I}(\mathfrak{P})$ is an internal proposition called the **effect**, where $\mathrm{causes}(RC)$ is a finite set of internal literals called the **causes** and where $0 \leq \pi(RC) \leq 1$ is a number called the **probability** of RC. The ProbLog clause RC is an abbreviation for the following pair of a random fact and a logical clause.

$$RF(RC) := (\pi(RC) :: u(RC)) \qquad LC(RC) := (h \leftarrow b_1, ..., b_n, u(RC)),$$

where $u(RC) \in \mathfrak{E}(\mathfrak{P})$ is a distinct external literal. From now on, by abuse of language, a **ProbLog program** $\mathbf{P}$ is a finite set of ProbLog clauses, i.e. a ProbLog program consisting of the logic program $\mathbf{L}(\mathbf{P}) := \{LC(RC) : RC \in \mathbf{P}\}$ and of the random facts $\mathrm{Facts}(\mathbf{P}) := \{RF(RC) : RC \in \mathbf{P}\}$.

Example 7. *Observe that the program* $\mathbf{P}_1$ *from the introduction is an abbreviation for the ProbLog program in Example 2.*

The **class dependency graph** $\mathrm{Graph}(\mathbf{P})$ of a ProbLog program $\mathbf{P}$ is the directed graph on the internal propositions $\mathfrak{I}(\mathfrak{P})$ obtained by drawing an edge $p_1 \to p_2$ if and only if there exists a ProbLog clause $RC \in \mathbf{P}$ with a cause p_1 or $\neg p_1$ and with effect p_2. We say that the program $\mathbf{P}$ is **acyclic** if its class dependency graph $\mathrm{Graph}(\mathbf{P})$ is a directed acyclic graph. Moreover, the program $\mathbf{P}$ is **positive** if the causes of every ProbLog clause $RC \in \mathbf{P}$ form a set of positive literals.

Example 8. *The class dependency graph of the programs* $\mathbf{P}_{1/2}$ *in the introduction is given by the edge* $treatment \to recovery$. *Further, program* $\mathbf{P}_1$ *is positive whereas* $\mathbf{P}_2$ *is not.*

Next, we quickly recall the overview over the structure learning techniques for propositional ProbLog programs from Riguzzi [5, §10]. Given suitable data, all those algorithms search the space of programs defined by a language bias and background knowledge with heuristics relying on statistical tests.

Here, the **language bias** defining the clause and therefore the program space is defined by mode declarations of the form $modeb\,(*, q)$ and $modeh\,(*, p)$ as well as declarations of the form $determination(p, q)$

for propositions p and q. A positive ProbLog clause $\pi :: \text{effect}(RC) \leftarrow \text{causes}(RC)$ lies in the language defined by our bias if we declared $modeh(*, \text{effect}(RC))$, $modeb(*, c)$ for all c in $\text{causes}(RC)$ and $determination(\text{effect}(RC), c)$ for all c in $\text{causes}(RC)$.

Moreover, we can express **background knowledge** in a logic program defining further propositions in terms of the given data. This is also the way how one can learn clauses with negation by adding a clause $neg\_p \leftarrow \neg p$ to the background knowledge for every proposition p. We call the pair of a language bias and a background knowledge the **setting** for a structure learning algorithm.

Example 9. *To learn the program $\boldsymbol{P}_1$ of the introduction we nee to specify the setting $\mathfrak{S}_1$ which consists of the bias $modeh(*, recovery)$, $modeb(*, treatment)$, $determinantion(recovery, treatment)$ and an empty background knowledge. If we want to consider the program $\boldsymbol{P}_2$ as well, we additionally need the declarations $modeb(*, neg\_treatment)$ and $determinantion(recovery, neg\_treatment)$ together with the the clause $neg\_treatment \leftarrow \neg treatment$ in the background knowledge resulting in the setting $\mathfrak{S}_2$.*

Note that via the $determination/2$ predicate the language bias essentially provides the class dependency graph, i.e. the corresponding cause-effect relationships, of our program as prior knowledge to the structure learning algorithm.

3 Results

Generally, in structure learning, from some prior knowledge encoded by a setting one wants to derive a program that describes a given set of data. In most of the cases, the data consists of observations. We additionally assume that our data consists of samples drawn from the distribution induced by a hidden ProbLog program $\tilde{\mathbf{P}}$ of interest and the prior knowledge consists of a language bias, i.e. of the class dependency graph of $\tilde{\mathbf{P}}$. Further, to decide how good a candidate program $\mathbf{P}$ represents our dataset we process statistical tests. However, statistical tests only measure how well the induced distribution of the program $\mathbf{P}$ fits a given set of observations. They generally reveal no information about the causal mechanism generating our data. Hence, we cannot measure whether the causal mechanism represented by a candidate ProbLog program $\mathbf{P}$ coincides with the causal mechanism underlying our data, i.e. with the causal mechanism described by $\tilde{\mathbf{P}}$.

Example 10. *Consider the programs $\boldsymbol{P}_{1/2}$ of the introduction. While they both represent different causal models yielding to different counterfactual estimations, they yield the same distribution semantics and share the same class dependency graph.*

Hence, if we take $\tilde{\boldsymbol{P}} := \boldsymbol{P}_{1/2}$ for the hidden program, we sample from the same distribution in both cases. That means that even with the correct language bias a structure learning algorithm cannot determine which of the two programs actually generated the provided data unless it is given further knowledge.

More drastically, Example 10 illustrates that without further prior knowledge, even under the assumption of perfect learning, it is only possible to learn a program $\mathbf{P}$ which is ensured to represent the correct distribution. In particular, we expect that a learned program $\mathbf{P}$ does not necessarily answer counterfactual queries correctly.

In the following, we study the fragment of acyclic proper positive ProbLog programs in normal form. A ProbLog program $\mathbf{P}$ is **proper in normal form** if every clause $RC \in \mathbf{P}$ has a probability $0 < \pi(RC) < 1$, if any two distinct clauses $RC_{1/2} \in \mathbf{P}$ have distinct causes $\text{causes}(RC_1) \neq \text{causes}(RC_2)$ or distinct effects $\text{effect}(RC_1) \neq \text{effect}(RC_2)$ and if every sink s in the class dependency graph gives rise to a random fact $\alpha :: s$. The main result of this contribution now states that all programs lying in this fragment are uniquely determined by their class dependency graph and their underlying distribution.

Theorem 1. *Every acyclic proper positive ProbLog program in normal form* $\mathbf{P}$ *can be reconstructed from its class dependency graph* $\mathrm{Graph}(\mathbf{P})$ *and the induced distribution* π.

Proof. We proceed by induction on the number n of nodes in the class dependency graph $\mathrm{Graph}(\mathbf{P})$.

$n = 1$: In this case, the program $\mathbf{P}$ consists only of one clause $\pi :: p \leftarrow$. Hence, we set $\pi := \pi(p)$ and we are done.

$n > 1$: Choose a sink $h \in \mathfrak{P}$ of $\mathrm{Graph}(\mathbf{P})$. Further, denote by $\mathbf{P} \setminus h$ the program that results from $\mathbf{P}$ if we erase all clauses with effect h. By maximality h does not occur in the causes of any other clause, i.e. $\mathbf{P} \setminus h$ induces the same distribution on $\mathfrak{I}(\mathfrak{P}) \setminus \{h\}$ as the program $\mathbf{P}$ and it has the graph $\mathrm{Graph}(\mathbf{P}) \setminus h$ as its class dependency graph. Here, $\mathrm{Graph}(\mathbf{P}) \setminus h$ denotes the graph that results from $\mathrm{Graph}(\mathbf{P})$ if we erase the node h together with all edges pointing into it. Now, by the induction hypothesis we can reconstruct the program $\mathbf{P} \setminus h$ from the given data.

Hence, we are left to reconstruct the clauses defining h itself. Note that the parents $b \in \mathrm{pa}(h)$ of h in $\mathrm{Graph}(\mathbf{P})$ are the only propositions that may occur in the body of a clause defining h. Further, note that each of these occurrences is positive. We consider the function

$$\mathrm{Ind}_h^{\mathbf{P}} : \mathscr{P}(\mathrm{pa}(h)) \to [0,1] \qquad T \mapsto \pi(h | \{t, \neg s : t \in T,\ s \in \mathrm{pa}(h) \setminus T\}),$$

where $\mathscr{P}(\_)$ denotes the power set operator. Recall from Pearl [4, §3] that observing the parents of h is the same as intervening on them. Hence, we see that

$$Ind_h^{\mathbf{P}}(T) := \pi\left(\bigvee_{\substack{RC \in \mathbf{P} \\ \mathrm{causes}(RC) \subseteq T \\ \mathrm{effect}(RC) = h}} u(RC) \right) = \sum_{\substack{RC_1, \ldots, RC_k \in \mathbf{P} \\ k \in \mathbb{N},\ \mathrm{causes}(RC) \subseteq T \\ \mathrm{effect}(RC) = h}} (-1)^k \prod_{i=1}^{k} \pi(RC_i) \tag{1}$$

Now, it is easy to see that $T \subseteq \mathrm{pa}(\mathbf{P})$ are the causes of a clause in $\mathbf{P}$ if and only if $\mathrm{Ind}_h^{\mathbf{P}}(S) < \mathrm{Ind}_h^{\mathbf{P}}(T)$ for all $S \subseteq \mathrm{pa}(h)$ with $S \subsetneq T$. Further, we obtain the parameters of $\mathbf{P}$ by recursion: We find that $\pi(RC) = \mathrm{Ind}_h^{\mathbf{P}}(\mathrm{body}(RC))$ for every clause $RC \in \mathbf{P}$ with a minimal body. Further, in the recursion step Equation (1) yields a one-dimensional linear equation for the parameter of interest.

□

In a forthcoming paper, we prove that the answers to all counterfactual queries uniquely determine a proper ProbLog program in normal form. Thus if we want to answer counterfactual queries based on a learned program $\mathbf{P}$, we almost need to fully reconstruct the hidden program $\tilde{\mathbf{P}}$.

Remark 1. *If we estimate the functions* $Ind_h^{\mathbf{P}}$ *using relative frequencies, we obtain a structure learning algorithm that recovers an acyclic proper positive ProbLog program in normal form from a known causal structure and a sufficiently large set of samples. The resulting distributions (one for every counterfactual query) are guaranteed to converge in probability. Further, the complexity is exponential in the maximal number of parents of a node in the class dependency graph and linear in the size of the alphabet* $\mathfrak{P}$.

Finally, assume we apply a structure learning algorithm [5, §10] with a language bias encoding the class dependency graph $\mathrm{Graph}(\tilde{\mathbf{P}})$ to obtain a program $\mathbf{P}$. Let us further assume that we learned perfectly, i.e. that the program $\mathbf{P}$ encodes the same distribution as $\tilde{\mathbf{P}}$. If we now assume that both programs $\mathbf{P}$ and $\tilde{\mathbf{P}}$ are acyclic proper positive ProbLog programs in normal form, Theorem 1 yields that $\mathbf{P}$ and $\tilde{\mathbf{P}}$ coincide, i.e. $\mathbf{P}$ expresses the full causal content of $\tilde{\mathbf{P}}$.

Since without background knowledge each currently available structure learning algorithm [5, §10] only searches for positive programs fitting a given dataset, the assumption to learn proper positive ProbLog programs in normal form is realized easily. Causally, the absence of background knowledge implies that we assume our data to be generated by a proper positive ProbLog program in normal form.

Corollary 1. *Assume we are given data sampled from a hidden proper positive ProbLog program in normal form* $\tilde{\mathbf{P}}$ *and assume we are aware of the class dependency graph* $\mathrm{Graph}(\tilde{\mathbf{P}})$ *of* $\tilde{\mathbf{P}}$*. Every structure learning algorithm that is able to learn a proper positive ProbLog program in normal form with the correct class dependency graph and the correct distribution reconstructs* $\tilde{\mathbf{P}}$ *from the provided data. In particular, the result of such a structure learning algorithm supports counterfactual reasoning.* □

4 Conclusion

In the introduction, we show that the distribution semantics does not uniquely determine counterfactual query outcomes for ProbLog programs, making it unfeasible to use the currently available structure learning algorithms for counterfactual reasoning. However, our main result reveals that proper positive ProbLog programs in normal form are actually uniquely determined by their distribution semantics and their class dependency graph. Hence, if applied without background knowledge and if we assume perfect learning, the currently available structure learning algorithms can recover these programs when provided with the correct language bias i.e. they support counterfactual reasoning in this setting.

In a forthcoming paper, we show that counterfactual reasoning uniquely determines a proper program in normal form, i.e. it is not sufficient to learn programs under a coarser notion of equivalence. Determining the equivalence classes of ProbLog programs representing the same distributions and predicting the behaviour of the available structure learning algorithm for more general fragments of ProbLog are promising directions for future work extending this contribution.

References

[1] Luc De Raedt, Angelika Kimmig & Hannu Toivonen (2007): *ProbLog: A Probabilistic Prolog and Its Application in Link Discovery*. In: 20th International Joint Conference on Artificial Intelligence, 7, AAAI Press, Hyderabad, India, pp. 2462–2467, doi:`10.5555/1625275.1625673`.

[2] Nicole Van Hoeck (2015): *Cognitive Neuroscience of Human Counterfactual Reasoning*. Frontiers in Human Neuroscience 9, doi:`10.3389/fnhum.2015.00420`.

[3] Rafael Kiesel, Kilian Rückschloß & Felix Weitkämper (2023): *"What if?" in Probabilistic Logic Programming*. Accepted for TPLP Proceedings of ICLP 2023. Available at `https://arxiv.org/abs/2305.15318`.

[4] Judea Pearl (2000): *Causality*, 2 edition. Cambridge University Press, Cambridge, UK, doi:`10.1017/CBO9780511803161`.

[5] Fabrizio Riguzzi (2020): *Foundations of Probabilistic Logic Programming: Languages, Semantics, Inference and Learning*. River Publishers, doi:`10.1201/9781003338192`.

[6] Kilian Rückschloß & Felix Weitkämper (2022): *Exploiting the Full Power of Pearl's Causality in Probabilistic Logic Programming*. In: Proceedings of the International Conference on Logic Programming 2022 Workshops, CEUR Workshop Proceedings 3193, CEUR-WS.org, Haifa, Israel. Available at `http://ceur-ws.org/Vol-3193/paper1PLP.pdf`.

[7] Taisuke Sato (1995): *A Statistical Learning Method for Logic Programs with Distribution Semantics*. In: Logic Programming: The 12th International Conference, The MIT Press, Tokyo, Japan, pp. 715–729, doi:`10.7551/mitpress/4298.003.0069`.

Multiple Query Satisfiability
of Constrained Horn Clauses

Emanuele De Angelis
IASI-CNR, Rome, Italy
`emanuele.deangelis@iasi.cnr.it`

Fabio Fioravanti
DEc, University 'G. d'Annunzio', Chieti-Pescara, Italy
`fabio.fioravanti@unich.it`

Alberto Pettorossi
DICII, University of Rome 'Tor Vergata', Italy
`pettorossi@info.uniroma2.it`

Maurizio Proietti
IASI-CNR, Rome, Italy
`maurizio.proietti@iasi.cnr.it`

We address the problem of checking the satisfiability of a set of constrained Horn clauses (CHCs) possibly including more than one query. We propose a transformation technique that takes as input a set of CHCs, including a set of queries, and returns as output a new set of CHCs such that the transformed CHCs are satisfiable if and only if so are the original ones. The transformed CHCs incorporate in each new query suitable information coming from the other ones so that the CHC satisfiability algorithm is able to exploit the relationships among all queries. We show that our proposed technique is effective on a non trivial benchmark of sets of CHCs that encode many verification problems for programs manipulating algebraic data types such as lists and trees.

1 Introduction

Constrained Horn Clauses (CHCs) are a logical formalism very well suited for automatic program verification and analysis [1]. Indeed, many verification problems can be reduced to problems of checking satisfiability of CHCs and several effective CHC *solvers* are currently available as back-end tools for program verification purposes [5, 6].

Following the CHC-based verification approach, a program is translated into a set of definite CHCs (that is, clauses whose head is different from *false*), which capture the semantics of the program, together with a set of queries (that is, clauses whose head is *false*), which specify the program properties to be verified. Very often the program includes several functions, each one having its own contract (that is, a pair of a pre-condition and a post-condition) and the CHC translation of the verification problem generates several queries.

CHC solvers try to show the satisfiability of a set of CHCs of the form: $P \cup \{\mathit{false} \leftarrow G_1, \ldots, \mathit{false} \leftarrow G_n\}$, where P is a set of definite CHCs, and $\mathit{false} \leftarrow G_1, \ldots, \mathit{false} \leftarrow G_n$ are queries, by trying to show in a separate way the satisfiability of each set $P \cup \{\mathit{false} \leftarrow G_i\}$, for $i = 1, \ldots, n$. This approach may not always be effective, as the solver may not be able to exploit, possibly mutual, dependencies among the various queries.

2 The Transformation Algorithm

In this paper we propose a CHC transformation technique that, given a set $P \cup \{\mathit{false} \leftarrow G_1, \ldots, \mathit{false} \leftarrow G_n\}$ of CHCs, derives an *equisatisfiable* set $P' \cup \{\mathit{false} \leftarrow G'_1, \ldots, \mathit{false} \leftarrow G'_n\}$, for whose satisfiability

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 65–68, doi:10.4204/EPTCS.385.8

proof a CHC solver may exploit the mutual interactions among the *n* satisfiability proofs, one for each query.

The CHC transformation algorithm T_{mq} we present here, is based on adaptations of the usual unfold/fold transformation rules for CHCs (and CLP programs), and on a novel rule, called *Query-based Strengthening*, specifically designed for incorporating into the clauses relative to a particular query Q some additional properties and constraints derived from other queries that are relative to predicates on which the predicates in Q depend.

Algorithm T_{mq} is both sound and complete, that is, the transformed clauses are satisfiable if and only if so are the original ones. The completeness of T_{mq} is very important because if a property does *not* hold, it allows us to infer the unsatisfiability of the original clauses, and hence to deduce the invalidity of the property to be verified.

The proposed algorithm improves over the T_{cata} algorithm presented in a previous paper [4] which works by *eliminating* ADTs from sets of CHCs. T_{cata} is only sound (indeed, it can be seen as computing an *abstraction* of the initial clauses), and thus if the transformed clauses are unsatisfiable we cannot infer anything about the satisfiability of the initial CHCs. The new algorithm T_{mq} does not eliminate ADT variables, but the additional constraints it discovers are often very beneficial to the CHC solvers when trying to check the satisfiability of a given set of clauses, thereby enhancing their ability to verify program properties.

In order to define a class of CHCs for which our transformation algorithm always terminates, we have considered predicates for expressing properties that are *catamorphisms*, that is, predicates that are total and functional with respect to some of their arguments, and have a somewhat restricted recursive structure.

3 Experimental Evaluation

We have implemented algorithm T_{mq} in a tool, called VeriCaT$_{mq}$, which extends VeriCaT [4] by guaranteeing a sound and complete transformation. VeriCaT$_{mq}$ is based on: (i) VeriMAP [2] for transforming CHCs, and (ii) SPACER [6] (with Z3 4.11.2) to check the satisfiability of the transformed CHCs.

We have considered 170 problems, as sets of CHCs, with 470 queries in total, equally divided between the class of satisfiable (sat) problems and unsatisfiable (unsat) problems (for each class we have 85 problems and 235 queries).

The problems we considered refer to programs that manipulate: (i) lists of integers by performing concatenation, permutation, reversal, and sorting, and (ii) binary search trees, by inserting and deleting elements. For list manipulating programs, we have considered properties encoded by catamorphisms such as: list length, minimum and maximum element, sum of elements, list content as sets or multisets of elements, and list sortedness (in ascending or descending order). For trees, we have considered tree size, tree height, minimum and maximum element, tree content and the binary search tree property.

In addition to satisfiable problems, we have also considered unsatisfiable problems that have been obtained from their satisfiable counterparts by introducing bugs in the programs: for instance, by not inserting an element in a list, or adding an extra constraint, or replacing a non-empty tree by an empty one.

In summary, VeriCaT$_{mq}$ was able to prove all the 170 considered problems whereas SPACER was able to prove the properties of 84 ‘unsat’ problems out of 85, and none of the ‘sat’ problems. The total time needed for transforming the CHCs was 275 seconds (1.62 s per problem, on average), and checking

Program	Problems	Queries	SPACER		VeriCaT$_{mq}$	
			sat	unsat	sat	unsat
List Membership	2	6	0	1	1	1
List Permutation	8	24	0	4	4	4
List Concatenation	18	18	0	8	9	9
Reverse	20	40	0	10	10	10
Double Reverse	4	12	0	2	2	2
Reverse w/Accumulator	6	18	0	3	3	3
Bubblesort	12	36	0	6	6	6
Heapsort	8	48	0	4	4	4
Insertionsort	12	24	0	6	6	6
Mergesort	18	84	0	9	9	9
Quicksort (version 1)	12	38	0	6	6	6
Quicksort (version 2)	12	36	0	6	6	6
Selectionsort	14	42	0	7	7	7
Treesort	4	20	0	2	2	2
Binary Search Tree	20	24	0	10	10	10
Total	170	470	0	84	85	85

Table 1: Programs and problems proved by SPACER and VeriCaT$_{mq}$.

the satisfiability of the transformed CHCs took about 174 s in total (about 1s average time, 0.10 s median time). For comparison, SPACER took 30.27 s for checking the unsatisfiability of 84 problems (0.36 s average time, 0.15 s median time). The benchmark and the tool are available at `https://fmlab.unich.it/vericatmq`.

For instance, for all the list sorting programs we have considered (Bubblesort, Heapsort, Insertionsort, Mergesort, Quicksort, Selectionsort and Treesort), VeriCaT$_{mq}$ was able to prove properties stating that the output list is sorted and has the same multiset of elements of the input list. Similarly, VeriCaT$_{mq}$ was able to prove that those properties do *not* hold, if extra elements are added to the output list, or some elements are not copied from the input list to the output list, or a wrong comparison operator is used.

The results obtained by the prototype implementation of our method are encouraging and show that, when used in combination with state-of-the-art CHC solvers, it can greatly improve their effectiveness to prove satisfiability of sets of CHCs with multiple queries, while it does not inhibit their remarkable ability to prove unsatisfiability, although some extra time due to the transformation process may be required.

Full details about the contributions of this paper can be found in the original publication [3].

References

[1] E. De Angelis, F. Fioravanti, J. P. Gallagher, M. V. Hermenegildo, A. Pettorossi & M. Proietti (2022): *Analysis and Transformation of Constrained Horn Clauses for Program Verification*. *Theory and Practice of Logic Programming* 22(6), pp. 974–1042, doi:10.1017/S1471068421000211.

[2] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2014): *VeriMAP: A Tool for Verifying Programs through Transformations*. In: *Tools and Algorithms for the Construction and Analysis of Systems, TACAS '14*, Lecture Notes in Computer Science 8413, Springer, pp. 568–574, doi:10.1007/978-3-642-54862-8_47.

[3] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2023): *Multiple Query Satisfiability of Constrained Horn Clauses*. In M. Hanus & D. Inclezan, editors: *Practical Aspects of Declarative Languages - 25th International Symposium, PADL 2023, Boston, MA, USA, Proceedings*, *Lecture Notes in Computer Science* 13880, Springer, pp. 125–143, doi:10.1007/978-3-031-24841-2_9.

[4] E. De Angelis, M. Proietti, F. Fioravanti & A. Pettorossi (2022): *Verifying Catamorphism-Based Contracts using Constrained Horn Clauses*. Theory and Practice of Logic Programming 22(4), pp. 555–572, doi:10.1017/S1471068422000175.

[5] H. Hojjat & Ph. Rümmer (2018): *The ELDARICA Horn Solver*. In: Formal Methods in Computer Aided Design, FMCAD '18, IEEE, pp. 1–7, doi:10.23919/FMCAD.2018.8603013.

[6] A. Komuravelli, A. Gurfinkel & S. Chaki (2016): *SMT-Based Model Checking for Recursive Programs*. Formal Methods in System Design 48(3), pp. 175–205, doi:10.1007/s10703-016-0249-4.

Towards One-Shot Learning for Text Classification using Inductive Logic Programming

Ghazal Afroozi Milani, Daniel Cyrus, Alireza Tamaddoni-Nezhad
Department of Computer Science, University of Surrey, United Kingdom
`{g.afroozimilani,d.cyrus,a.tamaddoni-nezhad}@surrey.ac.uk`

With the ever-increasing potential of AI to perform personalised tasks, it is becoming essential to develop new machine learning techniques which are data-efficient and do not require hundreds or thousands of training data. In this paper, we explore an Inductive Logic Programming approach for one-shot text classification. In particular, we explore the framework of Meta-Interpretive Learning (MIL), along with using common-sense background knowledge extracted from ConceptNet. Results indicate that MIL can learn text classification rules from a small number of training examples, even one example. Moreover, the higher complexity of chosen example for one-shot learning, the higher accuracy of the outcome. Finally, we utilise two approaches, Background Knowledge Splitting and Average One-Shot Learning, to evaluate our model on a public News Category dataset. The outcomes validate MIL's superior performance to the Siamese net for one-shot learning from text.

1 Introduction

Machine learning, in particular, Deep Neural Networks (DNNs), has achieved outstanding outcomes in numerous real world applications such as image and text classification [7, 20]. However, these learning methods require a huge number of training instances, which is not always possible to be provided in advance. An automated software tool, for example, should be able to learn from a small number of interactions with the user in order to be efficiently customised with every new user's requirement. Inductive Logic Programming (ILP) and, in particular, Meta Interpretive Learning (MIL) can learn human-readable hypotheses from a small amount of training data [17, 24]. This capability is very promising for medical and industrial usage, especially when we do not have access to a large amount of training data. The main contribution of this study is to develop and evaluate a new learning algorithm, Meta-Goal Learner (MGL), based on the framework of Meta-Interpretive Learning (MIL), which can be utilised for one-shot learning from textual data along with using a common-sense knowledge-base (i.e. ConceptNet) as background knowledge for learning.

2 Inductive Logic Programming

Inductive Logic Programming (ILP) is a subfield of symbolic artificial intelligence, which works based on inductive reasoning to detect generalisations, rules, or models [13, 18]. In this method, the system learns from training examples with the help of background information and using logic programming [24]. Indeed, the primary purpose of ILP [13], like the other types of machine learning, is to induce a model/ hypothesis that can generalise training examples but, unlike them, uses logic programs to represent data and learns relations [2].

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 69–79, doi:10.4204/EPTCS.385.9

Using logic programs brings several benefits to ILP. Firstly, as they represent hypotheses in ILP and are related to relational databases, they can support relational data like graphs. On the other hand, because of its expressivity, it can learn complex relational theories like cellular automata, event calculus theories, and Petri nets [2]. Secondly, it is similar to natural language; therefore, readable and valuable for explainable AI and scientific discovery ([17]).

Every standard ILP setting consists of three input sets, including Background Knowledge (B), positive training examples (E^+), and negative training examples (E^-) [11, 13]. The output is a Hypothesis H such that:

- It is necessary $\forall$ e $\in E^+$ $B \not\models$ e
- H is complete/ sufficient if $\forall$ e $\in E^+$ and $H \wedge B \models$ e
- H is strong consistent if $\forall$ e $\in E^-$ and $H \wedge B \not\models$ e
- H is weak consistent if $H \wedge B \not\models \Box$

2.1 Meta-Interpretive Learning

Meta-Interpretive Learning (MIL) [15] is a type of Inductive Logic Programming which allows learning logic programs from background knowledge, training examples and a declarative bias called metarules. Metarules are datalog clauses with variables quantified over predicate symbols (i.e., second-order variables) [19, 10].

The advantage of MIL is its capability to (1) automatically introduce new sub-definitions when it learns a predicate definition [16, 15]. This is referred to as Predicate Invention. (2) learning recursive clauses. MIL uses metarules to restrict hypotheses space. In other words, a meta-rule is a higher-order expression in which the form of permitted clauses in the hypothesis space is instructed[16, 12].

A **metarule** is a higher-order well formed formula as follows:

$$\exists\sigma\forall\tau P(s_1,...,s_m) \leftarrow ..., Q_i(t_1,...,t_n), ...$$

where σ, τ are disjoint variables, ρ and ζ are sets of predicate symbols and constants respectively $P, Q_i \in \sigma \cup \tau \cup \rho$ and $s_1,...,s_m,t_1,...,t_n \in \sigma \cup \tau \cup \zeta$. Metarules are shown normally without quantifiers as: $P(s_1,...,s_m) \leftarrow ..., Q_i(t_1,...,t_n), ...$. P and Q_i are existentially quantified higher-order variables, while the other variables present universally quantified variables.

The standard MIL problem [15] is defined as follows: Given metarules (M), definite program background knowledge (B) and ground positive and negative unit examples (E^+, E^-), **MIL** returns a higher-order datalog program hypothesis (H) if one exists such that: $M \wedge B \wedge H \models E^+$ and $M \wedge B \wedge H \not\models E^-$ is consistent.

In ILP, selecting proper Background Knowledge (B) is essential for appropriate learning performance. However, this could be difficult or expensive as often this should be done by experts. There are two main problems related to having a balance background knowledge, including too little and too much Background Knowledge. If using too little Background Knowledge, we might lose the target hypothesis. Having too much irrelevant Background Knowledge is another ILP challenge that can increase hypothesis space size and decrease learning performance ([4]).

Category	Source	Relations	Example
Commonsense KGs	ConceptNet	37	mother-related_to-family
	Atomic	9	Person X bakes bread-xEffect-eat food
	Glucose	10	someone A -eats- something A
	WebChild	4groups	eating -type of- consumption
	Quasimodo	78636	pressure cooker-cook faster-food
	SenticNet	1	cold food -polarity- negative
	HasPartKB	1	dairy food -has part- vitamin
	Probase	1	apple -is a- food
	Isacore	1	snack food -is a- food
Common KGs	Wikidata	6.7k	food -has quality- mouthfeel
	YAGO4	116	banana chip -rdf:type- food
	DOLCE	1	n/p
	SUMO	1614	food -hyponym- food_product
Lexical resources	WordNet	10	food -hyponym- comfort food
	Roget	2	dish -synonym- food
	FrameNet	8(f2f)	eating -evoke- Ingestion
	MetaNet	14(f2f)	food -has role- food_consumer
	VerbNet	36(roles)	eating -haspatient- comestible
Visual sources	Visual Genome	42374	food -on- plate
	Flickr30k	1	eating place -confers with- their kitchen
Corpora& LMs	GenericsKB	n/a	Animals search for food
	GPT-2	n/a	eating at home will not lead to weight gain
	ChatGPT	n/a	Food provides nourishment: People eat food to.. (huge amount of information about eating)

Table 1: Overview of common-sense knowledge sources partly taken from [6]

There are two methods to overcome this issue, one is to enable an ILP system to invent a new predicate instead of missing background knowledge [23]. The second method is to use transfer learning to discover knowledge that can help mitigate the effects of background knowledge shortage [8].

3 Commonsense Knowledge and ConceptNet

Commonsense knowledge is an ordinary information that helps people make sense of everyday situations without mentioning them in their communications. Due to its implicitness, capturing this knowledge is beneficial for designing an effective human-computer interface and various types of Artificial Intelligence (AI) [6, 5]. These common-sense knowledge sources are in various forms and about different types of knowledge. Ilievski et al. [6] divided them into five categories, common-sense knowledge graphs (e.g. ConceptNet and Atomic), common knowledge graphs and ontologies(e.g. Wikidata and Yago), lexical resources (e.g. WordNet and Roget), visual common-sense sources (e.g. Visual Genome), and Corpora and language models (e.g. GenericsKB and Language models) as shown in Table 1.

ConceptNet is a multilingual common-sense knowledge graph derived from the crowd-sourced Open Mind Common Sense project and knowledge from other resources like WordNet, DBPedia, Games with

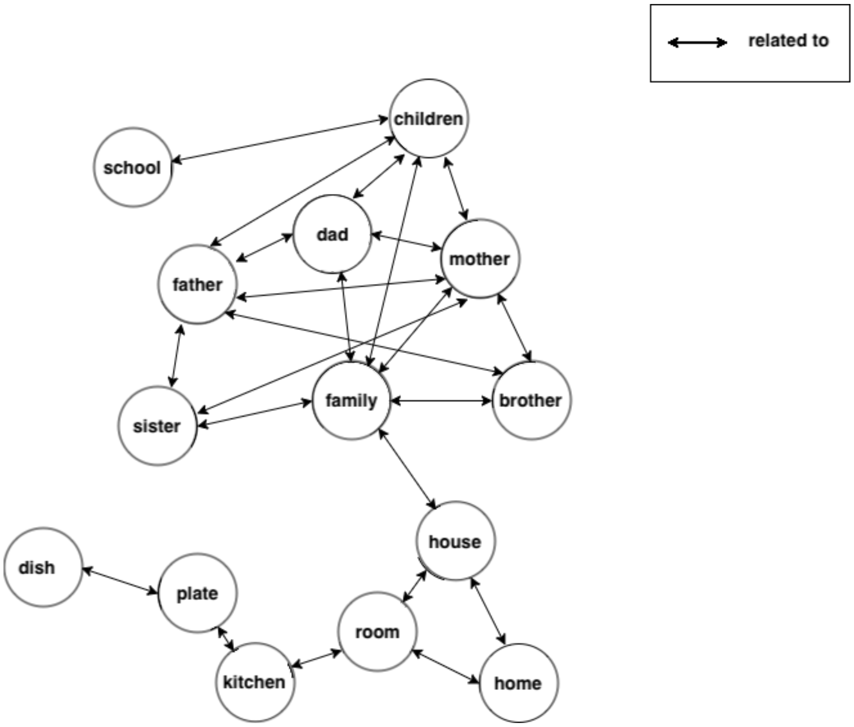


Figure 1: An illustration of ConceptNet used by our system.

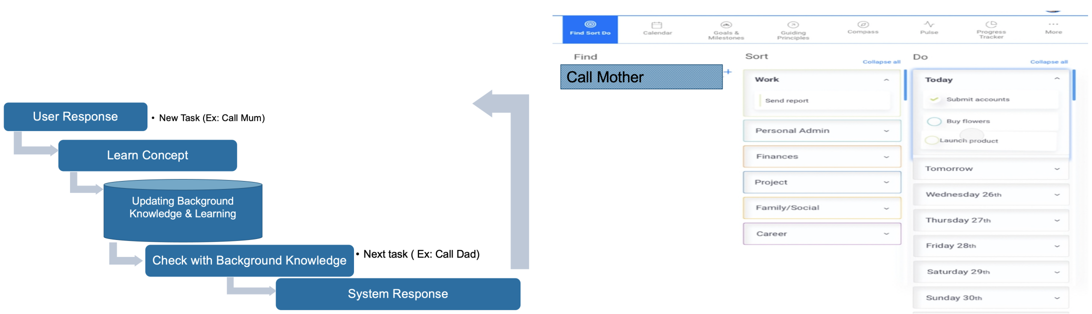
a purpose, Wiktionary, OpenCyc and JMDict, connecting words and terms of natural language as nodes with labelled weighted edges with 37 types of relations [6].

The standard definition of a Common-sense Knowledge Graph is as follows [26]:
G= (V,E,R) is a Common-sense Knowledge Graph, where V, E and R are the node, edge and relation sets. Edges comprise triplets (h, r, t) where h and t are head and tail entities connected by relation $r : E = (h,r,t) \mid h \in V, t \in V, r \in R$, and nodes are along with a free-text description.

These node relations in ConceptNet5.5 have been categorised [21] to two sub-classes: Symmetric relations like LocatedNear, RelatedTo, SimilarTo, and Asymmetric relations such as AtLocation, CapableOf, CreatedBy, DefinedAs, HasA, and Entails [6]. We use relations of "RelatedTo", which demonstrate an undefined meaning relation between two words. As shown in Figure 1, every node represents a word extracted from ConceptNet and connected to another node by a bidirectional edge. These symmetric edges show the relatedness of the connected words and, therefore, are bidirectional.

4 One-shot text classification using ILP and MIL

The basis of an interactive learning system is to learn from the user. This challenge requires incremental learning techniques that help the system to learn from time to time. In this section, we introduce Meta-Goal Learner (MGL), a novel machine learning from user interactions. We explore Metagol[3] learning engine for the task of few-shot learning in MGL.



(a) Model Prototype (b) MGL illustration

Figure 2: Meta-Goal Learner.

MGL initially fetches the words of the task (i.e. sentence given by the user) and obtains relevant background knowledge from ConceptNet. The system can then learn and categorise the upcoming tasks, as shown in Figure 2. For instance, when a user writes a task (e.g. "call mother"), the system will wait for the first answer/ label from the user and then will consider it as a positive example. In this case, the positive example represents the task and its category name. The system then generates relevant background knowledge as follows:

1. Tokenise the sentence and generate clauses as part of the background knowledge:
   e.g., contains(X,call). contains(X,mum).
2. Send a request to ConceptNet web API for fetching the relations of the words separately and adding them to the background knowledge:
   e.g., related_to(call, phone). related_to(mother, family).

Given the positive example and background knowledge, the system will learn the hypothesis using ILP. When the user adds the next sentence, the system will follow steps one and two and checks with the hypotheses learned before. If there is a match, the system will label the sentence as before; otherwise, the system will try to learn a new classification rule using the current input task as a positive example and examples from other tasks as negative examples. In the next section, we present some of the experimental evaluations of the learning algorithms which we have been exploring for integration within MGL.

4.1 ILP Experiments using the learning engine Aleph

In this experiment, we check whether it is possible to use the ILP learning engine Aleph[22] for few-shot text classification. Examples and background knowledge are defined as follows:

Examples We use two positive examples for each of the "family", "work", and "sport" categories. For negative examples, we use every example of each category as the negative example for the other category.

Background Knowledge To provide Background knowledge, we initially omit stop-words using NLTK library. Then, the system generates predicates for each sentence in two steps:

1. Indicating every word of the sentences in a "contains" predicate
   (e.g. contains([registering,gym],gym)).

2. Sending a request to ConceptNet API for each word of the sentence to fetch their "related" terms and then generating their "related_to" predicate (e.g. related_to(mother,family)).

Results and discussion Using the learning engine Aleph, three rules (hypotheses) have been generated to classify the three categories.

Rule 1: category(A,family) :- contains(A,B), related_to(B,shop).
Rule 2: category(A,work) :- contains(A,B), related_to(B,letter).
Rule 3: category(A,sport) :- contains(A,B), related_to(B,exercise).

As explained above the training data for each category included 2 positive examples. We also experimented with one positive example (one-shot learning) but Aleph was unable to learn any rule. To evaluate the accuracy of the above rules on separate test data, we use a different dataset with 29 positive examples, including 15 "family", 9 "work" and 5 "sport" examples and the negative test data for each group consists of examples from other groups (i.e., 15 family examples used as negative for work, 9 work examples are used as negative for sport, and 5 sport examples are used as negative for family) and reached to the overall predictive accuracy of 64%.

4.2 MIL Experiments using the learning engine Metagol

In this experiment, we first check whether it is possible to learn a hypothesis from one positive and one negative example using MIL then explore the possibility of learning more general rules using predicate invention and recursive rules in MIL.

We employ the learning engine Metagol [14] to utilise MIL with the same Background Knowledge and examples applied in the Aleph experiment. In addition to training examples and background knowledge, Metagol also needs metarules to induce the hypothesis. We test our system with two different types of metarules. At first, we utilise a *chain* metarule to obtain the learning hypothesis. In the second experiment, we use *indent*, *chain*, and *recursive* metarules along with using constants.

Results and discussion The system induces one hypothesis from one positive and one negative example. Note that in this experiment, the positive example is a simple case where the category name is directly related to one of the words in the sentence. Consequently, one chain metarule is enough to generate the hypothesis. For instance, in category([call, mother], family), 'mother' directly connects to 'family' with only one edge of 'related_to' relation in ConceptNet. Hence, the acquired hypothesis consists of one 'related_to' predicate as follows:

category(A,B) :- contains(A,C), related_to(C,B).

This rule is similar to the rule which was learned by Aleph in the previous section when we provided at least two positive examples for each category and used variable in mode declaration but Aleph (unlike Metagol) was unable to learn from just one positive example. However, the rule above cannot cover all relevant positive examples as it only considers 'related_to' once. To demonstrate this, we provided more training examples (4 positive and 6 negative) which include cases where we need at least two levels of connections, along with using chain metarule. Consequently, the system invents a new predicate (i.e., category_1(A,B):- related_to(A,C),related_to(C,B).) within its hypothesis set as follows:

```
category(A,B) :- contains(A,C), category_1(C,B).
category_1(A,B):- related_to(A,C),related_to(C,B).
```

The new rule can capture cases such as the category name 'home' which does not directly connect to 'mother' but is connected via 'family'. However, in order to have a general transitive rule for 'related_to' we need to learn a recursive rule.

Finally, we demonstrate that Metagol can be used to learn a recursive rule to capture transitive 'related_to' relations. In this experiment, we provide two positive examples with several relatednesses in their structure. Moreover, we use constant and variable at the indent metarule and consequently, the following results including a complete recursive hypothesis are achieved:

```
category(A,B):-contains(A,C),category_1(C,B).
category_1(A,B):-related_to(A,C),category_1(C,B).
category_1(A,home):-related_to(A,home).
```

As shown above, it is notable that whatever a chosen example for one-shot learning has more relatedness in its structure, i.e. higher complexity compared to the other examples, it could achieve a complete recursive hypothesis. This, in turn, is especially beneficial for choosing the best possible shot to do one-shot learning.

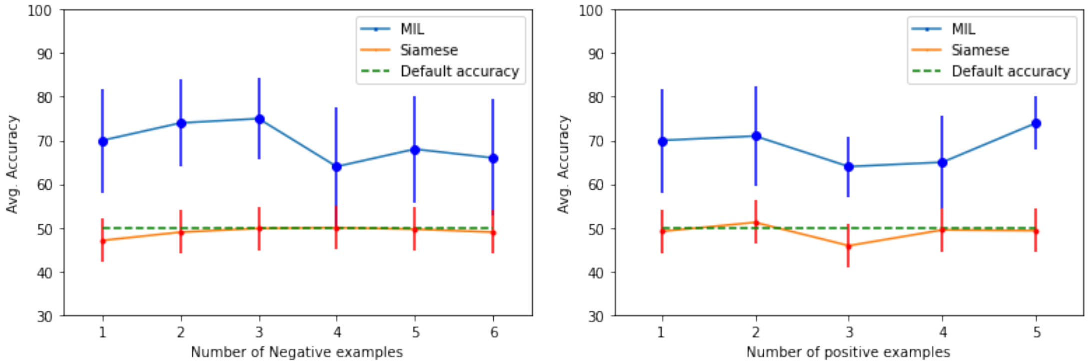
To evaluate one-shot learning, we employ the same test dataset used for the experiments with Aleph in the previous section. As evidenced by the results, Metagol could achieve around 70% accuracy, while Aleph could not learn any rule from one example.

In summary, we demonstrated that the Metagol learning engine can be used in our system to learn text classification rules from one example (one-shot learning). This feature clearly distinguishes Metagol (MIL) from Aleph (standard ILP). Moreover, we also demonstrated learning recursive rules in addition to predicate invention. However, this capability required metarules defined for the given application. In future, we will explore new implementations of MIL which can automatically generate the required metarules from background knowledge and examples.

4.3 Experiments comparing MIL with Deep Learning

In this experiment, we use Siamese Network for few-shot learning for text classification and compare the results with MIL from the previous sections. A Siamese network includes two identical sub-networks with shared weights. Each sub-network works simultaneously and compares its outputs at the end [1]. If the input of the twin sub-networks is the same, they extract similar semantic features, and the distance between their output will be less; otherwise, the distance will be more significant. The Siamese Network used in this experiment is similar to the CNN-based Siamese model described in [27] where each twin sub-networks consist of a convolutional layer, a max pooling layer and a fully connected layer. The model sets with a batch size of 50, and a learning rate of 0.00001. To achieve optimal results, the network is trained over ten epochs.

Following the approach described in [25], we evaluate the performance of the Siamese network and compare the results with our one-shot learning approach using Metagol. The training data for both systems are one positive example and one or more negative examples (examples from other categories) and the test data is the same as the one used in the previous section.



(a) Increasing negative samples

(b) Increasing positive samples

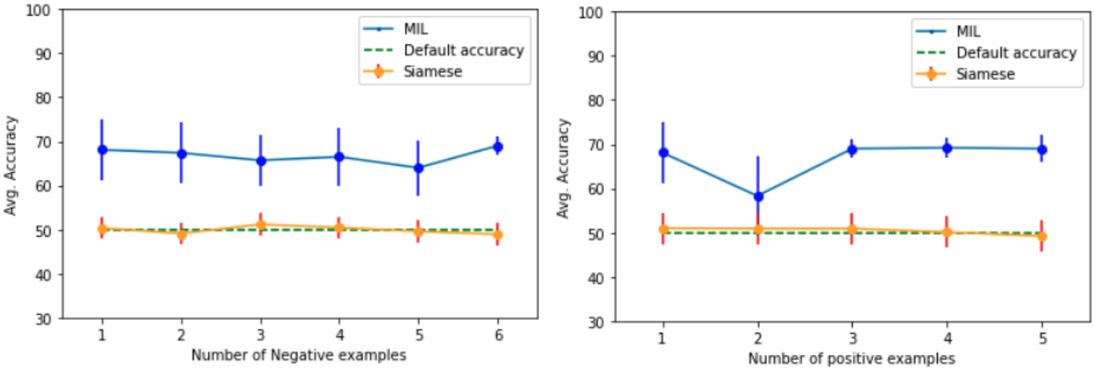
Figure 3: Comparison between the average predictive accuracy of MIL (Metagol) vs Deep Learning (Siamese Net) on Task Classification dataset. (a) learning with one positive example and one or more negative examples. (b) learning with one or more positive examples, and the same number of negatives. The training and test data have been randomly chosen from the Task Classification dataset and the test data includes 15 positive and 14 negative examples.

As shown in Figure (3a), the Siamese model started from 47% average accuracy with one positive and one negative example and stays around the default accuracy after learning from 6 examples(50%). Our system achieved an average accuracy of 70% from one positive and one negative example and maintained higher average accuracy than the Siamese model throughout. Figure (3b), depicts the average accuracy of our system and Siamese model in the case of an increasing number of positive examples. It also confirms the outperformance of our model over Siamese model.

To evaluate our model on a publicly available dataset, we use News Category Dataset [9] which contains news articles from different sources and covers various topics such as sports, technology and the environment. We randomly selected five positive examples from the environment news headlines to train our model and test the gain hypotheses on randomly selected 50 positive and 50 negative samples. As shown in Figure 4, the results suggest the advantage of our method based on MIL over the Siamese model when we have few training examples, e.g. one positive and one negative example.

It is noticeable that for training and evaluating a large dataset, we use two techniques as follows:

1. To solve the problem of requiring large amounts of background knowledge for training and testing, we use the background knowledge of each example separately.
2. To train our model on a large dataset with only one or a few examples (one-shot learning), we randomly select 10 samples and make an average from several possible one-shots to calculate the average of different types of one-shots (ex: the average of possible one positive and one negative example shots or an average of one positive and two negative examples etc).



(a) Increasing negative samples

(b) Increasing positive samples

Figure 4: Comparison between the average predictive accuracy of MIL (Metagol) vs Deep Learning (Siamese Net) on News Category dataset. (a) learning with one positive example and one or more negative examples. (b) learning with one or more positive examples, and the same number of negatives. The training and test data have been randomly chosen from News Category Dataset and the test data includes 50 positive and 50 negative examples.

5 Conclusion

Short text classification is challenging due to the freely constructed sentence structures and their limited length. This study aims to provide a novel algorithm to classify short texts called "tasks" automatically based on the user's interests and without having any prior information about user behaviours. Thus, the first research question sought to determine a reliable source for providing pertinent background information to feed the system. We utilised Conceptnet to prepare the required background information related to the task. Our model was then designed based on different types of ILP and MIL systems, including Aleph and Metagol. We also used a Siamese Network model to compare with the deep learning methods. Siamese nets are the most popular DL approaches for one or few-shot learning. We applied the Convolutional-Siamese model with one positive and one or more negative examples. According to the current results, the MIL-based approach achieved 70% accuracy for one-shot learning, while the accuracy of ILP and Deep Learning was around default accuracy (50%) in this task.

Finally, we used News Category Dataset as a publicly available dataset to evaluate our model. This led us to utilise two techniques, including Background Knowledge Splitting and the Average One Shot Learning approach to train and evaluate the large dataset. The final results validated MIL's superior performance to the Siamese network for one-shot learning from text.

Supplemental Materials
The data and code used in the experiments can be found in the following Github repository:
https://github.com/ghazalmilani/One-Shot-Learning-from-Text-ICLP2023.git

Acknowledgement
The first author would like to acknowledge EIT-Digital and Goalshaper Ltd, especially Ms Tracey Carr for supporting her PhD studies. We also thank Dany Varghese for the discussions related to this project.

References

[1] Davide Chicco (2021): *Siamese Neural Networks: An Overview*, pp. 73–94. Springer US, New York, NY, doi:10.1007/978-1-0716-0826-5_3.

[2] Andrew Cropper & Sebastijan Dumančić (2022): *Inductive logic programming at 30: a new introduction*. Journal of Artificial Intelligence Research 74, pp. 765–850, doi:10.1613/jair.1.13507.

[3] Andrew Cropper & Stephen H. Muggleton (2016): *Metagol System*. https://github.com/metagol/metagol. Available at `https://github.com/metagol/metagol`.

[4] Andrew Cropper & Sophie Tourret (2020): *Logical reduction of metarules*. Machine Learning 109, pp. 1323–1369, doi:10.1007/s10994-019-05834-x.

[5] Ji Han, Serhad Sarica, Feng Shi & Jianxi Luo (2020): *Semantic Networks for Engineering Design: A Survey*. arXiv:2012.07060.

[6] Filip Ilievski, Alessandro Oltramari, Kaixin Ma, Bin Zhang, Deborah L McGuinness & Pedro Szekely (2021): *Dimensions of commonsense knowledge*. Knowledge-Based Systems 229, p. 107347, doi:10.1016/j.knosys.2021.107347.

[7] Yann LeCun, Yoshua Bengio & Geoffrey Hinton (2015): *Deep learning*. nature 521(7553), pp. 436–444, doi:10.1038/nature14539.

[8] Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B Tenenbaum & Stephen H Muggleton (2014): *Bias reformulation for one-shot function induction*. doi:10.3233/978-1-61499-419-0-525.

[9] Rishabh Misra (2022): *News category dataset*. arXiv preprint arXiv:2209.11429, doi:10.48550/arXiv.2209.11429.

[10] Rolf Morel, Andrew Cropper & C.-H. Luke Ong (2019): *Typed Meta-interpretive Learning of Logic Programs*. In: JELIA, doi:10.1007/978-3-030-19570-0_13.

[11] Stephen Muggleton (2009): *Inverse entailment and progol*. New Generation Computing 13, pp. 245–286, doi:10.1007/BF03037227.

[12] Stephen Muggleton, Wang-Zhou Dai, Claude Sammut, Alireza Tamaddoni-Nezhad, Jing Wen & Zhi-Hua Zhou (2018): *Meta-interpretive learning from noisy images*. Machine Learning 107(7), pp. 1097–1118, doi:10.1007/s10994-018-5710-8.

[13] Stephen Muggleton & Luc de Raedt (1994): *Inductive Logic Programming: Theory and methods*. The Journal of Logic Programming 19-20, pp. 629–679, doi:10.1016/0743-1066(94)90035-3. Special Issue: Ten Years of Logic Programming.

[14] Stephen Muggleton, Dianhuan Lin, Niels Pahlavi & Alireza Tamaddoni-Nezhad (2014): *Meta-interpretive learning: Application to grammatical inference*. Machine Learning 94, doi:10.1007/s10994-013-5358-3.

[15] Stephen Muggleton, Dianhuan Lin & Alireza Tamaddoni-Nezhad (2015): *Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited*. Machine Learning 100, pp. 49–73, doi:10.1007/s10994-014-5471-y.

[16] Stephen H. Muggleton (2017): *Meta-Interpretive Learning: Achievements and Challenges (Invited Paper)*. In Stefania Costantini, Enrico Franconi, William Van Woensel, Roman Kontchakov, Fariba Sadri & Dumitru Roman, editors: Rules and Reasoning, Springer International Publishing, Cham, pp. 1–6, doi:10.1007/978-3-319-61252-2_1.

[17] Stephen H Muggleton, Ute Schmid, Christina Zeller, Alireza Tamaddoni-Nezhad & Tarek Besold (2018): *Ultra-strong machine learning: comprehensibility of programs learned with ILP*. Machine Learning 107, pp. 1119–1140, doi:10.1007/s10994-018-5707-3.

[18] S.H. Nienhuys-Cheng, R. de Wolf, J.G. Carbonell & J. Siekmann (1997): *Foundations of Inductive Logic Programming*. Lecture Notes in Artificial Intelligence, Springer, doi:10.1007/3-540-62927-0.

[19] S. Patsantzis & S. Muggleton (2022): *Meta-interpretive learning as metarule specialisation*. Machine Learning, doi:10.1007/s10994-022-06156-1.

[20] SJ Russell & Peter Norvig (2022): *Artificial Intelligence: A Modern Approach, 4th, Global ed.*

[21] Robyn Speer, Joshua Chin & Catherine Havasi (2017): *Conceptnet 5.5: An open multilingual graph of general knowledge*. In: *Proceedings of the AAAI conference on artificial intelligence*, 31, doi:10.1609/aaai.v31i1.11164.

[22] A Srinivasan (2007): *Aleph: A learning engine for proposing hypotheses*. *Software available at http://web2. comlab. ox. ac. uk/oucl/research/areas/machlearn/Aleph/aleph. pl*. Available at `https://www.cs.ox.ac.uk/activities/programinduction/Aleph/aleph.html`.

[23] Andrew Cropper Stephen Muggleton (2015): *Can predicate invention compensate for incomplete background knowledge?* In: *Thirteenth Scandinavian Conference on Artificial Intelligence: SCAI 2015*, 278, IOS Press, p. 27, doi:10.3233/978-1-61499-589-0-27.

[24] Alireza Tamaddoni-Nezhad, David Bohan, Ghazal Afroozi Milani, Alan Raybould & Stephen Muggleton (2021): *Human–Machine Scientific Discovery*, doi:10.1093/oso/9780198862536.003.0015.

[25] Dany Varghese, Roman Bauer, Daniel Baxter-Beard, Stephen Muggleton & Alireza Tamaddoni-Nezhad (2021): *Human-like rule learning from images using one-shot hypothesis derivation*. In: *International Conference on Inductive Logic Programming*, Springer, pp. 234–250, doi:10.1007/978-3-030-97454-1_17.

[26] Bin Wang, Guangtao Wang, Jing Huang, Jiaxuan You, Jure Leskovec & C-C Jay Kuo (2021): *Inductive learning on commonsense knowledge graph completion*. In: *2021 International Joint Conference on Neural Networks (IJCNN)*, IEEE, pp. 1–8, doi:10.1109/IJCNN52387.2021.9534355.

[27] Leiming Yan, Yuhui Zheng & Jie Cao (2018): *Few-Shot Learning for Short Text Classification*. *Multimedia Tools Appl.* 77(22), p. 29799–29810, doi:10.1007/s11042-018-5772-4.

On the Potential of CLIP for Compositional Logical Reasoning

Justin Brody
Franklin and Marshall College
Lancaster, PA, USA
`justin.brody@fandm.edu`

In this paper we explore the possibility of using OpenAI's CLIP to perform logically coherent grounded visual reasoning. To that end, we formalize our terms and give a geometric analysis of how embeddings in CLIP's latent space would need to be configured in order for the system to be logically coherent. Our main conclusion is that, as usually configured, CLIP cannot perform such reasoning.

Introduction

Recent work in machine learning has seen stunning results in combining generative language models with vision models. By tying vision and language, such models are capable of generating images from descriptions [9], [10] or engaging in sophisticated visual question answering [2] and other forms of *visual reasoning*. One striking feature of models trained on the latter class is the reasoning they do tends to produce new inferences via the generative mechanisms of the language model rather than through any kind of formal logical reasoning. However, logical reasoning exhibits a number of desirable properties, including the known soundness and completeness of first-order logic. These in turn constitute a particular form of systemic *compositionality* – the capacity noted in symoblic engines to systematically combine symbols to produce more complex representations [12]. Indeed, in old debates between symbolic and connectionist approaches to artificial intelligence, a frequent critique made by proponents of the symbolic approach was that connectionist systems lacked such compositionality. To a certain degree, the tremendous success of large language models and their derivatives has muted this criticism – indeed large language models are connectionist systems which easily recombine symbolic tokens to produce complex sentences (see, e.g. [13]). Further, their obvious power leads one to wonder whether the systematic compositionality found in classical AI system might be present in contemporary neural systems as well.

This in turn leads specifically to the question of whether a more logic-based kind of reasoning can occur in vision-language models. This paper presents an analytical study of this possibility in one such model, OpenAI's CLIP [7], which embeds sets of images and sets of captions in a common vector space and uses cosine similarity to measure how well each caption describes each image. We are specifically interested in the possiblity of using CLIP as a basis for grounded, logical reasoning. By the first, I simply mean that reasoning about an image should in some way be connected to the image (*grounded* in the image). The joint-embedding mechanism of CLIP, along with cosine similiarity, provides precisely such a mechanism. Logical reasoning refers to a way of reasoning about the image that is based in symbolic logic. As noted above, this differs from most of what is called *visual reasoning* in the literature, where large language models (rather than logic) are usually used to generate implications. In the taxonomy Daniel Kahnemahn laid out in *Thinking Fast and Slow*[5], the logical visual reasoning we are interested

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 80–86, doi:10.4204/EPTCS.385.10

in would be an instance of slow, System 2, reasoning, whereas the usual literature is (arguably) concerned with System 1. As such, it loses many of the desirable characteristics (e.g. soundness and completeness) of logical reasoning.

Our final analysis will show that, as usually configured, CLIP **cannot** serve as a basis for grounded logical reasoning about images. Our final contributions are thus:

1. We give a formal analysis of grounded logical reasoning in systems like CLIP; and
2. We show that such systems ultimately cannot support such reasoning.

We by no means see this negative result as the end of the story. Indeed, our formal analysis will point strongly to CLIP's use of cosine similarity as imposing severe geometric contraints on what a grounded logical reasoning system based on CLIP would look like. This points to the desirability of analyzing whether other ways of measuring similarity between a caption and an image might prove better suited to building such a system.

CLIP

We preface our analysis with a summary of the operation of CLIP – full details are in [7]. Given a set of images $\{I_1,\ldots,I_m\}$ and a set of potential captions $\{C_1,\ldots,C_n\}$ as input, CLIP will process the two sets as follows:

1. Each image is passed through a fixed visual feature detector (often a vision-transformer) and the detected features are then passed into a learned embedding network which maps each I_i to a vector $\mathbb{I}_i$ in some finite dimensional vector space $\mathbf{L} = \mathbf{R}^d$. Thus CLIP implicitly defines a function f which maps images to vectors in $\mathbf{L}$.
2. Similarly, each caption C_i is passed through a language model (e.g GPT2 [8]) to obtain vector embedding of the caption, which is then also passed into a an embedding network to produce a vector $\mathbb{C}_i \in \mathbf{L}$. Thus CLIP implicitly defines a function g which maps captions to $\mathbf{L}$.
3. For each image I_i, the probability distribution on captions describing I_i is determined by taking a softmax of the cosine similiarities between each $\mathbb{C}_j$ and $\mathbb{I}_i$. That is,

$$p = \text{softmax}\left(\begin{bmatrix} \frac{\mathbb{I}_i \cdot \mathbb{C}_1}{\|i_i\|\|c_1\|} \\ \vdots \\ \frac{\mathbb{I}_i \cdot \mathbb{C}_n}{\|i_i\|\|c_n\|} \end{bmatrix}\right)$$

   where $\text{softmax}(\begin{bmatrix} x_1 & \ldots & x_n \end{bmatrix}^T)$ is defined as $\frac{1}{\sum_{i=1}^{n} e^{x_i}} \begin{bmatrix} e^{x_1} & \ldots & e^{x_n} \end{bmatrix}^T$

We note that CLIP has been shown to be a very effective model for recognizing a wide variety of classes [7], and indeed its flexibility seems to offer tremendous potential for using CLIP as a basis for visual reasoning in the wild. This promise largely underwrites the interest in this paper in studying CLIP as a basis for grounded logical reasoning.

Related Work

The genesis of this paper is a desire to combine the seemingly grounded object recognition capabilities of CLIP with the power of traditional logical reasoning. This places the current work within the body of literature studying visual reasoning using language model; some prominent examples include Flamingo [1], Visual Comet [6] and more generally the large contemporary literature on visual question answering and visual commonsense reasoning. While our analysis takes place in the context of CLIP, similar analyses can be done for other vision-language systems.

Our particular emphasis here is on compositionality in such systems, and in particular in our theoretical analysis we will focus on some geometric considerations on the latent space of CLIP. This ties the current work to the literature on compositionality in symbolic systems [12]. The geometric analysis finds some precedent in the work of Peter Gärdenfors [3].

There is other work specifically dealing with compositionality in language models and language-vision models. For example, [4] shows that the BERT-like language models often fail at logical reasoning, while [11] raises the question of whether VQA models are truly compositional.

Geometric Analysis

The question in this paper is whether or not CLIP (or a variant thereof) can support grounded logical reasoning. As noted, the explicit tying of the images and the captions which desribe them provides some kind of grounding. Indeed, we can view the behavior of CLIP as providing a kind of logical interpretation function: for any fixed image i and description d, we can think of d being true in i exactly when CLIP gives higher probability to d than to $\neg d$. In classical logic, such an interpretation then extends to a notion of truth in a model for arbitrarily complex combinations of atomic descriptions. Moreover, this notion of truth is exactly reflected in the inference rules of classical logic – this is precisely the soundness and completeness of first-order (or propositional) logic. That is, starting with a consistent set of axioms and applying inference rules not only yields logical consequences of the axioms, but every such consequence can be attained this way.

We would like to examine whether the grounding obtained by viewing CLIP as providing an interpretation function can be extended in a similar vein. To address this question, we first sharpen our definitions and introduce some notation. Suppose we are working with some fixed set of image descriptions $\mathscr{D}$, and further that $\mathscr{D}$ is the closure under some set of logical operations of some generating set of atomic descriptions $\mathscr{A}$. For example, $\mathscr{A}$ might be a set of natural language descriptions of images as belonging to a set of basic categories, containing strings like "An image of a cat" and "An image of a dog". Then closing under logical operators would add descriptions like "An image of a cat or a dog" and "Not an image of a cat" (we will formalize these notions below). For any $d \in \mathscr{D}$, let us say that d is true of i when d accurately describes i, and write this as $i \models_C d$ [1] . We will write $i \vdash_C d$ when CLIP assigns higher probability to d than to $\neg d$ given the image set $\{i\}$ and the caption set $\{d, \neg d\}$. In analogy with the soundness and completeness of propositional calculus, our hope is that $i \models_C d$ exactly when $i \vdash_C d$.

We begin our discussion by defining our basic syntax and semantics. Throughout, we will subscript standard logical notation with $_C$ (e.g. $\vee_C$) to indicate that we are working with related but different notions. Since CLIP uses natural language descriptions, this is slightly more complicated than the siutation in propostional calculus, where atomic propositions are combined in a natural way. Let $\mathscr{N}$ denote the set of all English language descriptions. We will fix functions $\phi_\vee : \mathscr{N} \times \mathscr{N} \to \mathscr{N}, \phi_\wedge : \mathscr{N} \times \mathscr{N} \to \mathscr{N}$ and $\phi_\neg : \mathscr{N} \to \mathscr{N}$ which determine how logical operators combine descriptions. For example, we might have $\phi_\vee$("An image of a cat", "An image of a dog") $=$ "An image of either a cat or a dog". Given these

---

[1] We recursively define $i \models_C d$ for any $d \in \mathscr{D}$ in Definition 3 below.

functions and given $d, e \in \mathscr{A}$, we will write $d \vee_C e$ for $\phi_\vee(d,e)$, and similarly for $\wedge_C$ and $\neg_C$. The syntax and semantics of these operations are defined recursively in the expected way.

Definition 1. Let $\mathscr{A}$ be any set of descriptions, which we take to be atomic. Then we define $\mathscr{D}_\mathscr{A}$, the set of *well-formed sentences generated by* $\mathscr{A}$ as the set of strings of the form:

- d for $d \in \mathscr{A}$; OR
- $\neg_C(d)$ for well-formed $d \in \mathscr{D}$; OR
- $(d \vee_C e)$ for well-formed $d, e \in \mathscr{D}$; OR
- $(d \wedge_C e)$ for well-formed $d, e \in \mathscr{D}$

When context makes them unnecessary, we will omit the parentheses in well-formed sentences (adopting the usual precedence conventions for logical operators).

We now turn to the models under consideration; these are meant to have the same basic structure as CLIP and use cosine similarity to measure the closeness of embeddings.

Definition 2. Let $\mathscr{I}$ denote any set of images and let $\mathscr{D}$ be as in convention 5. Then a *CLIP-like* model is a pair of functions (f,g) where for some Euclidean vector space $\mathbf{L}$, we have that $f : \mathscr{I} \to \mathbf{L}$ and $g : \mathscr{D} \to \mathbf{L}$. That is, f and g embed images and descriptions into a shared latent space.

If C is CLIP-like, then for $i \in \mathscr{I}, d \in \mathscr{D}$ we define $\alpha(i,d)$ to be the cosine similarity between the respective embeddings of i and d.

$$\alpha_C(i,d) := \frac{f(i) \cdot g(d)}{||f(i)||||g(d)||}$$

We will use this notion to recursively define the relation $i \models_C d$.

Definition 3. Let $C = (f,g)$ be CLIP-like. For any image i and well-formed $d, e \in \mathscr{D}_\mathscr{A}$:

- If $d \in \mathscr{A}$, then $i \models_C d$ if and only if C assigns higher probability to d then to $\neg d$ when C uses input image $\{i\}$ and caption set $\{d, \neg d\}$. That is, $i \models_C d$ if and only if $\alpha_C(i,d) > \alpha_C(i, \neg_C d)$.
- $i \models_C \neg_C d$ if and only if it is not the case that $i \models_C d$.
- $i \models_C d \vee_C e$ if and only if $i \models_C d$ or $i \models_C e$.
- $i \models_C d \wedge_C e$ if and only if $i \models_C d$ and $i \models_C e$

We want to introduce some modest constraints on the types of images and their potential descriptions which we will consider.

Definition 4. Let $\mathscr{I}$ be a set of images and let $\mathscr{A}$ be a set of strings which represents atomic descriptions of elements of $\mathscr{I}$, as above. Let C be CLIP-like.

- We say that $\mathscr{I}$ *is describable by* $\mathscr{A}$ if for $i \in \mathscr{I}$, there is some $d \in \mathscr{A}$ such that $i \models_C d$ and conversely for every $d \in A$ there is some $i \in \mathscr{I}$ such that $i \models_C d$.
- If $\mathscr{I}$ is describable by $\mathscr{A}$ we say that $(\mathscr{I}, \mathscr{A})$ is *separable* if:
  - For every $i \in \mathscr{I}$, there is some $e \in \mathscr{A}$ such that $i \models_C \neg_C e$.
  - For every $e \in \mathscr{A}$, there is some $i \in \mathscr{I}$ such that $i \models_C \neg_C e$.

  Separability prevents a situation where all descriptions are true of any one image or any one description is true of all images.

Convention 5. *For the remainder of this paper, we fix a set of images* $\mathscr{I}$ *and a set of strings* $\mathscr{A}$ *with* $\mathscr{I}$ *describable by* $\mathscr{A}$ *and* $(\mathscr{I}, \mathscr{A})$ *separable. Further, we shall abbreviate* $\mathscr{D}_\mathscr{A}$ *simply as* $\mathscr{D}$.

In our analysis, the following observations will be used repeatedly

Remark 6. Let C be CLIP-like with latent space $\mathbf{L}$. For any two points $a,b \in \mathbf{L}$, let $\chi(x,y) := \frac{a \cdot b}{\|a\|\|b\|}$ denote the cosine similartiy between x and y

1. $\chi(a,b)$ is not defined for $a = 0$ or $b = 0$.
2. The cosine similarity of a,b is maximal when the angle between a,b is 0 and $\chi(a,b) = 1$; it is minimal when the angle is $\frac{\pi}{2}$ and $\chi(a,b) = -1$
3. The set of all $c \in \mathbf{L}$ such $\chi(a,c) = 1$ forms a ray in $\mathbf{L}$ as does the set of all $c \in \mathbf{L}$ with $\chi(a,c) = -1$. Further, the latter ray is the reflection of the former ray over the origin. We refer to the latter ray as the *anti-ray* of the former ray. Since χ is not defined at the origin, any ray is disjoint from its anti-ray.

We now define C-completeness for a CLIP-like structure.

Definition 7. If C is CLIP-like, we say that

- *C respects basic descriptions* if for every $i \in \mathscr{I}, d \in \mathscr{A}$, if $i \models_C P$ then $\alpha_C(i,d) = 1$.
- *C respects negation* if for $i \in \mathscr{I}, d \in \mathscr{A}$, if $i \models_C \neg_C d$ then $\alpha_C(i,d) = -1$
- *C respects disjunction* if for $i \in \mathscr{I}, d, e \in \mathscr{D}$, if $i \models_C d \vee_C e$, then $\alpha_C(i, d \vee_C e) = 1$.
- *C respects conjunction* if for $i \in \mathscr{I}, d, e \in \mathscr{D}$, if $i \models_C d \wedge_C e$, then $\alpha_C(p, d \wedge_C e) = 1$.
- C is *C-complete* if it respects descriptions, negation, disjunction and conjunction.

A simple induction (on sentence length) shows that if C is C-complete, then for **any** description $d \in \mathscr{D}$ and image $i \in \mathscr{I}$, if $i \models_C d$ then $\alpha_C(i,d) = 1$ and, *a fortiori*, $i \vdash_C d$. The idea here is that the relation $i \vdash_C d$ represents some kind of notion of d being inferred by C. In first-order logic, completeness means that everything which can be inferred IS inferred. We would like something similar for $\vdash_C$ derivations – in particular the ability to derive all true descriptions of an image.

What we will see is that this desire places extreme geometric constraints on the embeddings $f(i), g(d)$. In fact, assuming that C-completeness holds, we will show that all descriptions which are true of *any* image will necessarily live on a single ray, while all descriptions which are false of any image will live on that ray's anti-ray. This will imply that all descriptions are on both the ray and it's anti-ray, which is a contradiction since the two are disjoint.

Proposition 8. *If $C = (f,g)$ is CLIP-like then C is not C-complete.*

We will prove this via a sequence of lemmas.

Lemma 9. *If $C = (f,g)$ is C-complete, then there exists a ray $\mathbf{R}$ in $\mathbf{L}$ such that $f(\mathscr{I}) \subseteq \mathbf{R}$ and $g(\mathscr{D}) \subseteq \mathbf{R}$.*

Proof. Fix $i, j \in \mathscr{I}$ and $d, e \in \mathscr{D}$ such that $i \models d, j \models e$. We will show that $f(i), f(j), g(d), g(e)$ are all on the same ray. Since C respects basic descriptions, we have

$$\alpha_C(i,d) = 1 \tag{1}$$

Since $i \models_C d$, we have $i \models_C d \vee_C e$ as well. Since C respects disjunctions, we have

$$\alpha_C(i, d \vee_C e) = 1 \tag{2}$$

Similarly, we have

$$\alpha_C(j,e) = 1 \tag{3}$$

$$\alpha_C(j,d \vee e) = 1 \tag{4}$$

By (3) and (4), $f(j)$ is on the same ray as $g(d \vee_C e)$, as is $g(e)$. By (1) and (2), these are on the same ray as $f(i)$ and $g(d)$. $\square$

Lemma 10. *If $C = (f,g)$ respects descriptions, disjunctions and negations, then there exists a ray* $\mathbf{A}$ *in* $\mathbf{L}$ *such that for every $i \in \mathscr{I}, d \in \mathscr{D}$, if $i \models \neg d$ then $g(d) \in \mathbf{A}$. Moreover,* $\mathbf{A}$ *is the anti-ray of the ray* $\mathbf{R}$ *guaranteed by Lemmma 9.*

Proof. By Lemma 9, $f(i) \in \mathbf{R}$. Since C respects negations, $\alpha_C(p,d) = -1$; thus $g(d) \in \mathbf{A}$. $\square$

Fix $\mathbf{R}$ as in Lemma 9 and $\mathbf{A}$ as in Lemma 10. We note that Lemma 10 showed that if d is false for *any* $i \in \mathscr{I}$, then $g(d) \in \mathbf{A}$. We also know from Lemma 9 that if d is true *any* $i \in I$, then $g(d) \in \mathbf{R}$.

Corollary 11. *For $C = (f,g)$ as above and any $d \in \mathscr{D}$,*

1. *If there is any $i \in \mathscr{I}$ for which $i \models_C \neg_C d$, then $g(d) \in \mathbf{A}$.*
2. *If there exist $p,q \in \mathscr{I}$ such that $p \models_C d, q \models_C \neg_C d$, then $g(d) \in \mathbf{R} \cap \mathbf{A}$*

Finally we prove our main result.

Proof of Proposition 8. Suppose, by way of contradiction, that $C = (f,g)$ is CLIP-like and C-complete. Fix $d \in \mathscr{A}$. Since $\mathscr{I}$ is describable by $\mathscr{A}$, there exists some $i \in \mathscr{I}$ such that $i \models d$. By Lemma 9, $f(i) \in \mathbf{R}$ and $g(d) \in \mathbf{R}$. Using separability and describability, choose $j \in \mathscr{I}$ and $e \in \mathscr{A}$ such that $j \models \neg d \wedge e$. Since $j \models \neg d$, we have $g(d) \in \mathscr{A}$ as well by Corollary 11. However, we have that $\alpha_C(i,d) = 1$ so that $f(i) \in \mathbf{A}$ as well. Thus $\{f(i), g(d)\} \subseteq \mathbf{R} \cap \mathbf{A}$. But $\mathbf{R} \cap \mathbf{A} = \emptyset$, contradicting that f,g are maps into $\mathbf{L}$. $\square$

Conclusions and Future Work

Our theoretical analysis shows no matter how accurately a CLIP-like model detects basic categories, this cannot extend to arbitrary boolean combinations. This is by no means a fatal blow to the use of CLIP in compostional reasoning; indeed two immediate possibilities present themselves. The most obvious is simply to use CLIP for basic category recognition and deploy an external system to handle composition of categories. For example, to determine whether an image is described by $P \wedge \neg(Q \vee R)$ we can simply run CLIP to recognize each of the categories P, Q, R and combine the results using something like fuzzy logic. In a sense, this is a validation of using hybrid neurosymbolic models rather than trying to reason within a vision-language model.

That said, there are reasons it would be desirable to work in a visual reasoning system in which the latent space was organized in a logically coherent manner as discussed in the paper. While this cannot be done for a CLIP-like system using cosine similarity, it is still open whether a similar system organized according to Euclidean distance or some other metric might work. This will be explored in future work.

We also note that the results in this paper represent a kind of limiting case – the question of logical coherence basically amounts to asking what the geometry of a perfectly coherent system would look like. We have not addressed what the possibilities would be for a system where the requirement of perfection was relaxed, for example by having $i \models_C d$ correspong to $\alpha_C(i,d) \geq 1 - \varepsilon$ for some small $\varepsilon > 0$. This too will be explored in future work.

References

[1] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, Roman Ring, Eliza Rutherford, Serkan Cabi, Tengda Han, Zhitao Gong, Sina Samangooei, Marianne Monteiro, Jacob L. Menick, Sebastian Borgeaud, Andy Brock, Aida Nematzadeh, Sahand Sharifzadeh, Mikolaj Binkowski, Ricardo Barreira, Oriol Vinyals, Andrew Zisserman & Karén Simonyan (2022): *Flamingo: a Visual Language Model for Few-Shot Learning*. Available at `http://papers.nips.cc/paper_files/paper/2022/hash/960a172bc7fbf0177ccccbb411a7d800-Abstract-Conference.html`.

[2] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C. Lawrence Zitnick & Devi Parikh (2015): *VQA: Visual Question Answering*. In: *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, IEEE Computer Society, pp. 2425–2433, doi:`10.1109/ICCV.2015.279`.

[3] Peter Gardenfors (2004): *Conceptual spaces: The geometry of thought*. MIT press, doi:`10.7551/mitpress/2076.001.0001`.

[4] Chadi Helwe, Chloé Clavel & Fabian M. Suchanek (2021): *Reasoning with Transformer-based Models: Deep Learning, but Shallow Reasoning*. In Danqi Chen, Jonathan Berant, Andrew McCallum & Sameer Singh, editors: *3rd Conference on Automated Knowledge Base Construction, AKBC 2021, Virtual, October 4-8, 2021*, doi:`10.24432/C5W300`.

[5] Daniel Kahneman (2011): *Thinking, fast and slow*. Macmillan.

[6] Jae Sung Park, Chandra Bhagavatula, Roozbeh Mottaghi, Ali Farhadi & Yejin Choi (2020): *VisualCOMET: Reasoning About the Dynamic Context of a Still Image*. In Andrea Vedaldi, Horst Bischof, Thomas Brox & Jan-Michael Frahm, editors: *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part V*, *Lecture Notes in Computer Science* 12350, Springer, pp. 508–524, doi:`10.1007/978-3-030-58558-7_30`.

[7] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger & Ilya Sutskever (2021): *Learning Transferable Visual Models From Natural Language Supervision*. In Marina Meila & Tong Zhang, editors: *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, *Proceedings of Machine Learning Research* 139, PMLR, pp. 8748–8763. Available at `http://proceedings.mlr.press/v139/radford21a.html`.

[8] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever et al. (2019): *Language models are unsupervised multitask learners*. *OpenAI blog* 1(8), p. 9.

[9] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu & Mark Chen (2022): *Hierarchical Text-Conditional Image Generation with CLIP Latents*. *CoRR* abs/2204.06125, doi:`10.48550/arXiv.2204.06125`. arXiv:`2204.06125`.

[10] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser & Björn Ommer (2022): *High-Resolution Image Synthesis with Latent Diffusion Models*, pp. 10674–10685. doi:`10.1109/CVPR52688.2022.01042`.

[11] Sanjay Subramanian, Sameer Singh & Matt Gardner (2019): *Analyzing Compositionality in Visual Question Answering*. Available at `https://vigilworkshop.github.io/static/papers/43.pdf`.

[12] Zoltán Gendler Szabó (2022): *Compositionality*. In Edward N. Zalta & Uri Nodelman, editors: *The Stanford Encyclopedia of Philosophy*, Fall 2022 edition, Metaphysics Research Lab, Stanford University.

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser & Illia Polosukhin (2017): *Attention is All you Need*, pp. 5998–6008. Available at `https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html`.

Bottom-Up Stratified Probabilistic Logic Programming with Fusemate

Peter Baumgartner
Data61/CSIRO
Canberra, Australia
`Peter.Baumgartner@data61.csiro.au`

Elena Tartaglia
Data61/CSIRO
Melbourne, Australia
`Elena.Tartaglia@data61.csiro.au`

This paper introduces the Fusemate probabilistic logic programming system. Fusemate's inference engine comprises a grounding component and a variable elimination method for probabilistic inference. Fusemate differs from most other systems by grounding the program in a bottom-up way instead of the common top-down way. While bottom-up grounding is attractive for a number of reasons, e.g., for dynamically creating distributions of varying support sizes, it makes it harder to control the amount of ground clauses generated. We address this problem by interleaving grounding with a query-guided relevance test which prunes rules whose bodies are inconsistent with the query. We present our method in detail and demonstrate it with examples that involve "time", such as (hidden) Markov models. Our experiments demonstrate competitive or better performance compared to a state-of-the art probabilistic logic programming system, in particular for high branching problems.

1 Introduction

Probabilistic Logic Programming (PLP) combines logic programming with probability theory. Most PLP systems implement the *distribution semantics* (DS) [13]. The DS was introduced for ground definite programs whose facts are annotated with probabilities. More expressive languages require additional concepts to equip their programs with a DS. ProbLog [8], for instance, features probability annotations of rule heads, not just facts, disjunctive heads, for expressing distributions, and an expressive form of default negation. These are dealt with by a combination of program transformation, and defining the DS via two-valued well-founded models of the grounded transformed program (if it exists) [7]. After grounding, a probabilistic inference engine solves a given inference problem, e.g., the probability of a query being satisfied in the models of the program. Many PLP systems implement their grounding component (or monolithic reasoner) in a top-down way.

In this paper, we investigate the potential of bottom-up grounding as an alternative to the more common top-down grounding. Our main motivation is efficient reasoning in applications with a high branching rate, for approximating continuous distributions, in a timed setting (Markov models). Our main results are a novel method for bottom-up grounding, its implementation in our PLP system Fusemate, and the favourable experimental results we obtained with it. A main challenge with bottom-up grounding is controlling the amount of ground rules generated. Our method addresses this problem by pruning redundant rules based on a certain inconsistency test. The test is applied dynamically, at each step of the grounding process along a certain program stratification ordering. Our method supports an expressive input language with default negation, where default negation is eliminated on-the-fly at each step.

In the main part of the paper, we describe the method in detail and prove its correctness. The rest of this introduction has more motivation, overviews the main ideas, and discusses related work.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 87–100, doi:10.4204/EPTCS.385.11

Bottom-up grounding. We propose a method for grounding the program in a bottom-up way as an alternative to the top-down approach described above. Our grounding is defined for a certain form of program stratification called *SBTP*, defined in Section 2 below. The result ground program is obtained via fixpoint computation along this stratification order. In each step of the fixpoint computation the program rules of the proper stratification level are grounded thereby also eliminating occurrences of default negation. This is advantageous for on-the-fly removal of redundant ground rules, which can be informed with the result program lower than the stratum of the current step. See below for details.

The default negation operator is expressive and allows for negation over (implicitly) existentially quantified conjunctions of atoms. As an example consider this program (Fusemate uses ProbLog syntax):

```
0.5 :: q(0).  0.5 :: q(1).  0.5 :: q(2).  0.5 :: p(T) :- q(T), \+ (q(S), S < T).
```

Its grounding has the same three facts and the rule above replaced by the following three ground rules:

```
0.5::p(0) :- q(0).  0.5::p(1) :- q(1), -q(0).  0.5::p(2) :- q(2), -q(0), -q(1).
```

These rules explain all ways for `p`-literals to become true in the least Herbrand model extending a given choice of the probabilistic facts, where $-a$ is true if the fact a is false. They are computed bottom-up with a stratification that puts `q`-atoms before `p`-atoms. With this approach we obtain an extension of the DS that is compatible with, e.g., the transformation-based approach of ProbLog mentioned above.

Motivation. We are interested in exploring the potential of bottom-up grounding for practical applications. An example are *dynamic distributions*, i.e., distributions whose domain is dynamically altered by the program. Dynamic distributions are not supported by most PLP systems (exceptions are, e.g., CP-logic [18], and the approach by Gutmann et al. [9]). Fusemate has a construct $t \sim l$ where t is a term for a random variable and $l = [v_1, \ldots, v_n]$ is a Prolog list of values to draw from, uniformly by default. The list can be constructed by the program and draw cases can be queried as equations $t = v_i$. The following program models iterated drawing balls from an urn without replacement (an annotation $a \,@\, T$ stands for a dedicated "time" argument T considered part of atom a's arguments; list append ++ and difference – are built-in infix operators):

```
urn([r(1), r(2), g(1)]) @ 0.          %% Initially two red and one green distinguishable balls
draw ~ Balls @ T :- urn(Balls) @ T, Balls \= [].     %% Draw  a ball uniformly if urn is not empty
urn(Balls -- [B]) @ T+1 :- urn(Balls) @ T, draw = B @ T.    %% Drawing a ball removes it from urn
some(red) @ T :- draw=r(_) @ T.                              %% Abstract from ball id, color only
some(green) @ T :- draw=g(_) @ T.
```

To give an idea of the overall system, Fusemate can solve queries like the following:

```
?- some(green) @ 0. % 0.333333
?- some(green) @ 1 | some(red) @ 0. % 0.5 conditional query
?- some(C1) @ 1, some(C2) @ 2 | some(red) @ 0. % Non-ground conditional query, two solutions:
% 0.5 :: [C1 = red, C2 = green], 0.5 :: [C1 = green, C2 = red]
```

As in the example above, we are particularly interested in applications that involve "time", such as hidden Markov models and their enrichments with rules for domain constraints. Such applications are within the scope of many PLP systems, see [17, 14, e.g.]. In Section 5 below we report on experimental results on Markov models which demonstrate the viability of our bottom-up approach.[1]

---

[1] Bottom-up grounding also makes it easy to catch positive loops as it does not need mechanisms like tabling (in the presence

Query-guided grounding and inconsistency pruning. The advantages of bottom-up grounding are offset by some drawbacks. A major drawback is its inherent lack of goal-orientedness. Compared to the satisfiability problem for (stratified) non-probabilistic programs, the problem is amplified in a probabilistic setting as exponential branching into model candidates is unavoidable. To address this problem we developed a technique for *query-guided* bottom-up grounding. It has two components which reinforce each other: *goal regression* and *inconsistency pruning*. They rest on a fixed interpretation of equations as *right-unique relations*, which is also taken advantage of in the inference component.

To explain, let a query be a set of ground literals. Goal regression makes the query more constrained by adding literals to it. To ensure models are preserved, only literals that are included in *every* model of the query are added. This is done iteratively along the stratification order and using the grounded rules so far at each step. Inconsistency pruning prevents, at each step in the iteration, applying rules for grounding that cannot contribute to a model of the query. As an example consider this Hidden Markov Model:

```
1 state ~ [[rainy, 0.6], [sunny, 0.4]] @ 0.  %% T = 0, non-uniform distribution given
2 obs ~ [3..30] @ 0 :- state=rainy @ 0. %% Observation is accumulated rainfall over time in mm
3 obs ~ [0..5] @ 0 :- state=sunny @ 0.
4 state ~ [[rainy, 0.7], [sunny, 0.3]] @ T+1 :- state=rainy @ T.  %% T -> T+1
5 state ~ [[rainy, 0.4], [sunny, 0.6]] @ T+1 :- state=sunny @ T.
6 obs ~ [R+3..R+30] @ T :- state=rainy @ T, T > 0, obs=R @ T-1. %% Increase observation at T-1
7 obs ~ [R..R+5] @ T :- state=sunny @ T, T > 0, obs=R @ T-1.
```

Take the query `?- obs=0 @ 0, obs=4 @ 1, obs=20 @ 2, obs=24 @ 3`. The task is to compute its success probability. Note that an increase of `obs`erved rain by 0 at time `0` can be proven *only* through the subgoal `state=sunny@0` as per the rule on line 3. Goal regression detect this and extends the query with `state=sunny@0`. Then, inconsistency pruning applies and rejects the rule on line 2 because its body literal `state=rainy@0` is *inconsistent* with the (extended) query. This inconsistency follows from our fixed semantics of equations for representing functions, i.e., right-unique relations. With that, `state=sunny@0` and `state=rainy@0` are inconsistent as `sunny` $\neq$ `rainy` (by unique name assumption). The ground instance `state ~ [[rainy, 0.7], [sunny, 0.3]] @ 1 :- state=rainy @ 0` is rejected for the same reason. The justification for inconsistency pruning is that no model of the query can satisfy the body of a rule that is inconsistent with the query. Rejected rules can only "fire" in non-models of the query. As these interpretations receive probability 0 anyway, such rules can just as well be ignored.

Goal regression and inconsistency happen during each step of the bottom-up grounding process. For instance, the query is extended later with `state=rainy@2` and all ground instances of the rules with a body literal `state=sunny@2` are rejected. The general technique is detailed in Section 3.

Related work. Too many PLP systems have been developed to discuss them all in detail. De Raedt and Kimmig [7], and Riguzzi et al. [12] provide overviews. Fierens et al. [8] explains ProbLog in depth and compares it with related systems. Systems with built-in top-down grounding are based on variants of SLD-resolution (e.g., ProbLog, PITA [11]) and construct (ground-level) stratified programs by design. The most closely related work comes from systems with built-in bottom-up grounding and that optimize the grounding in some way. This excludes systems that assume an external grounding component but otherwise have a related execution model, e.g., CP-Logic [18], which also supports dynamic distributions. Vlasselaer et al. [19] describe a fixpoint approach for incrementally grounding stratified programs

---

of symmetry and transitivity axioms, for instance). Fusemate deals with cyclic programs by means of certain query rewriting techniques. The details are beyond the scope of this paper.

with possibly positive cycles. Their grounding works incrementally and supports any-time reasoning for varying queries. Grounding can be controlled heuristically, which guarantees lower success probabilities bounds, or, in certain cases, exact probabilities, e.g., for filtering queries in Hidden Markov Models. Several authors have considered (variants of) magic set transformations for controlling the grounding process. The idea is to avoid bottom-up inferences with program rules that are impossibly relevant for proving the query at hand. Relevance is determined in a regression process from the query towards the facts using the program rules. Gutmann et al. [9] introduced magic sets for probabilistic logic programs. Their approach supports dynamic distributions but not default negation. Tsamoura et al. [16] extend the work by Vlasselaer et al. [19] by adding an incremental magic set transformation during grounding. While the goal of magic sets and our inconsistency pruning are similar, their realizations are different. Our inconsistency pruning also regresses the query, however, by adding literals in the *intersection* of all rule applications instead of taking unions. Furthermore, pruning is based on *inconsistency* of grounded bodies with the query, which is not the case in magic sets. The HMM example above, for instance, is not covered by magic sets. As future work, it seems promising to combine these orthogonal techniques. The first author introduced SBTP in an earlier *non-probabilistic* Fusemate system [2, 4] and combined it with description logic reasoning and the event calculus [3]. In this paper we adapt SBTP to a probabilistic setting and integrate query guidance into it. The unguided grounding in these earlier `Fusemate` systems is unsuitable for all but the smallest problems. Skarlatidis et al. [15] use ProbLog for event calculus reasoning, which is an interesting application for us as well.

2 Stratified Logic Programs

Assume as given a first-order logic signature comprised of function and predicate symbols of fixed arities. Assume a countable set of variables. A *term* is either a variable or an expression of the form $f(t_1,\ldots,t_n)$ where f is a function symbol and $t_1,\ldots,t_n$ are terms. Atoms are of the form $p(t_1,\ldots,t_n)$ where p is a predicate symbol. A *ground* term or atom does not contain any variable. We also use vector notation $\vec{t}$ to denote a list $t_1,\ldots,t_n$ of terms (or other objects), for some $n \geq 0$. We distinguished between ordinary (free) symbols and "built-in" symbols. The latter include the integer constants $\mathbb{Z}$ and infix operators like $<$ and $+$. A term or atom with a built-in symbol is called an *interpreted* term or atom, otherwise it is *ordinary*. We require that all terms are well-sorted, so that interpreted ground terms can effectively be evaluated to a unique irreducible term. Interpreted ground atoms must be valuable to a Boolean $\top$ or $\bot$. We assume built-in function symbols for forming lists as in Prolog.We require that every ordinary atom has a dedicated $\mathbb{Z}$-sorted argument, say, the first one, for a *time term* tt. When we write $p(\vec{t})\,\texttt{@}\,tt$ we mean $p(tt,\vec{t})$ instead. If a is an atom we write $a\,\texttt{@}\,tt$ to indicate it has time term tt, i.e., a is of the form $p(tt,\vec{t})$. We assume an ordinary ternary infix predicate symbol = and write =-atoms as $s\,\texttt{=}\,t\,\texttt{@}\,tt$. An =-atom is *admissible* if its left hand side term s is an ordinary functional term $f(\vec{t})$. We only work with admissible =-atoms. An admissible =-atom $f(\vec{t})\,\texttt{=}\,t\,\texttt{@}\,tt$ could be taken as an ordinary atom $f(\vec{t},t)\,\texttt{@}\,tt$ with f as a predicate symbol. The infix notation is warranted as equations have built-in semantics as right-unique relations (see Section 3). Let e be a term or atom. By $var(e)$ we denote the set of variables occurring in e. Let σ be a substitution. We write $e\sigma$ for the term or atom resulting from applying σ to e. The domain of σ is $dom(\sigma)$. A substitution γ is a *grounding substitution for a set of variables X* iff $dom(\gamma) = X$ and $x\gamma$ is ground for every $x \in dom(\sigma)$. In the following, the letters x,y,z stand for variables, p,q,r for predicate symbols, and s,t for terms, possibly indexed. A *(Fusemate) rule* is an implication of the form

$$H \;\texttt{:-}\; b_1,\ldots,b_k,\backslash\texttt{+}\vec{c}_1,\ldots,\backslash\texttt{+}\vec{c}_n \quad . \tag{1}$$

where $b_1,\ldots,b_k$ is a sequence of atoms and each $\vec{c}_j$ is a sequence of atoms, for some $k,n \geq 0$. The part to the right of `:-` is called the *body* B. Every b_i is called a *positive (body) literal* and every `\+`$\vec{c}_j$ is called a *negative body element*. A singleton `\+`$\vec{a}$ is also called a *negative (body) literal* and also written as $\neg a$. The *head H* is one of the following: (a) an *ordinary* head pr `::` a `@` tt, (b) a *distribution* head $f(\vec{t}) \sim t$ `@` tt, or (c) a *sum* head pr_1 `::` a_1 `@` $tt + \cdots + pr_m$ `::` a_m `@` tt, for some $m \geq 2$. Here, $a_{(i)}$ are ordinary atoms, t is an ordinary term, $pr_{(j)}$ are variables or built-in real-sorted terms, and tt is the *time term (of the rule)*. We write a `@` tt as short form for the ordinary head 1.0 `::` a `@` tt.

A body B is *variable-free* iff $X_B := \mathit{fvar}(b_1,\ldots,b_k) = \emptyset$. A rule is *range-restricted* if $\mathit{fvar}(H) \subseteq X_B$. A range-restricted rule is *variable-free* iff $X_B = \emptyset$. It is *ground* iff additionally $\mathit{fvar}(\vec{c}_i) = \emptyset$ for each $\vec{c}_i$. (In variable-free rules, the extra variables in the $\vec{c}_i$ are implicitly existentially quantified.) A *fact* is a rule with $k,n = 0$, still matching "H `:-` B" where B is empty, but also written as H. Notice that range-restricted facts are always ground. In the following we work with range-restricted rules only. A ground body B is *normal* if all its negative body elements are literals, i.e., B has the form $b_1,\ldots,b_k,\neg c_1,\ldots,\neg c_n$ where all c_i are ordinary atoms. A set of ground rules is *normal* iff all non-fact rules have a normal body and an ordinary head with probability 1. Facts with a head probability < 1.0 are also called *probabilistic facts*.

The standard notion of stratification [1] is equivalent to saying that the call graph of a normal logic program has no cycles going through negative body literals. Every strongly connected component of the call graph is called a stratum and contains the predicates that are defined mutually recursive with each other. In stratified programs, negative body literals can only contain atoms with predicates defined in lower strata. The semantics of stratified programs is defined by bottom-up model computation on the program's ground instances from lower to higher strata until fixpoint. In that, rule heads are inserted into the model only if their body is true. Importantly, the truth value of a negative body literal depends alone from the models at the lower strata, which have all been computed in full earlier.

Fusemate employs a weaker notion of stratification that we call *stratification by time and by predicates (SBTP)* [4]. SBTP generalizes standard stratification to a lexicographic combination of the standard ordering on time (the non-negative integers) and standard stratification. For the purpose of defining SBTP, every =-atom $f(\vec{t})$`=`t `@` tt or distribution head $f(\vec{t}) \sim t$ `@` tt is taken as an ordinary atom with *predicate* symbol f. We say that a set of rules is *time constrained* if for every non-fact rule (1) there is a variable n (for "now") that is the time term of a positive body literal b_i and such that

(i) the time term of every other ordinary positive body atom b_j is constrained to $\geq n$.

(ii) the time term of H is either (a) the variable n or else (b) constrained to $> n$.

(iii) the time term of every ordinary atom b in every c_j is constrained to either (a) $\geq n$ or else (b) $> n$.

Every such atom b_i is called a *pivot (of the rule)*.[2] To time constrained sets of rules we associate a call graph that ignores all rules satisfying (ii)-(b) and in all remaining rules ignores all ordinary atoms b in all c_j that satisfy (iii)-(b). Intuitively, (ii)-(b) rules define an atom in the future and, hence, are irrelevant for time-stratified model computation at "now"; the (iii)-(b) atoms are all computed strictly earlier than "now" and this way are already time-stratified. (All other cases need a second-tier call graph based stratification.) A predicate symbol of an ignored atom or an atom in an ignored rule is added as an isolated node to the call graph if it does not already occur in it. We say that a set of rules *is SBTP* if it is time-constrained and its call graph has no cycles through negative body elements. A *(Fusemate) program*

---

[2] Pivots are not needed in variable-free rules as the head can be used in lieu. For example, a rule `all_innocent @ 0 :- \+ guilty(Person) @ 0` can be accommodated by taking `all_innocent @ 0` as the pivot.

P is a finite set of range-restricted rules that is SBTP. The *stratification* induced by the call graph is a set $\{s_1, \ldots, s_m\}$ of sets of predicates equipped with a strict partial order $\prec$ ("lower-than") induced by the edges in the call graph. Every s_i is a strongly connected component in the call graph. We identify $\prec$ with an arbitrary extension to a linear order $s_1 \prec \cdots \prec s_m$. A *timed stratum* is a pair (n, s_k) where $n \geq 0$ and $1 \leq k \leq m$. We extend $\prec$ lexicographically to timed strata, also denoted by $\prec$. If a is a ground ordinary atom $p(\vec{t})$ `@` n define $strat(a) = (n, s_k)$ where $s_k \ni p$.

For example, a program rule `p@N :- q@N, \+ (r@T, T<=N)` has to be predicate stratified with `r` $\prec$ `q` to be SBTP. The rule `p@N :- q@N, \+ (r@T, T<N)` can have `r` and `q` unrelated because `T<N` already provides time constrained stratification. We found the two-level stratification useful in practice. It helps, for instance, to embed the fluent calculus. See [3, 2] for more motivating usages.

3 Bottom-Up Grounding and Distribution Semantics

In this section we define our bottom-up grounding, define a Distribution semantics for Fusemate programs, and prove the correctness of our pruning technique.

Let D be a set of ordinary atoms, the *domain*. A *(Herbrand) interpretation I over D* is any subset $I \subseteq D$. The atoms in I are "true" and those in $D \setminus I$ are "false". We define a satisfaction relation $\models$ as follows: for ground ordinary atoms a, $I \models a$ iff $a \in I$ for ground built-in atoms a, $I \models a$ iff a evaluates to $\top$; for sequences $\vec{a}$ of ground atoms, $I \models \vec{a}$ iff $I \models a$ for all $a \in \vec{a}$. A *matcher* for a sequence $\vec{a}$ of atoms to I is a grounding substitution γ for $fvar(\vec{a})$) such that $I \models \vec{a}\gamma$. For a negative body element, $I \models \backslash\texttt{+}\,\vec{c}$ iff there is no matcher for $\vec{c}$ to I. Notice that for ground ordinary atoms a, $I \models \neg a$ iff $a \notin I$ iff $I \not\models a$.

The following procedure eliminates negative body elements from a variable-free body B by introducing ground negative body literals. Parameterized in a domain D, it returns a (disjunctive) set of bodies that is equisatisfied with B for any interpretation I with domain D, see Lemma 3.3 below.

Procedure 3.1 (Body grounding $gnd_D(B)$) Let B be a variable-free body and D a set of ground ordinary atoms. Initiate a result set $R := \{B\}$. As long as R contains a body $B', \backslash\texttt{+}\,\vec{c}$ with $\backslash\texttt{+}\,\vec{c}$ not a ground negative body literal, do the following: let $E = \{\vec{c}\gamma_1, \ldots, \vec{c}\gamma_k\}$ where $\gamma_1, \ldots, \gamma_k$ are all matchers of $\vec{c}$ to D ($k = 0$ is possible). Let H be the set of all hitting sets of E.[3] H determines explanations for making the $\vec{c}\gamma_i$ false in D, as follows: let $R' = \{B', \neg d_1, \ldots, \neg d_k \mid \{d_1, \ldots, d_k\} \in H\}$ and set $R := (R \setminus \{B', \backslash\texttt{+}\,\vec{c}\}) \cup R'$. After completion, R is a set of ground bodies with negative body literals only. Finally apply obvious Boolean simplification rules to R, enabled by the constants $\top$ and $\bot$ present after evaluating built-ins. The resulting set is denoted by $gnd_D(B)$. ☐

Example 3.2 (Grounding) Consider the following program (time is implicitly 0).

```
0.5 :: p(a).    q(a).    0.5 :: p(c).
0.5 :: p(b).    q(b).    s :- \+ (p(X), q(X)).
```

Let $D = \{$`p(a)`$,$`p(b)`$,$`q(a)`$,$`q(b)`$,$`p(c)`$\}$. The grounding of the body $gnd_D($`\+ (p(X), q(X))`$)$ has four bodies `(¬q(a), ¬q(b))`, `(¬q(a), ¬p(b))`, `(¬p(a), ¬q(b))`, and `(¬p(a), ¬p(b))`. They represent all four ways to make the body true in interpretations $I \subseteq D$. Notice that $\{$`p(c)`$,$`q(c)`$\}$ is not in E and hence is ignored for grounding. ☐

---

[3] Given a set $E = \{E_1, \ldots, E_n\}$, $n \geq 0$, where each E_i is a sequence of elements, a hitting set h of E is a subset-minimal set that contains an element from each E_i. If $E = \emptyset$ then uniquely $h = \emptyset$.

Lemma 3.3 states the core property for correctness of grounding (proofs are in [5]).

Lemma 3.3 (Grounding preserves semantics) *Let D a domain, $I \subseteq D$ and $\vec{c}$ a sequence of atoms. Let $E = \{\vec{c}\gamma_1,\ldots,\vec{c}\gamma_k\}$ where $\gamma_1,\ldots,\gamma_k$ are all matchers of $\vec{c}$ to D ($k = 0$ is possible) and H the set of all hitting sets of E. Then $I \models \backslash\texttt{+}\vec{c}$ iff there is a $\{d_1,...,d_k\} \in H$ such that $I \models \neg d_1,...,\neg d_k$.*

The following function $\textsc{Normal}(r)$ expands a ground rule with a sum head into normal form. It splits disjunctions into disjoint cases using default negation, and extracts head probabilities by introducing probabilistic facts. This technique is well-known [10].

```
$\textsc{Normal}(pr_1 :: a_1 @ tt + \cdots + pr_m :: a_m @ tt \text{ :- } B)$          // $m \geq 1$, possibly empty $B$
  $M := 1.0$  // probability mass available for remaining cases
  $G := \{\mathrm{head}_B @ tt \text{ :- } B\}$  // result set of rules, introduce name $\mathrm{head}_B @ tt$ for head
  for $i \leftarrow 1..m$ do
      $G := G \cup \{a_i @ tt \text{ :- } \mathrm{head}_B @ tt, \neg\mathrm{case}_{B,1} @ tt, .., \neg\mathrm{case}_{B,i-1} @ tt, \mathrm{case}_{B,i} @ tt\}$
      $G := G \cup \{(M \cdot pr_i) :: \mathrm{case}_{B,i} @ tt\}$
      $M := M \cdot (1.0 - pr_i)$
  return $G$
```

$\textsc{Normal}$ is applied to ground rules with distribution heads $f(\vec{t}) \sim [t_1,\ldots,,t_m] @ tt$, where $[t_1,\ldots,,t_m]$ is a non-empty term list, by prior conversion to the sum head $1/m :: f(\vec{t}) = t_1 @ tt + \cdots + 1/m :: f(\vec{t}) = t_m @ tt$.

To define bottom-up grounding, let B be a body as in (1), D a set of ordinary atoms, and S a timed stratum. A matcher γ for $b_1,\ldots,b_k$ to D is called an *S-matcher for B to D* if either B is empty or else $strat(b_i\gamma) = S$ for some $1 \leq i \leq k$ and pivot b_i. Let $D_{\preceq S} = \{a \in D \mid strat(a) \preceq S\}$, analogously for $\prec$. For sets G of normal rules let $G_{\preceq S} = \{a \text{ :- } B \in G \mid strat(a) \preceq S \text{ for some } B\}$. A *query* is a normal body.

```
$\textsc{Ground}(P, Q, eot)$  // Input: $P$ program, $Q$ query, $eot \geq 0$ "end of time". Output: normal program
  $D := \emptyset$  // current domain
  $G := \emptyset$  // result ground program
  $R := Q$  // current regressed query
  for $n \leftarrow 0..eot$ do
    for $s \leftarrow s_1..s_m$ do  // $s_1 \prec \cdots \prec s_m$ is linearization of $P$'s stratification
      let $S = (n, s)$  // current timed stratum
      let $D^{\neg} = D_{\prec S}$  // finished domain for strictly earlier timed strata
      for $H \text{ :- } B \in P$ do
        for $\gamma$ sth $\gamma$ is an $S$-matcher of $B$ to $D_{\preceq S}$ do
          for $B' \in gnd_{D^{\neg}}(B\gamma)$ do
            if $B'$ is consistent with $R$ then  // see below, assume "true" for now
              let $G' = \textsc{Normal}(H\gamma \text{ :- } B')$
              $D := D \cup \{a \mid a \text{ :- } B'' \in G' \text{ for some } B'' \text{ or } pr :: a \in G' \text{ for some } pr\}$
              $G := G \cup G'$
      $R := reg_{\{r \in G_{\preceq S} \mid r \text{ is not a fact}\}}(R)$  // update regressed goal – ignore for now
  return $G_{\preceq(eot, s_m)}$
```

The $\textsc{Ground}$ procedure iterates over all timed strata, in increasing order, capped after *eot*. It thereby monotonically grows a domain D and a set of normal rules G obtained by S-matchers of the bodies of all rules to $D_{\preceq S}$, thereby grounding out negative body elements over $D^{\neg} = D_{\prec S}$. Notice that while D

and $D_{\preceq S}$ can grow while S is fixed, the subset $D^{\neg}$ remains unchanged. This holds thanks to using pivot literals stratified at S in S-matchers. For, by SBTP, all ordinary atoms in a negative body element are lower than S, but head atoms are at S or higher and, hence, cannot contribute to $D^{\neg}$. The consistency test on line 11 is ignored for now, i.e., treated as always "true". In Section 3 we will show that enabling it, as well as regression on line 15, does not affect semantics. Example 3.2 has an example for grounding. The ground rules and set D are computed by a run of GROUND on the program there.

We are going to define the Distribution semantics for `Fusemate` programs in terms of their grounding via the GROUND procedure and standard (ground level) stratification. The latter enables a standard fixpoint construction, which includes the standard Distribution semantics for definite programs as a special case.

Proposition 3.4 *Let P be a program and $n \geq 0$. Then* $\mathrm{GROUND}(P,n)$ *is SBPT.*

Let $P_g = \mathrm{GROUND}(P,n)$. From Proposition 3.4 it follows that P_g is stratified in the standard way. For, if P_g were not stratified then the call graph of P_g has a cycle going through a negative body literals. This cycle can be lifted to a cycle in the call graph of P by following the rules in P corresponding to the ones witnessing the cycle in P_g. However, with Proposition 3.4 this cycle does not exist.

This allows us to replace SBTP by standard stratification for ground programs. As said in the introduction, we do not consider programs with positive cycles here.[4] The call graph of P_g, hence, has no cycles. Similarly to above, let $\prec_g$ denote both the edge relation and its transitive closure. We can now define the distribution semantics of P in terms of least fixpoint models of probabilistic choices over P_g, as follows.

Definition 3.5 (Distribution semantics) Let P be a program, $n \geq 0$ and $P_g = \mathrm{GROUND}(P,n)$. Let $P_g = F \uplus R$ where $F = \{pr :: a \in P_g \mid pr < 1\}$ are all probabilistic facts in P_g. A *(probabilistic) choice* is any subset $X \subseteq F$. Define the probability of the choice X as $\mathrm{P}(X) = \Pi\{pr \mid pr :: a \in X \text{ for some } a\} \cdot \Pi\{1 - pr \mid pr :: a \in F \setminus X \text{ for some } a\}$. Define $I_R(X)$ as the least fixpoint model of the (non-probabilistic, $\prec_g$-stratified, normal) program $\{a \mid pr :: a \in X \text{ for some } pr\} \cup R$. Let Q be a query (i.e., a ground normal body). Define the *success probability of Q in P* as $\mathrm{P}(Q \mid P) := \Sigma\{\mathrm{P}(X) \mid X \subseteq F \text{ and } I_R(X) \models Q\}$. ☐

In Definition 3.5, any $I_R(X)$ is called a *model of P*, and any $I_R(X)$ such that $I_R(X) \models Q$ is called a *model of Q and P*. An interpretation *I satisfies right-uniqueness of* = iff there are no terms s, t and $u \neq t$ such that $\{s = t,\ s = u\} \subseteq I$. A program P is *admissible* if for every $n \geq 0$, each model of P satisfies right-uniqueness of =. Admissibility of models is needed for correctness of our query-guided grounding (Theorem 3.7).

Query-guided grounding. We complete the description of the GROUND procedure by definitions of the consistency test and the regression operator *reg*. We say that two normal bodies B_1 and B_2 are *consistent* if there is no atom a such that $\{a, \neg a\} \subseteq B_1 \cup B_2$ and there are no terms tt, s, t and $u \neq t$ such that $\{s = t \,@\, tt, s = u \,@\, tt\} \subseteq B_1 \cup B_2$. The term *inconsistent* means "not consistent". The idea is that B_1 and B_2 should be simultaneously satisfiable which is impossible if one of the these cases applies.[5] For example, $\{$`f(a) = b`$,$`f(a) = c`$\}$ is inconsistent. See Section 1 for an additional example. The semantics of consistency is given by right-uniqueness of =, as follows:

---

[4] A program like `fib(0, 0). fib(1, 1) . fib(N+1, N1+N2) :- fib(N, N1), N > 0, N <= 20, fib(N-1, N2).` is in scope as it is SBTP and its grounding has no positive cycle. A symmetry rule `r(X,Y) :- r(Y,X).` is out of scope.

[5] While in first-order logic $s = t$ and $s = u$ enforces $t = u$ this is not consistent with the unique name assumptions that is usually assumed with *Herbrand* interpretations.

Lemma 3.6 *Let B_1 and B_2 be inconsistent normal bodies. For every interpretation I satisfying right-uniqueness of* =, *not both $I \models B_1$ and $I \models B_2$.*

Goal regression. Let Q be a query, G a set of normal rules, and a an ordinary atom. (Goal) regression repeatedly expands subgoals a by SLD resolution but keeps only new subgoals common to *all* bodies of the rules for a. Formally, define *one-step regression* $reg^1_P(a) := \bigcap\{B \mid a \texttt{:-} B \in P\}$ and $reg^1_P(Q) := Q \cup \bigcup_{a \in Q} reg^1_P(a)$. Define $reg_P(Q)$ as the least fixpoint of reg^1_P starting from Q, i.e., $reg_P(Q) := (reg^1_P(Q))^k$ where k is the least integer such that $(reg^1_P(Q))^k = (reg^1_P(Q))^{k+1}$. See Section 1 for an example.

By *query-guided grounding* we mean the GROUND procedure with the call to *reg* on line 15 and the consistency test on line 11 enabled. By *unguided grounding* we mean line 15 disabled and the condition on line 11 be replaced by "true". The following theorem is our main theoretical result.

Theorem 3.7 (Correctness of query-guided grounding) *Let P be an admissible program and Q be a ground query. Then the success probability of Q in P is the same with query-guided grounding and with unguided grounding.*

Note 3.8 (Top-down grounding) The grounding procedure implemented in `Fusemate` also integrates an SLD-like procedure for top-down grounding. For instance, in the urn example a top-down rule `no(C) @ T :- \+ ( some(C) @ S, S <= T )` is useful to express that up to time `T` no ball of color `C` has been drawn. The call site has to instantiate both `C` and `T`. Top-down grounding is structurally similar to bottom-up grounding in that it also requires elimination of subgoals by introducing sets of possible explanations. This makes it not too difficult to incorporate when positive cycles are not allowed. □

4 Inference

We implemented the grounding algorithm of Section 3 in our `Fusemate` system. We also implemented functionality for probabilistic query answering for non-ground conditional queries based on a ground-level variable elimination algorithm at its core. In this section we briefly describe the latter components.

An *conditional input query* has the form $B \mid E$ consisting of a body $B = \ ?\text{-}\ b_1, \ldots, b_k, \backslash\texttt{+}\vec{c}_1, \ldots, \backslash\texttt{+}\vec{c}_n$ as in (1) and optional *evidence* $E = e_1, \ldots, e_m$ of ground ordinary atoms (see Section 1 for examples). Given $B \mid E$ and a program P, we want to compute the success probability of $B \mid E$ in P.We do this by grounding $B \mid E$ and P and reduction to query success probability (Definition 3.5). The grounding executes several calls to GROUND(P, Q, eot): (a) a call with query $Q = E$, (b) a call with query Q comprised of all ground literals in the conjunction (B, E), and (c) calls with queries $Q = (B\gamma, E)$, for every S-matcher for B to domain D computed in GROUND in step (b) and biggest stratum S. The value for eot is given as input.

The rationale for this staged process is to obtain in (b) a domain D as small as possible for exhaustive grounding of (B, E) in (c). The result reported to the user are answer substitutions γ and their probabilities, i.e., pairs $(\gamma, \mathrm{P}(B\gamma \mid E))$ where $\mathrm{P}(B\gamma \mid E) = \mathrm{P}(B\gamma, E \mid P)/\mathrm{P}(E \mid P)$ for every *answer substitution* γ in step (c). Notice that for each of these computations the optimized ground programs P_g computed by GROUND in steps (a), (b) and (c) are available for that. Theorem 3.7 guarantees the correctness.

With the above considerations, the algorithmic problem now is to compute $P(Q \mid P_g)$ for a query Q and ground program P_g. Many PLP systems would convert P_g into an equivalent weighted Boolean formula and apply weighted model counting algorithms on its d-DNNF normal form (or other) [8, 6]. For now, `Fusemate` implements a variable elimination algorithm based on finding all SLD proofs of the query.

To explain the main ideas, every SLD proof provides a factor by multiplying the probabilities of the facts at its leaves. The success probability of Q in P_g then is the sum of the probabilities of these factors. For correctness, the bodies of all rules with the same head have to be made disjoint (inconsistent) first. This can be achieved with indicator variables [10]: every collection $\{h \mathrel{:\!-} B_1, \cdots, h \mathrel{:\!-} B_m\}$ of all rules in P_g with the same head h is replaced by the rules $\bigcup_i(\{h_i \mathrel{:\!-} B_i\} \cup \{h \mathrel{:\!-} \neg h_1, \ldots, \neg h_{i-1}, h_i\})$ where h_i is a fresh predicate symbol indicating which case is tried exclusively. (Probabilistic facts are not affected, as they are all made with pairwise distinct atoms, by design of the NORMAL procedure.) Our variable elimination algorithm VE can be applied to such programs and (ground, conjunctive) queries Q. It caches intermediate results, and it prunes inconsistent queries as soon as they come up:

```
VE($P_g, Q$)        // Input: $P_g$ normal program with disjoint bodies, $Q$ query. Output: $\mathrm{P}(Q \mid P)$
 1 $Cache := \emptyset$  // cached pairs (Query, Probability)
 2 RETURN($Q, R$)  // Caches and returns result $R$ for $Q$
 3    $Cache := Cache \cup \{(Q, R)\}$
 4    $R$
 5 INNER($Q$)  // Input: worked off and remaining query literals $Q$. Output: probability of $Q$
 6    if $Q = \emptyset$ then 1.0
 7    else if $Q$ is inconsistent then 0.0  // from here on $Q$ is consistent
 8    else if $(Q, pr) \in Cache$ then $pr$
 9    else let $L \in Q$ sth $L$ is maximal in $Q$ wrt. the stratification order $\prec_g$
10       let $Q' = Q \setminus \{L\}$
11       if $L = \neg A$ then  // $\mathrm{P}(\neg A, Q') = \mathrm{P}(Q') - \mathrm{P}(A, Q')$
12          RETURN($Q$, INNER($Q'$) − INNER($Q' \cup \{A\}$))
13       else if $pr :: L \in P_g$ then
14          RETURN($Q$, $pr \cdot$ INNER($Q'$))
15       else let $Bs = \{B \mid L \mathrel{:\!-} B \in P_g\}$  // $Bs$ is new subgoals as per rules for $L$ in $P_g$
16          RETURN($Q$, $\Sigma_{B \in Bs}$INNER($B \cup Q'$))
17 INNER($Q$)
```

The main ideas behind VE should be clear from the comments above, but some notes are in order. Starting from INNER(Q) the query literals in Q are successively resolved away using facts and rules from P_g (except negative literals, which are expanded in the obvious way in line 12). Line 7 is the mentioned early pruning check. For example, let P_g contain, among others, `0.5::p.` `a=1 :- p.` `a=2 :- -p.` and $Q = \{\ldots,$ `a=1`, `a=2`$\}$. Notice that the stated rules do not offend admissibility as no model can make both `a=1` and `a=2` simultaneously true. That is, the conjunction `a=1, a=2` does not hold in any model. Early pruning discovers this fact via inconsistency of `a=1, a=2` and assigns probability 0 to Q. This avoids redundantly solving potentially present subgoals of Q that are greater wrt. $\prec_g$ than `a=1` and `a=2`. Another points to notice is that no probabilistic fact is multiplied in more than once. This follows from working off the query literals in decreasing stratification order.

5 Application Examples and Benchmarks

In this section we evaluate the practical effectiveness of Fusemate and query-guided grounding. Fusemate and the problem sets described below can be downloaded from `https://bitbucket.csiro.au/projects/MOVEMENTANALYTICS/repos/fusemate-distrib/`. All experiments were run on a Dell Latitude 7330 laptop, with Windows 11 operating system (32 GB RAM, 3.2 GHz). For each problem, we compared timings in Fusemate and ProbLog by running several sets of queries for increasing complexity in the query. We only summarize our findings here. More details on the ProbLog and Fusemate encodings are in [5].

Markov model. This problem is from the ProbLog tutorial Markov chain example. [6] The code models the movement in time between three locations *a*, *b* and *c*, where the probability of moving to the next location depends only on the current location. We timed three kinds of queries which we refer to as *timesteps*, *specificity* and *timepoint*. The timesteps query asks for the probability of being at position *a* during a whole period of $N = 0,\ldots,80$ timesteps. The specificity query asks for the probability of being in a certain location across nine timesteps, where complexity *N* is increased by decreasing the specificity of those locations. We do that by replacing fixed location *a* with a variable, so the program has to repeat the calculation for all possible locations, increasing the branching. The timepoint query asks for the probability of being at location *a* at a final time point *N*. The logic program, sample queries and runtime performance are in Figure 1.

```
%% Markov Model
in ~ [a, b, c] @ 0.
in ~ [[a, 0.9],[b, 0.05],
     [c, 0.05]] @ T+1 :- in=a @ T.
in ~ [[a, 0.7],[c, 0.3]] @ T+1 :- in=b @ T.
in ~ [[a, 0.8],[c, 0.2]] @ T+1 :- in=c @ T.

%% Time steps N = 20
?- in=a@0, in=a@1,.., in=a@20.

%% Specificity, N = 7
?- in=a@0, in=a@1, in=L2@2,..,in=L8 @ 8.

%% Timepoint, N = 20
?- in=a@23.
```

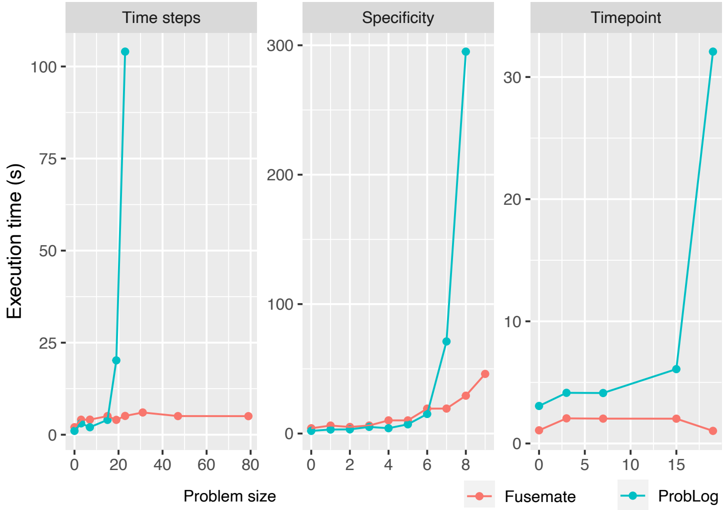


Figure 1: Timings for three kinds of queries to the Markov chain in Fusemate (red) and ProbLog (blue).

Results in Figure 1 indicate that Fusemate performs better than ProbLog for increasing problem complexity. The Fusemate times are measured with query-guided grounding on. However, Fusemate can also solve the problems without it. Runtimes and number of ground rules generated raise by a factor of 2-3 without guidance. It seems that Fusemate's variable elimination procedure is better suited to this example than the ProbLog inference engine. Inconsistency pruning (line 7 in VE) did not have an impact on these problems.

Hidden Markov model. This example is based on the example from the Wikipedia page for hidden Markov models[7]. It models the situation where a prediction of the weather (sunny/rainy) is determined by observations of how much water due to precipitation has been collected in a bowl that is placed outside. The idea is that more water will be collected when it rains more, which is the measurement of

---

[6] https://dtai.cs.kuleuven.be/problog/tutorial/various/08_bayesian_dataflow.html

[7] https://en.wikipedia.org/wiki/Hidden_Markov_model

how the weather has been. For simplicity, this example assumes no evaporation, so the observations of water in the bowl are non-decreasing. The logic program was described in Section 1.

We timed three kinds of filtering queries for estimating the current state based on past observations trending around *slightly rainy*, *sunny*, and *mixed*, scenarios, respectively. In the slightly rainy scenario each timepoint increased the amount of water observed by four, a value that can occur from either the sunny or the rainy distributions. In the sunny scenario, each observation was of zero precipitation, so the observations must have come from the sunny distribution. In the mixed weather scenario, a variety of observations were included in this query, which meant there was a mixture of observations from both the sunny and rainy distributions. For each scenario, the query complexity N was increased by increasing the number of observations and the timepoint at which the final state is predicted. The results and sample queries are described in Figure 2.

```
%% See Introduction for program

%% Queries for N=3
%% Sunny
?-state=X @ 3 | obs=0 @ 1, obs=0 @ 2, obs=0 @ 3.

%% Rainy
?-state=X @ 3 | obs=4 @ 1, obs=8 @ 2, obs=12 @ 3

%% Mixed
state=X @ 3 | obs=0 @ 1, obs=4 @ 2, obs=24 @ 3.
```

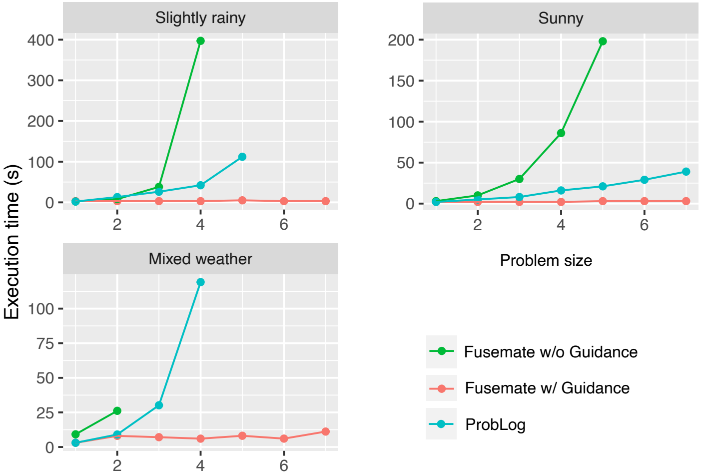


Figure 2: Timings for three kinds of queries to the hidden Markov model precipitation example: Fusemate without query-guidance (green), Fusemate with query-guidance (red), and ProbLog (blue).

For all problems, Fusemate with unguided grounding was the slowest, but Fusemate with query-guided grounding consistently outperformed ProbLog. What sets this example apart from the Markov Chain example is its high branching rate (30 support values in a rain state, vs. two or three Markov Chain) , and the implicit dependence of each state on its history via the accumulated rainfall up to that state.

Inconsistency pruning did not play a role in Fusemate's good performance in the problem above. We tested this by disabling the test on line 7 in the VE procedure. With less constraining queries, however, inconsistency pruning can lead to drastic performance improvements. We experimented with relaxing the evidence of a slightly modifed "sunny" scenario by leaving out observations. For a query size $N = 4$, for instance, we obtained the following runtime results (in seconds):

		Inconsistency pruning	
Query		**Off**	**On**
`?- state=S @ 4 | obs=0 @ 1, obs=0 @ 2, obs=0 @ 3,`	`obs=10 @ 4`	7.5	7.5
`?- state=S @ 4 | obs=0 @ 1, obs=0 @ 2,`	`obs=10 @ 4`	44.5	**13.5**
`?- state=S @ 4 | obs=0 @ 1,`	`obs=10 @ 4`	>2000	**30.0**
`?- state=S @ 4 |`	`obs=10 @ 4`	>2000	**180.75**

In addition to overall solution times we were also interested in comparing groundings. For ProbLog we observed unusual high grounding times, and for Fusemate with unguided grounding we observed large but quickly computed groundings (which become unmanageable for inference quickly). For the mixed weather problem, we observed:

	Fusemate #ground rules		**ProbLog**		
N	**query-guided**	**unguided**	**total time**	**grounding time**	**#ground rules**
2	2200	6500	9.0	8.3	53
3	2270	12900	30	19	276
4	2300	21400	119	33	499
5	2400	32000		50	682
6	2470	45000		65	839
7	2500	60000		95	1068

Grounding sizes between Fusemate and ProbLog are only roughly comparable. Fusemate outputs ground normal rules, where ProbLog outputs ground rules with annotated disjunctions, where head probabilities are left in place. The Fusemate total times are between 1 and 7 seconds and not listed in the table. For ProbLog, grounding time is still well below solving time but well above Fusemate total times.

6 Conclusion

In this paper, we proposed a bottom-up grounding approach for an expressive probabilistic logic programming language (expressive form of stratification, expressive default negation, dynamic distributions). We defined the semantics of the input languages as an extension of the standard Distribution semantics via a standard fixpoint construction after grounding and transforming away default negation. As the main contribution of this paper, we integrated and proved correct a novel technique for avoiding ground instances that are irrelevant for proving a given query. Grounding, transforming away default negation, and query-guided pruning are tightly integrated and carried out incrementally along the program's stratification order. They rest on a built-in semantics for equations as right-unique relations, which is appropriate, e.g., for representing (finite) distributions.

We showed the effectiveness of query-guided pruning experimentally. The rationale is to tackle combinatorial explosion *during* grounding instead of attempting optimizations *afterwards*. Without guidance, example problems tend to grow to unmanageable size quite quickly. Our system outperformed ProbLog on hidden Markov model filtering problems with a high branching rate. (On other domains not reported here we found that ProbLog often performs better than Fusemate.) We also showed that the performance of our top-down variable elimination algorithm benefits from building-in inconsistency pruning. While the result are somewhat limited in scope, we suggest that the research direction begun in this paper looks promising for further exploration. We will consider optimized off-the-shelf backends for weighted inference as a possibly better alternative to our variable elimination algorithm in such cases. We also plan to integrate a magic set transformation, which is complementary to our query-guided grounding.

We conjecture that query-guided grounding enables Fusemate to solve filtering queries in linear time. This would be the same complexity as the dedicated forward-backward algorithm. The proof hinges on an analysis of the solution caching mechanism in Fusemate's variable elimination inference algorithm.

Acknowledgements. We thank the reviewers for their valuable comments.

References

[1] K. R. Apt, H. A. Blair, A. Walker. Towards a theory of declarative knowledge. In *Foundations of deductive databases and logic programming*. Morgan Kaufmann, 1988. doi:10.1016/B978-0-934613-40-8.50006-3.

[2] P. Baumgartner. Possible Models Computation and Revision – A Practical Approach. In N. Peltier, V. Sofronie-Stokkermans, eds.,*IJCAR*, LNCS, Springer, 2020. doi:10.1007/978-3-030-51074-9_19

[3] P. Baumgartner. Combining Event Calculus and Description Logic Reasoning via Logic Programming. In B. Konev, G. Reger, eds., *FroCoS*, LNCS, Springer, 2021. doi:10.1007/978-3-030-86205-3_6.

[4] P. Baumgartner. The Fusemate Logic Programming System. In A. Platzer, G. Sutcliffe, eds., *CADE 28*, LNCS, Springer, 2021. doi:10.1007/978-3-030-79876-5_34.

[5] P. Baumgartner, E. Tartaglia. Bottom-Up Grounding in the Probabilistic Logic Programming System Fusemate, May 2023. doi:10.48550/arXiv.2305.18924.

[6] M. Chavira, A. Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6):772–799, 2008. doi:10.1016/j.artint.2007.11.002.

[7] L. De Raedt, A. Kimmig. Probabilistic (logic) programming concepts. *Machine Learning*, 100 (1), 2015. doi:10.1007/s10994-015-5494-z.

[8] D. Fierens, G. Van Den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, L. De Raedt. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *TPLP*, 15(3), 2015. doi:10.1017/S1471068414000076.

[9] B. Gutmann, I. Thon, A. Kimmig, M. Bruynooghe, L. De Raedt. The magic of logical inference in probabilistic programming. *TPLP*, 11(4-5), 2011. doi:10.1017/S1471068411000238.

[10] D. Poole. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64(1), 1993. doi:10.1016/0004-3702(93)90061-F.

[11] F. Riguzzi, T. Swift. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *TPLP*, 11(4-5), 2011. doi:10.1017/S147106841100010X.

[12] F. Riguzzi, E. Bellodi, R. Zese, G. Cota, E. Lamma. A survey of lifted inference approaches for probabilistic logic programming under the distribution semantics. *IJAR*, 80, 2017. doi:10.1016/j.ijar.2016.10.002.

[13] T. Sato. A Statistical Learning Method for Logic Programs with Distribution Semantics. In *ICLP*, 1995. doi:10.7551/mitpress/4298.003.0069.

[14] T. Sato, Y. Kameya. PRISM: a language for symbolic-statistical modeling. In *IJCAI*, 1997.

[15] A. Skarlatidis, A. Artikis, J. Filippou, G. Paliouras. A Probabilistic Logic Programming Event Calculus. *TPLP*, 15(2), 2015. doi:10.1017/S1471068413000690.

[16] E. Tsamoura, V. Gutierrez-Basulto, A. Kimmig. Beyond the Grounding Bottleneck: Datalog Techniques for Inference in Probabilistic Logic Programs. In *AAAI Conference on Artificial Intelligence*, 34(06), 2020. doi:10.1609/aaai.v34i06.6591.

[17] J. Vennekens, S. Verbaeten, M. Bruynooghe. Logic Programs with Annotated Disjunctions. In *ICLP*, Springer, 2004. doi:10.1007/978-3-540-27775-0_30.

[18] J. Vennekens, M. Denecker, M. Bruynooghe. CP-logic: A language of causal probabilistic events and its relation to logic programming. *TPLP*, 9(3), 2009. doi:10.1017/S1471068409003767.

[19] J. Vlasselaer, G. Van den Broeck, A. Kimmig, W. Meert, L. De Raedt. $T_{\mathcal{P}}$ -Compilation for inference in probabilistic logic programs. *IJAR*, 78, 2016. doi:10.1016/j.ijar.2016.06.009.

Generalizing Level Ranking Constraints for Monotone and Convex Aggregates

Tomi Janhunen
Computing Sciences, Tampere University, Finland
Department of Computer Science, Aalto University, Finland
tomi.janhunen@tuni.fi

In answer set programming (ASP), answer sets capture solutions to search problems of interest and thus the efficient computation of answer sets is of utmost importance. One viable implementation strategy is provided by translation-based ASP where logic programs are translated into other KR formalisms such as Boolean satisfiability (SAT), SAT modulo theories (SMT), and mixed-integer programming (MIP). Consequently, existing solvers can be harnessed for the computation of answer sets. Many of the existing translations rely on program completion and level rankings to capture the minimality of answer sets and default negation properly. In this work, we take level ranking constraints into reconsideration, aiming at their generalizations to cover aggregate-based extensions of ASP in more systematic way. By applying a number of program transformations, ranking constraints can be rewritten in a general form that preserves the structure of monotone and convex aggregates and thus offers a uniform basis for their incorporation into translation-based ASP. The results open up new possibilities for the implementation of translators and solver pipelines in practice.

1 Introduction

Answer set programming (ASP, [8]) offers rich rule-based languages for knowledge representation (KR) and reasoning. Given some search or optimization problem of interest, its *encoding* in ASP is a logic program whose answer sets capture solutions to the problem. Thus the efficient computation of answer sets is of utmost importance. One viable implementation strategy is provided by *translation-based* ASP where logic programs are translated into other KR formalisms such as Boolean satisfiability (SAT, [5]), SAT modulo theories (SMT, [4]), or mixed-integer programming (MIP, [32]). Consequently, existing solver technology can be harnessed for the computation of answer sets.

The semantics of answer set programming rests on *stable models* [14] that incorporate a notion of minimality and give a declarative semantics for default negation. Capturing these aspects in satisfaction-based formalisms such as pure SAT is non-trivial; see, e.g., [17, 22]. There are also various syntactic aggregations [2] that enable compact encodings but whose translation is potentially expensive if there is no respective primitive in the target formalism. A typical translation consists of several steps such as (i) *normalization* [7], (ii) *instrumentation* for loop prevention [6, 17, 23], and (iii) *completion* [10]. The first step concerns the removal of syntactic extensions that have been introduced to increase the expressive power of ASP in favor of *normal* rules. The second step either transforms the program or adds suitable constraints so that the difference between stable and *supported models* disappears. The third step captures supported models by transforming rules into equivalences. Ideally, the syntactic details of the target language are deferred during translation and incorporated at the very end; either after or while forming the completion. This strategy realizes a *cross-translation* approach [19] in analogy to modern compiler designs.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 101–115, doi:10.4204/EPTCS.385.12

Many of the existing translations [20, 25, 26] essentially rely on *level ranking constraints* formalized by Niemelä [27] as formulas in difference logic [28]. Such constraints describe *level numbers* that order the atoms of a normal program in such a way that stable models can be distinguished among the supported ones [11]. Thus, level numbers are essential when it comes to capturing the minimality of stable models and the semantics of default negation properly. As shown in [17], level numbers can be made unique so that they match with the levels of atoms obtained by applying the *immediately true* operator $\mathbf{T}_P$ iteratively. Uniqueness can also be enforced in terms of *strong* level ranking constraints [27]. Unique level numbers are also highly desirable when aiming at one-to-one correspondences with stable models, e.g., when counting solutions to problems or carrying out probabilistic inference [12].

In this work, we take level ranking constraints into reconsideration, aiming at generalizations that cover aggregate-based extensions of ASP in a more systematic way. So far, only normal programs are truly covered [17, 27] and the normalization of input programs is presumed. The generalization for weight constraint programs (WCPs), as sketched by Liu et al. [25], concerns only weak constraints and is confined to translations into MIP. However, the idea of avoiding or delaying normalization is interesting as such, opening up new possibilities for ordering the translation steps discussed above. For instance, if $\mathrm{NORM}(\cdot)$ and $\mathrm{LRC}(\cdot)$ stand for translations based on normalization and level ranking constraints, respectively, it would be highly interesting to compare $\mathrm{LRC}(\mathrm{NORM}(P))$ with potential generalizations $\mathrm{LRC}(P)$ that express level ranking constraints at an aggregated level. Such representations are expected to be more compact and to favor level rankings with fewer variables. The resulting formulas can also be *Booleanized* afterwards [16], if translations toward SAT are desirable, or rewritten in some other form that complies with the intended back-end formalism. In the sequel, we use translations $\mathrm{LRC}(\mathrm{NORM}(P))$ of increasingly complex programs P as guidelines when lifting level ranking constraints for aggregates. The idea is to cover program classes involving standard aggregations subject to recursion. It turns out that the structure of monotone and convex aggregates can be preserved to a high degree, offering a uniform basis for their incorporation into translation-based ASP. On the one hand, the resulting generalizations exploit *ordered completion* [3] in the reformulation of *weak* level ranking constraints but, on the other hand, make novel contribution when imposing the uniqueness of level rankings with *strong* ones.

The rest of this article is structured as follows. We begin by recalling the basic notions of *logic programs* in Section 2, including the usual syntactic fragments, stable and supported model semantics, and other concepts relevant for this work. Then, in Section 3, we explain the details of ranking constraints in their standard form corresponding to *normal* logic programs. Actually, we present them in a slightly rewritten form in order to pave the way for their generalization for monotone aggregates in Section 4. Therein, we begin the analysis from the case of (positive) cardinality and weight rules and, eventually, incorporate negative conditions to ranking constraints. To illustrate the generality of the results even further, we investigate how certain convex aggregates get also covered via appropriate program transformations in Section 5. Finally, the conclusions of this work are presented in Section 6.

2 Preliminaries

In the sequel, we will consider *propositional logic programs* that are finite sets rules of the forms (1)–(5) below. In the rules, a, a_i's, b_j's, and c_k's are *propositional atoms* (or *atoms* for short) and $\sim$ denotes *default negation*. The rules in (1)–(5) are known as *normal*, *choice*, *cardinality*, *weight*, and *disjunctive* rules, respectively. Each rule has a *head* and a *body* separated by the $\leftarrow$ sign, and the rough intuition is that if the *condition(s)* formed by the rule body are satisfied, then the respective head atom a in (1)–(4), or some of the head atoms $a_1, \ldots, a_h$ in (5) can be derived.

$$\begin{array}{rcll} a & \leftarrow & b_1,\ldots,b_n,{\sim}c_1,\ldots,{\sim}c_m. & (1)\\ \{a\} & \leftarrow & b_1,\ldots,b_n,{\sim}c_1,\ldots,{\sim}c_m. & (2)\\ a & \leftarrow & l \leq \{b_1,\ldots,b_n,{\sim}c_1,\ldots,{\sim}c_m\}. & (3)\\ a & \leftarrow & w \leq \{b_1=w_{b_1},\ldots,b_n=w_{b_n},{\sim}c_1=w_{c_1},\ldots,{\sim}c_m=w_{c_m}\}. & (4)\\ a_1 \mid \ldots \mid a_h & \leftarrow & b_1,\ldots,b_n,{\sim}c_1,\ldots,{\sim}c_m. & (5) \end{array}$$

The choice regarding the head of (2) is optional while (5) insists on deriving at least one head atom a_i in a minimal way, as to be detailed in Definition 1. A positive body condition b_j holds if b_j can be derived by some other rules whereas ${\sim}c_k$ holds if c_k cannot be derived. A cardinality rule (3) demands that at least l of such conditions are met to activate the rule. Weight rules (4) are similar but body conditions b_j and ${\sim}c_k$ are valued by their respective non-negative integer weights w_{b_j} and w_{c_k} when it comes to reaching the bound w. In the sequel, we use shorthands $\mathrm{B}^+(r)=\{b_1,\ldots,b_n\}$, $\mathrm{B}^-(r)=\{c_1,\ldots,c_m\}$, and $\mathrm{B}(r)=\{b_1,\ldots,b_n\}\cup\{{\sim}c_1,\ldots,{\sim}c_m\}$ when referring to the body conditions occurring in a rule r. The set of head atoms in r is denoted by $\mathrm{H}(r)$ and for entire programs P, we define $\mathrm{H}(P)=\bigcup_{r\in P}\mathrm{H}(r)$.

Typical (syntactic) classes of logic programs are as follows: *normal* logic programs (NLPs) consist of normal rules (1) and the same can be stated about *disjunctive* logic programs (DLPs) and disjunctive rules (5) that are normal as a special case ($h=1$). The class of *weight constraint programs* (WCPs) [30] is essentially based on normal rules (1) and the aggregated rule types in (2)–(4), out of which weight rules are expressive enough to represent the class of WCPs alone. Contemporary ASP systems—aligned with the ASP-core-2 language standard [9]—support these fragments so well that programmers can mix rule types freely in their encodings. When the fragment is not important, we may refer to *logic programs* or *programs* for short. Finally, we say that a rule is *positive*[1] if $m=0$ and it is of the forms (1), or (3)–(5). An entire program is called positive if its rules are all positive.

The *signature* $\mathrm{At}(P)$ of a logic program P is the set of atoms that occur in the rules of P. An *interpretation* $I\subseteq\mathrm{At}(P)$ of P tells which atoms $a\in\mathrm{At}(P)$ are *true* ($a\in I$, also denoted $I\models a$) whereas others are *false* ($a\in\mathrm{At}(P)\setminus I$, denoted $I\not\models a$). Atoms are also called *positive literals*. Any *negative literal* ${\sim}c$, where c is an atom, is understood classically, i.e., $I\models{\sim}c$, iff $I\not\models c$. The relation $\models$ extends for the bodies of normal/choice/disjunctive rules r as follows: $I\models\mathrm{B}(r)$ iff $\mathrm{B}^+(r)\subseteq I$ and $\mathrm{B}^-(r)\cap I=\emptyset$. The body $l\leq\mathrm{B}(r)$ of a cardinality rule r is satisfied in I iff $l\leq|\mathrm{B}^+(r)\cap I|+|\mathrm{B}^-(r)\setminus I|$. More generally, the body of a weight rule r in (4) is satisfied in I iff the *weight sum* $\mathrm{WS}_I(b_1=w_{b_1},\ldots,b_n=w_{b_n},{\sim}c_1=w_{c_1},\ldots,{\sim}c_m=w_{c_m})=\sum_{b\in\mathrm{B}^+(r)\cap I}w_b+\sum_{c\in\mathrm{B}^-(r)\setminus I}w_c$ is at least w. For rules r, we have $I\models r$ iff the satisfaction of its body implies the satisfaction of its head, except that choice rules (2) are always satisfied. An interpretation $I\subseteq\mathrm{At}(P)$ is a (*classical*) *model* of a program P, denoted $I\models P$, iff $I\models r$ for each $r\in P$. A model $M\models P$ is $\subseteq$*-minimal* iff there is no $M'\models P$ such that $M'\subset M$. The set of $\subseteq$-minimal models of P is denoted by $\mathrm{MM}(P)$. If P is positive and non-disjunctive then $|\mathrm{MM}(P)|=1$ and the respective *least model* of P is denoted by $\mathrm{LM}(P)$.

Definition 1 (Stable models [14, 15, 30])**.** *For a program P and an interpretation $I\subseteq\mathrm{At}(P)$, the* reduct *P^I of P with respect to I contains*

1. *a rule $a\leftarrow\mathrm{B}^+(r)$ for each normal rule* (1) *such that* $\mathrm{B}^-(r)\cap I=\emptyset$*, and for each choice rule* (2) *such that $a\in I$ and* $\mathrm{B}^-(r)\cap I=\emptyset$*;*
2. *a rule $a\leftarrow l'\leq\mathrm{B}^+(r)$ for each cardinality rule* (3) *and the bound* $l'=\max(0,l-|\mathrm{B}^-(r)\setminus I|)$*;*

---

[1] Note that the head $\{a\}$ of choice rule embeds hidden (double) negation since it can be expressed as $a\leftarrow{\sim}{\sim}a$.

3. *a rule* $a \leftarrow w' \leq \{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}\}$ *for each weight rule* (4) *and the bound* $w' = \max(0, w - \mathrm{WS}_I(\sim c_1 = w_{c_1}, \ldots, \sim c_m = w_{c_m}))$*; and*
4. *a rule* $a_1 \mid \ldots \mid a_h \leftarrow \mathrm{B}^+(r)$ *for each disjunctive rule* (5) *such that* $\mathrm{B}^-(r) \cap I = \emptyset$*.*

An interpretation $M \subseteq \mathrm{At}(P)$ *is a* stable model *of the program* P *iff* $M \in \mathrm{MM}(P^M)$.

Example 1. *Consider a cardinality rule* $a \leftarrow 1 \leq \{b_1, \ldots, b_n\}$ *with choice rules* $\{b_1\}. \ldots \{b_n\}$*. Besides the empty stable model* $\emptyset$*, these rules induce* $2^n - 1$ *stable models* $M = \{a\} \cup N$ *with* $\emptyset \subset N \subseteq \{b_1, \ldots, b_n\}$*: the head* a *is set true whenever* at least *one of* $b_1, \ldots, b_n$ *is chosen to be true.* ■

In the sequel, we mostly concentrate on non-disjunctive programs P. Then, the stability of $M \subseteq \mathrm{At}(P)$ can also be captured with the fixed point equation $M = \mathrm{LM}(P^M)$. Moreover, the well-known $\mathbf{T}_P$ operator, when applied to an interpretation $I \subseteq \mathrm{At}(P)$, produces the set of atoms $a \in \mathrm{At}(P)$ that are *immediately true* under I, i.e., for which there is a positive rule r having a as the head and whose body is satisfied by I. It follows that $M \models P$ holds for a non-disjunctive program P iff $\mathbf{T}_{P^M}(M) \subseteq M$ and M is a *supported model* of P iff $M = \mathbf{T}_{P^M}(M)$. Given a supported model M, the support is provided by the set of rules $\mathrm{SuppR}(P, M) \subseteq P$ whose bodies are satisfied by M. Since $\mathrm{LM}(P^M)$ is obtained as the least fixed point $\mathbf{T}_{P^M} \uparrow^{\infty} (\emptyset)$, each stable model of P is also supported. We write $\mathrm{SM}(P)$ and $\mathrm{SuppM}(P)$ for the sets of stable and supported models of P, respectively. Thus $\mathrm{SM}(P) \subseteq \mathrm{SuppM}(P)$ holds in general.

Next we recall some concepts related to modularity. First, given a WCP P, the set of *defining rules* for an atom $a \in \mathrm{H}(P)$ is $\mathrm{Def}_P(a) = \{r \in P \mid a \in \mathrm{H}(r)\}$. Thus P can be partitioned as $\bigcup_{a \in \mathrm{H}(P)} \mathrm{Def}_P(a)$. Second, the *positive dependency graph* of P is $\mathrm{DG}^+(P) = \langle \mathrm{At}(P), \succeq_P \rangle$ where $a \succeq_P b$ holds for $a, b \in \mathrm{At}(P)$, if $a \in \mathrm{H}(r)$ and $b \in \mathrm{B}^+(r)$ for some rule $r \in \mathrm{Def}_P(a)$. A *strongly connected component* (SCC) of $\mathrm{DG}^+(P)$ is a maximal subset $S \subseteq \mathrm{At}(P)$ such that all distinct atoms $a, b \in S$ depend on each other via directed paths in $\mathrm{DG}^+(P)$. For an atom $a \in \mathrm{H}(P)$, the SCC of a is denoted by $\mathrm{SCC}(a)$. As shown in [29], each SCC S of a WCP P gives rise to a *program module* $P_S = \bigcup_{a \in S} \mathrm{Def}_P(a)$ where pure body atoms $b \in \mathrm{At}(P_S) \setminus S$ are treated as *input atoms* taking any truth value, intuitively defined by choice rules $\{b\}$. This yields a set of stable models $\mathrm{SM}(P_S)$ for each module P_S based on Definition 1. Given two stable models $M \in \mathrm{SM}(P)$ and $N \in \mathrm{SM}(Q)$, we say that M and N are mutually *compatible*, if they agree on the truth values of atoms in $\mathrm{At}(P) \cap \mathrm{At}(Q)$, i.e., $M \cap \mathrm{At}(Q) = N \cap \mathrm{At}(P)$. The *module theorem* of [29] states that the stable models of P can be obtained as mutually compatible collections of stable models $M_1, \ldots, M_n$ for the program modules $P_{S_1}, \ldots, P_{S_n}$ induced by the SCCs $S_1, \ldots, S_n$ of P.

Finally, some notions of equivalence should be introduced. Logic programs P and Q are *weakly equivalent*, denoted $P \equiv Q$, iff $\mathrm{SM}(P) = \mathrm{SM}(Q)$. They are *strongly equivalent*, denoted $P \equiv_{\mathrm{s}} Q$, iff $P \cup R \equiv Q \cup R$ for any other context program R [21]. Then $P \equiv_{\mathrm{s}} Q$ implies $P \equiv Q$ but not vice versa. Strong equivalence can be characterized by using only contexts formed by *unary* positive rules $a \leftarrow b$, or semantically by using SE-models [31]. To address the correctness of various translations, however, more fine-grained relations become necessary. The signature $\mathrm{At}(P)$ of a logic program P can be split into *visible* and *hidden* parts $\mathrm{At}_{\mathrm{v}}(P)$ and $\mathrm{At}_{\mathrm{h}}(P)$, respectively. Given a stable model $M \in \mathrm{SM}(P)$, only its visible projection $M \cap \mathrm{At}_{\mathrm{v}}(P)$ is relevant when comparing P with other programs. Thus, P and Q are *visibly equivalent*, denoted $P \equiv_{\mathrm{v}} Q$, iff $\mathrm{At}_{\mathrm{v}}(P) = \mathrm{At}_{\mathrm{v}}(Q)$ and $M \cap \mathrm{At}_{\mathrm{v}}(P) = N \cap \mathrm{At}_{\mathrm{v}}(Q)$ holds for each pair of models $M \in \mathrm{SM}(P)$ and $N \in \mathrm{SM}(Q)$ in a bijective correspondence [17]. There is a generalization of both $\equiv_{\mathrm{v}}$ and $\equiv_{\mathrm{s}}$, viz. *visible strong equivalence* $\equiv_{\mathrm{vs}}$, that incorporates context programs R that *respect the hidden atoms* of P and Q for comparisons [7]. The correctness of normalization has been addressed in this sense. E.g., for a weight rule r in (4), $\{r\} \equiv_{\mathrm{vs}} \mathrm{NORM}(\{r\})$, which meas that r can be safely substituted by $\mathrm{NORM}(\{r\})$ in contexts respecting the hidden atoms introduced by the normalization.

3 Level Rankings and Ranking Constraints

When a stable model $M \subseteq \mathrm{At}(P)$ of a *non-disjunctive* logic program P is constructed using the reduct P^M and the $\mathbf{T}_{P^M}$ operator, atoms true in the model M get divided into *levels* $M_i = (\mathbf{T}_{P^M} \uparrow^i (I)) \setminus (\mathbf{T}_{P^M} \uparrow^{i-1} (I))$ where $i > 0$ and $I \subseteq \mathrm{At}(P) \setminus \mathrm{H}(P)$ is a set of input atoms. By default $I = \emptyset$ and $M_0 = \emptyset$, but if $I \neq \emptyset$, then $M_0 = I$. For finite programs P, the index i is bounded from above by $|\mathrm{At}(P)|$. Based on this division of atoms, it is possible to read off a *level ranking* $\# : \mathrm{At}(P) \to \mathbb{N} \cup \{\infty\}$ for the atoms of the program [27]: the rank $\#a = i$, if $a \in M_i$, and $\#a = \infty$, if $a \notin M$. A *level numbering* # [17] extends any level ranking for the supporting rules $r \in \mathrm{SuppR}(P,M)$ by the equality[2] $\#r = \max\{\#b \mid b \in \mathrm{B}^+(r)\} + 1$. Intuitively, the level $\#r$ of a rule r indicates when r can be applied to derive its head and, consequently, $\#a = \min\{\#r \mid r \in \mathrm{Def}_P(a) \cap \mathrm{SuppR}(P,M)\}$. By these interconnections, we may use level rankings and numberings interchangeably in the sequel. If $r \notin \mathrm{SuppR}(P,M)$, then $\#r = \infty$. The value ∞ emphasizes that an atom is never derived or a rule becomes never applicable. The other option is to restrict the domain of # to $M \cup \mathrm{SuppR}(P,M)$ for which finite values exist, but some big value greater than any level rank is useful in practice. E.g., given an SCC S of the program P, the level ranks $\#a$ of atoms $a \in S$ can be effectively constrained by $0 < \#a < |S| + 1$; cf. (6) below.

Many existing translations of logic programs into SAT, SMT, and MIP rely on program completion [10]. The idea is to translate a (normal) logic program P into classical equivalences that capture the *supported models* of the program. The purpose of level ranking constraints [27], however, is to distinguish the stable ones among them by incorporating a requirement that there is a level ranking # for a model $M \in \mathrm{SuppM}(P)$. These constraints can be expressed, e.g., as formulas in *difference logic* (DL). This SMT-style logic [28] enriches propositional formulas with difference constraints of the form $x - y \leq k$ where x, y are real/integer variables and k is a constant. The evaluation of a difference atom $x - y \leq k$ is based on an assignment $\tau : \mathscr{V} \to \mathbb{Z}$ on the set of variables $\mathscr{V}$ in use. Given τ, the constraint $x - y \leq k$ is satisfied by τ, denoted by $\tau \models x - y \leq k$, iff $\tau(x) - \tau(y) \leq k$. A *DL-interpretation* is a pair $\langle I, \tau \rangle$ where I a standard propositional interpretation and τ an assignment. A formula ϕ of DL is satisfied by $\langle I, \tau \rangle$, denoted $\langle I, \tau \rangle \models \phi$, if ϕ evaluates to true under I by the usual propositional rules extended by the evaluation of difference atoms subject to τ.

A difference constraint $x_b - x_a \leq -1$ (i.e., $x_a > x_b$) can express that a is derived *after* b under the assumption that x_a and x_b store the level ranks of a and b, respectively. Based on this idea, we introduce formulas for the representation of level ranks. Their *scope* is specified in terms of a set of atoms $S \subseteq \mathrm{At}(P)$ to be discussed in further detail below.

$$(1 \leq x_a \leq |S| + 1), \qquad \neg a \to (x_a \geq |S| + 1), \tag{6}$$

$$\mathrm{dep}(a,b) \quad \leftrightarrow \quad b \wedge (x_a > x_b), \tag{7}$$

$$\mathrm{gap}(a,b) \quad \leftrightarrow \quad b \wedge (x_a > x_b + 1). \tag{8}$$

By the two formulas in (6), level ranks are positive and fixed to $|S| + 1$ if an atom a is false. In addition, we introduce two kinds of new atoms to help with the formulation of the actual level ranking constraints. First, the atom $\mathrm{dep}(a,b)$ defined by (7), denotes an *active* dependency of a head atom a on a positive body atom b, i.e., b must be true. Such dependencies are deployed by Bomanson et al. [6], but we use a definition in terms of the difference constraint. Second, the atom $\mathrm{gap}(a,b)$, as defined by (8), means a similar relationship except that b is derived so early that it is not critical for determining the exact level rank of a. Note that $\mathrm{gap}(a,b)$ implies $\mathrm{dep}(a,b)$ in general but not vice versa. In particular, if $\mathrm{dep}(a,b)$ is true and $\mathrm{gap}(a,b)$ is false, then b must be true and $x_a = x_b + 1$, indicating that a is derived right after b.

---

[2] This holds for rules whose body is essentially normal (1) while generalizations for more complex bodies follow.

Such body atoms b from the preceding level are relevant when a is derived by some rule $r \in \mathrm{Def}_P(a)$ at level x_a.

In the following, we present a reformulation of level ranking constraints [27] by exploiting the dependency relations from (7) and (8). Our further goal is to incorporate the idea of *ordered completion* [3] for the sake of more compact representation. Given an atom $a \in \mathrm{At}(P)$, its completion is based on the set $\mathrm{Def}_P(a) = \{r_1, \ldots, r_k\}$ of its defining rules. In the sequel, the *applicability* of a rule r_i is denoted by a new atom $\mathrm{app}(r_i)$.

$$a \leftrightarrow \mathrm{app}(r_1) \vee \cdots \vee \mathrm{app}(r_k), \tag{9}$$

$$\mathrm{app}(r_i) \leftrightarrow \bigwedge_{b \in \mathrm{B}^+(r_i) \cap S} \mathrm{dep}(a,b) \wedge (\mathrm{B}^+(r_i) \setminus S) \wedge \neg \mathrm{B}^-(r_i) \quad (1 \leq i \leq k), \tag{10}$$

$$\mathrm{app}(r_i) \rightarrow \bigvee_{b \in \mathrm{B}^+(r_i) \cap S} \neg \mathrm{gap}(a,b) \quad (1 \leq i \leq k,\ \mathrm{B}^+(r_i) \cap S \neq \emptyset), \tag{11}$$

$$\mathrm{app}(r_i) \rightarrow (x_a \leq 1) \quad (1 \leq i \leq k,\ \mathrm{B}^+(r_i) \cap S = \emptyset). \tag{12}$$

Intuitively, the equivalence (9) sets the head atom a true if and only if at least one of its defining rules r_i is applicable. This, in turn, is defined by the equivalence (10) insisting that atoms in $\mathrm{B}^+(r_i) \cap S$ have been previously derived and all remaining positive and negative body conditions are satisfied. This formulation embeds both *weak* level ranking constraints [27] and ordered completion [3] but relative to the set S. The constraint (11) is the counterpart of *strong* level ranking constraints [27] enforcing the minimality of level ranks assigned to atoms. Besides this, the formula (12) resets the level of the head atom a to 1 when a can be derived by applying an *externally supporting* rule r_i with $\mathrm{B}^+(r_i) \cap S = \emptyset$.

Regarding the scope S, it is natural to think that the head atom a is included usually. Also, few special cases deserve further attention. (i) If $S = \mathrm{At}(P)$, then the completion becomes fully ordered, i.e., $\mathrm{B}^+(r_i) \setminus S$ becomes empty in (10) and the formula (12) is generated only for $r_i \in \mathrm{Def}_P(a)$ with an empty $\mathrm{B}^+(r_i)$. Moreover, if all atoms of $\mathrm{At}(P)$ are completed using (9)–(12), the resulting formulas capture stable models directly, including level ranks of atoms. (ii) If $S = \mathrm{SCC}(a)$, then the ordering becomes local to the component S. Then, if all atoms of S are completed, the formulas capture stable models M for the *program module* P_S induced by the component S [29]. It should be emphasized that the input atoms in $\mathrm{At}(P_S) \setminus S$ are not subject to completion and they may vary freely. Therefore, given a set of facts $I \subseteq \mathrm{At}(P_S) \setminus S$ as an actual *input* for P_S, the stable models of P_S become solutions to $M = \mathrm{LM}(P_S^M \cup I)$ whose levels i are determined by $\mathbf{T}_{P_S^M} \uparrow^i (I)$. (iii) Finally, if $S = \emptyset$ and $a \notin S$ as an exception, equations (9) and (10) capture the standard completion of a, the formula (11) becomes void, and the formula (12) ensures that $x_a = 1$ whenever a is true.

Example 2. *As a minimal example, consider $a \leftarrow a$ as the only rule r_1 of a program P and the SCC $S = \{a\} = \mathrm{SCC}(a)$. We obtain the following formulas: $(1 \leq x_a \leq 2)$, $\neg a \rightarrow (x_a \geq 2)$, $\mathrm{dep}(a,a) \rightarrow a \wedge (x_a > x_a)$, $\mathrm{gap}(a,a) \rightarrow a \wedge (x_a > x_a + 1)$, $a \leftrightarrow \mathrm{app}(r_1)$, $\mathrm{app}(r_1) \leftrightarrow \mathrm{dep}(a,a)$, $\mathrm{app}(r_1) \rightarrow \neg \mathrm{gap}(a,a)$. They can be satisfied by falsifying a and all new atoms, as well as by setting $x_a = 2$, indicating that $M_1 = \emptyset$ is stable. On the other hand, $M_2 = \{a\}$ is not stable, which can be realized by an attempt to make a true in the formulas listed above. Thus $\mathrm{app}(r_1)$ and $\mathrm{dep}(a,a)$ must be true, too, and $\mathrm{gap}(a,a)$ false. By further inspection of the formulas, it follows that $x_a > x_a$ is true and $x_a > x_a + 1$ is false, both indicating a contradiction.* ■

The case $S = \mathrm{SCC}(a)$ is the most general one and deserves justifications for correctness due to reformulations done in view of [27] and the limitations of ordered completion [3] with regard to (11).

Definition 2. *Given a normal logic program P and a scope $S \subseteq \mathrm{At}(P)$ of completion, the* tight ordered completion *(TOC) of P* relative *to S is the set of formulas* (6) *for $a \in S$,* (7) *and* (8) *for $a, b \in S$ whenever $a \succeq_P b$, and* (9)–(12) *for each $a \in \mathrm{At}(P)$ and $r_i \in \mathrm{Def}_P(a)$.*

The TOC of P relative to S is denoted by $\mathrm{TOC}^S(P)$ and we omit S from the notation $\mathrm{TOC}^S(P)$, if $S = \mathrm{At}(P)$. It is worth noting that the length $\|\mathrm{TOC}^S(P)\|$ stays linear in $\|P_s\|$.

Theorem 1. *Let P be a normal logic program, S an SCC of P, and P_S the module of P induced by S.*

1. *If $M \subseteq \mathrm{At}(P_S)$ is a stable model of P_S for an input $I \subseteq \mathrm{At}(P_S) \setminus S$ and $\# : M \cap S \to \mathbb{N}$ the respective level ranking, then there is a model $\langle N, \tau \rangle$ for $\mathrm{TOC}^S(P)$ such that $M = N \cap \mathrm{At}(P_S)$, $\tau(x_a) = \#a$ for each $a \in M \cap S$, and $\tau(x_a) = |S| + 1$ for each $a \in S \setminus M$.*
2. *If $\langle N, \tau \rangle$ is a model of $\mathrm{TOC}^S(P)$, then $M = N \cap \mathrm{At}(P_S)$ is a stable model of P_S for the input $I = N \cap (\mathrm{At}(P_S) \setminus S)$ and for each $a \in M \cap S$, $\#a = \tau(x_a) - \tau(z)$ is the respective level rank.*

As a preparatory step toward generalizations for aggregated rules, our final example in this section illustrates TOC^S in the context of a cardinality rule (3) that is normalized before completion.

Example 3. *Let us assume that an atom a is defined by a single cardinality rule $a \leftarrow 1 \leq \{b_1, \ldots, b_n\}$ as part of a larger program P having an SCC $S = \mathrm{SCC}(a)$ such that $\{b_1, \ldots, b_n\} \subseteq S$. The rule is compactly expressible even without auxiliary atoms in terms of n positive normal rules*

$$a \leftarrow b_1. \quad \ldots \quad a \leftarrow b_n.$$

The tight ordered completion produces the following formulas for the joint head atom $a \in S$:

$$a \leftrightarrow \mathrm{app}^1(a) \vee \cdots \vee \mathrm{app}^n(a), \tag{13}$$

$$\mathrm{app}^1(a) \leftrightarrow \mathrm{dep}(a, b_1)\ , \ldots,\ \mathrm{app}^n(a) \leftrightarrow \mathrm{dep}(a, b_n), \tag{14}$$

$$\mathrm{app}^1(a) \to \neg\mathrm{gap}(a, b_1)\ , \ldots,\ \mathrm{app}^n(a) \to \neg\mathrm{gap}(a, b_n). \tag{15}$$

In the above, we adopt a convention that $\mathrm{app}^i(a)$ denotes the application of $r_i \in \mathrm{Def}_P(a)$. Since each $b_i \in S$ the respective rules $a \leftarrow b_i$ may not contribute to external support via (12). ■

4 The Case of Monotone Aggregates

Cardinality rules (3) and weight rules (4) with *lower bounds* are widely used examples of monotone aggregates and, in particular, if the (anti-monotone) effect of negative literals is disregarded in the sense of stable models (cf. Definition 1). The level number $\#a$ of an atom $a \in \mathrm{At}(P)$ is generalized in a straightforward way when *positive* cardinality/weight rules are incorporated to the definition of the $\mathbf{T}_P$ operator [30]. As before, $\#a$ is the least value $i \in \mathbb{N}$ such that $a \in \mathbf{T}_P \uparrow^i (\emptyset)$ for positive programs P. Default negation is analogously treated via the reduct, i.e., given a stable model $M \subseteq \mathrm{At}(P)$, the operator $\mathbf{T}_{P^M}$ can be used to assign level ranks for $a \in \mathrm{At}(P)$. The goal of this section is to generalize tight ordered completion for rules involving monotone aggregates. The resulting formulas can be used to enforce stability in various settings where the semantics is no longer based on stable models themselves.

The normalization [7] of cardinality rules is used to guide our intuitions about the intended generalization of tight ordered completion. Besides this, to enable compact representations of aggregates as propositional formulas, we extend the language of difference logic by pseudo-Boolean constraints of the form $c_1a_1 + \cdots + c_ma_m \geq b$ where $a_1, \ldots, a_m$ are atoms, $c_1, \ldots, c_m$ their respective integer coefficients, and b an integer bound. Obviously, given an interpretation $\langle I, \tau \rangle$ in DL, we define $\langle I, \tau \rangle \models c_1a_1 + \cdots + c_ma_m \geq b$, iff $\sum_{I \models a_i} c_i \geq b$, since the truth values of $a_1, \ldots, a_m$ are determined by I independently of τ. Let us begin with an example that concentrates on a corner case ($l = 1$ and $m = 0$) of (3) from Example 1.

Example 4. *Recalling formulas* (13)–(15) *from Example 3, we pull them back to the setting of the original cardinality rule* $a \leftarrow 1 \leq \{b_1,\dots,b_n\}$ *where* $b_1,\dots,b_n$ *depend recursively on the head a. Based on a* connecting *formula* $\mathrm{app}^1(a) \vee \cdots \vee \mathrm{app}^n(a) \leftrightarrow \mathrm{app}(a)$ *on the applicability of the n rules in the normalization versus the applicability of the original rule, we rewrite* (13)–(15) *as follows:*

$$a \leftrightarrow \mathrm{app}(a), \tag{16}$$

$$\mathrm{app}(a) \leftrightarrow (\mathrm{dep}(a,b_1) + \cdots + \mathrm{dep}(a,b_n) \geq 1), \tag{17}$$

$$\mathrm{app}(a) \rightarrow (\mathrm{gap}(a,b_1) + \cdots + \mathrm{gap}(a,b_n) < 1), \tag{18}$$

where the new atoms $\mathrm{dep}(a,b_1),\dots,\mathrm{dep}(a,b_n)$ *and* $\mathrm{gap}(a,b_1),\dots,\mathrm{gap}(a,b_n)$ *are still to be interpreted subject to formulas* (6)–(8) *as in the context of Example 3.* ■

Note that the formula (18) expresses a *dynamic* check, i.e., it works for any subset B of $\{b_1,\dots,b_n\}$ of atoms *true* and *derived earlier* than a. If the cardinality rule is applied (i.e., $|B| \geq 1$), $\mathrm{gap}(a,b_i)$ must be false for each $b_i \in B$, amounting to the effect of the individual implications in (15). The formulas in Examples 3 and 4 are based on different auxiliary atoms denoting the applicability of rules. The connecting formula $\mathrm{app}^1(a) \vee \cdots \vee \mathrm{app}^n(a) \leftrightarrow \mathrm{app}(a)$ describes their intended semantic interconnection for propositional interpretations M and N, i.e., $M \models \mathrm{app}^1(a) \vee \cdots \vee \mathrm{app}^n(a)$ iff $N \models \mathrm{app}(a)$. Because (7) and (8) disconnect all integer variables from the formulas under consideration, the following proposition concentrates on the propositional parts of DL-interpretations that are intended to satisfy TOC formulas in the end.

Proposition 1. *The formulas* (13)–(15) *and* (16)–(18) *constrain the respective interpretations* $\langle M, \tau \rangle$ *and* $\langle N, \tau \rangle$ *equivalently, as conveyed by the satisfaction of the formula* $\mathrm{app}^1(a) \vee \cdots \vee \mathrm{app}^n(a) \leftrightarrow \mathrm{app}(a)$.

Proof. Assuming the connecting formula makes formulas (13) and (16) equivalent. Formulas (13) and (14) imply $a \leftrightarrow \mathrm{dep}(a,b_1) \vee \cdots \vee \mathrm{dep}(a,b_n)$ that is equivalent to (17) under (16). Finally, $\mathrm{gap}(a,b_1) + \cdots + \mathrm{gap}(a,b_n) < 1$ is the same as $\neg\mathrm{gap}(a,b_1) \wedge \cdots \wedge \neg\mathrm{gap}(a,b_n)$. These conditions are equally enforced by (15) and (18) when the connecting formula is satisfied. □

Proposition 1 indicates that the formulas (13)–(15) introduced for the normalizing rules can be safely substituted by the formulas (16)–(18) for the original rule. In this way, the aggregated condition is restored as a subformula in (17) while its negation incarnates in (18). Recall that the truth values of atoms $\mathrm{gap}(a,b_i)$ are determined by (8). If (18) were not satisfied by $\langle N, \tau \rangle$, at least one $\mathrm{gap}(a,b_i)$ atom must be true, i.e., $N \models b_i$ and $\tau \models (x_a > x_{b_i} + 1)$ assuming the satisfaction of (8). Thus b_i would be derived so early that the derivation of a is feasible before and the value of x_a could be decreased. Consequently, the joint effect of the formulas (17) and (18) is that $x_a = \min\{x_{b_i} + 1 \mid N \models b_i\}$ holds which is in harmony with the characterization of [17] when applied to the normalizing rules $a \leftarrow b_1. \dots a \leftarrow b_n$. Before addressing arbitrary cardinality rules, we draw the reader's attention to the other extreme.

Example 5. *When* $l = n$ *and* $m = 0$ *in* (3)*, the rule can be directly cast as a positive normal rule* $a \leftarrow b_1,\dots,b_n$*. Still assuming that* $\{b_1,\dots,b_n\} \subseteq \mathrm{SCC}(a)$*, the TOC formulas resulting from* (9)–(11) *are* $a \leftrightarrow \mathrm{app}^1(a)$, $\mathrm{app}^1(a) \leftrightarrow \mathrm{dep}(a,b_1) \wedge \cdots \wedge \mathrm{dep}(a,b_n)$, *and* $\mathrm{app}^1(a) \rightarrow \bigvee_{1 \leq i \leq n} \neg\mathrm{gap}(a,b_i)$*. The corresponding aggregated formulas can be seen in the formulas* (17) *and* (18) *if the bound* 1 *is substituted by n. The resulting* strong *level ranking constraint ensures that at least one body atom* b_i *is derived* just before *a and* $x_a = \max\{x_{b_i} \mid 1 \leq i \leq n\} + 1$. ■

The preceding example reveals our plan when it comes to covering more general lower bounds $1 < l < n$ in (3) still pertaining to the positive case $m = 0$ and $\{b_1,\dots,b_n\} \subseteq \mathrm{SCC}(a)$. In the sequel, we write $\subseteq_l$ to denote the l-subset relation restricted to subsets of size l *exactly*. Due to monotonicity, the satisfaction of the rule body $l \leq \{b_1, \dots, b_n\}$ essentially depends on the l-subsets of $\{b_1, \dots, b_n\}$. Thus, the cardinality rule (3) with $m = 0$ can be normalized by introducing a positive rule $a \leftarrow B$ for each $B \subseteq_l \{b_1, \dots, b_n\}$. The number of such rules $\binom{n}{l}$ is at maximum when l is n halved.[3] In spite of exponential growth, the resulting normalization serves the purpose of understanding the effect of l on the required TOC formulas. To update equations (13)–(15) for this setting, we need a new atom $\mathrm{app}^B(a)$ for every $B \subseteq_l \{b_1, \dots, b_n\}$ to capture the individual applicabilities of the respective positive rules $a \leftarrow B$:

$$\begin{aligned} a &\leftrightarrow \bigvee_{B \subseteq_l \{b_1,\dots,b_n\}} \mathrm{app}^B(a), & (19)\\ \mathrm{app}^B(a) &\leftrightarrow \bigwedge_{b \in B} \mathrm{dep}(a,b) \;\; (\text{for } B \subseteq_l \{b_1,\dots,b_n\}), & (20)\\ \mathrm{app}^B(a) &\rightarrow \bigvee_{b \in B} \neg\mathrm{gap}(a,b) \;\; (\text{for } B \subseteq_l \{b_1,\dots,b_n\}). & (21) \end{aligned}$$

The connecting formula $\bigvee_{B \subseteq_k \{b_1,\dots,b_n\}} \mathrm{app}^B(a) \leftrightarrow \mathrm{app}(a)$ links the above back to the original rule $a \leftarrow l \leq \{b_1, \dots, b_n\}$ suggesting the revisions of (16)–(18) for any lower bound $1 \leq l \leq n$:

$$\begin{aligned} a &\leftrightarrow \mathrm{app}(a), & (22)\\ \mathrm{app}(a) &\leftrightarrow (\mathrm{dep}(a,b_1) + \cdots + \mathrm{dep}(a,b_n) \geq l), & (23)\\ \mathrm{app}(a) &\rightarrow (\mathrm{gap}(a,b_1) + \cdots + \mathrm{gap}(a,b_n) < l). & (24) \end{aligned}$$

Most importantly, the length of the formulas (22)–(24) stays linear in n in contrast with their alternatives (19)–(21) based on l-subsets. The aggregate-based formulation covers all l-subsets of $\{b_1, \dots, b_n\}$ and their supersets that also satisfy the body of (3) with $m = 0$ by monotonicity.

Proposition 2. *The formulas* (19)–(21) *and* (16)–(24) *constrain the respective interpretations* $\langle M, \tau\rangle$ *and* $\langle N, \tau\rangle$ *equivalently, as conveyed by the satisfaction of the formula* $\bigvee_{B \subseteq_l \{b_1,\dots,b_n\}} \mathrm{app}^B(a) \leftrightarrow \mathrm{app}(a)$.

The proof is similar to that of Proposition 1 and amounts to showing that the (big) disjunctive formula $\bigvee_{B \subseteq_l \{b_1,\dots,b_n\}} \bigwedge_{b \in B} \mathrm{dep}(a,b)$ is expressible as $(\mathrm{dep}(a,b_1) + \cdots + \mathrm{dep}(a,b_n) \geq l)$. Similar aggregation is achieved in (24) in terms of $\mathrm{gap}(a,b_1), \dots, \mathrm{gap}(a,b_1)$. An important observation is that $\langle N, \tau\rangle \not\models$ (24) if and only if $N \models \mathrm{app}(a)$ and $\exists B \subseteq_l \{b_1, \dots, b_n\}$ such that for each $b \in B$, $N \models b$, $N \models \mathrm{gap}(a,b)$, and $\tau \models (x_a > x_b + 1)$. Since B reaches the bound l the value of x_a could be decreased to $\max\{\tau(x_b) \mid b \in B\} + 1$ or even further if $|B| > l$. Thus the satisfaction of (24) means that no $B \subseteq_l (\{b_1, \dots, b_n\} \cap N)$ of true atoms could be used to decrease the value of x_a. The net effect is that x_a has the critical minimum value. Since (23) is also satisfied there are at least l true atoms derived before a but sufficiently many of them are derived *just before a*. For those atoms b we have $x_a = x_b + 1$, $\mathrm{dep}(a,b)$ true, but $\mathrm{gap}(a,b)$ false!

Example 6. *Consider the rule* $a \leftarrow 2 \leq \{b_1, b_2, b_3, b_4\}$ *in the context of a model* $\langle N, \tau\rangle$ *where* $N = \{a, b_1, b_3, b_4, \mathrm{app}(a)\}$, $\tau(x_{b_1}) = \tau(x_{b_4}) = 2$, $\tau(x_{b_3}) = 1$, *and* $\tau(x_{b_2}) = 6$ *by default as* $N \not\models b_2$, *see* (6). *Now the rule body is satisfied by three* 2*-subsets* $B_1 = \{b_1, b_3\}$, $B_2 = \{b_1, b_4\}$, *and* $B_3 = \{b_3, b_4\}$ *justifying the level rank* $\tau(x_a) = 3$, *since* $\max\{\tau(x_b) \mid b \in B_i\} = 2$ *for each* $1 \leq i \leq 3$. *We have* $\mathrm{gap}(b_i, a)$ *true only for* $i = 3$ *and thus* (24) *is respected for* $l = 2$. *But, if* $\tau(x_a) = 4$ *had been alternatively set, the count of such* b_i*'s would be* 3, *falsifying* (24). ■

---

[3] Note that $\binom{n}{\lfloor n/2 \rfloor} \geq 3^{\lfloor n/2 \rfloor}$ from $n > 4$ onward.

4.1 Weights

So far, we have established relatively general form of TOC formulas (22)–(24) that cover cardinality rules (3) when $m = 0$. Before addressing negative body conditions ($m > 0$) and settings where SCCs play a major role, we take the weights of literals into consideration as already present in weight rules (4) when $m = 0$. Consequently, we have to substitute l-subsets used in (19)–(21) by *weighted* subsets $B \subseteq_w \{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}\}$. Such a subset B can be formally defined in terms of the condition $\mathrm{WS}_B(\{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}\}) \geq w$ from Section 2. It is clear by monotonicity that if $B \subseteq_w \{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}\}$, then $B' \subseteq_w \{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}\}$ for every B' with $B \subseteq B' \subseteq \{b_1, \ldots, b_n\}$. A weighted set $B \subseteq_w \{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}\}$ is defined to be $\subseteq$-minimal with respect to w, if for no $B' \subset B$, $B' \subseteq_w \{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}\}$. We use $\subseteq_w^{\min}$ to indicate such $\subseteq$-minimal weighted subsets of $\{b_1, \ldots, b_n\}$. Assuming orthogonal generalizations of (19)–(21) for a *positive* weight rule (4) and the weighted subsets $B \subseteq_w^{\min} \{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}\}$ of its body, we rather incorporate weights into the formulas (22)–(24) as follows:

$$a \leftrightarrow \mathrm{app}(a), \tag{25}$$

$$\mathrm{app}(a) \leftrightarrow (w_{b_1} \times \mathrm{dep}(a, b_1) + \cdots + w_{b_n} \times \mathrm{dep}(a, b_n) \geq w), \tag{26}$$

$$\mathrm{app}(a) \rightarrow (w_{b_1} \times \mathrm{gap}(a, b_1) + \cdots + w_{b_n} \times \mathrm{gap}(a, b_n) < w). \tag{27}$$

Proposition 3. *The formulas* (19)–(21) *revised for weighted subsets B subject to the bound w and the formulas* (25)–(27) *constrain the respective interpretations* $\langle M, \tau \rangle$ *and* $\langle N, \tau \rangle$ *equivalently, as conveyed by the satisfaction of the equivalence* $\bigvee_{B \subseteq_w^{\min} \{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}\}} \mathrm{app}^B(a) \leftrightarrow \mathrm{app}(a)$.

Proof. Due to high similarity with respect to Proposition 2, we just point out the equivalence of the formulas $\bigvee_{B \subseteq_w^{\min} \{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}\}} \bigwedge_{b \in B} \mathrm{dep}(a, b)$ and $(w_{b_1} \times \mathrm{dep}(a, b_1) + \cdots + w_{b_n} \times \mathrm{dep}(a, b_n) \geq w)$. The equivalence involving $\mathrm{gap}(a, b_1), \ldots, \mathrm{gap}(a, b_n)$ is analogous but negated. ☐

Again, $\langle N, \tau \rangle \not\models$ (27) implies $N \models \mathrm{app}(a)$ and for some $B \subseteq_w^{\min} \{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}\}$, $N \models B$ and for every $b \in B$, $\tau \models (x_a > x_b + 1)$. Then, the value of x_a could be decreased to $\max\{\tau(x_b) \mid b \in B\} + 1$. Thus the formula (27) makes $\tau(x_a)$ minimal as before.

Example 7. *Let us consider a positive weight rule* $a \leftarrow 7 \leq \{b_1 = 7, b_2 = 5, b_3 = 3, b_4 = 2, b_5 = 1\}$ *in the context of a model* $\langle N, \tau \rangle$ *where N sets all body atoms* $b_1, \ldots, b_5$ *true and* τ *the level numbers*

$$\tau(x_a) = 5, \quad \tau(x_{b_1}) = 5, \quad \tau(x_{b_2}) = 4, \quad \tau(x_{b_3}) = 3, \quad \tau(x_{b_4}) = 2, \quad \tau(x_{b_5}) = 1.$$

The $\subseteq$*-minimal satisfiers of the body are* $B_1 = \{b_1\}$, $B_2 = \{b_2, b_3\}$, *and* $B_3 = \{b_2, b_4\}$. *The only atom in* B_1 *has a weight that reaches the bound* 7 *alone, but it is derived too late to affect the derivation of a. Both* B_2 *and* B_3 *yield the same value* $\max\{\tau(x_b) \mid b \in B_i\} = 4$ *and hence justify the one higher value* 5 *assigned to* x_a. *Interestingly, there is also an atom* b_5 *that is derived early, but whose weight is irrelevant for satisfying the rule body nor deriving a any earlier. In fact, this weighted atom could be safely deleted from the rule (under strong equivalence).*

As regards the satisfaction of (27), *the relevant body atoms are* b_3, b_4, *and* b_5, *for which the atom* $\mathrm{gap}(a, b_i)$ *is made true by* (8). *The sum of the respective weights* $3 + 2 + 1$ *is less than* 7.

Also, note that the level numbers assigned by τ *to* $b_1, \ldots, b_5$ *can be easily arranged with positive rules, e.g., by using the chain of rules:* $b_1 \leftarrow b_2.$ $b_2 \leftarrow b_3.$ $b_3 \leftarrow b_4.$ $b_4 \leftarrow b_5.$ $b_5.$ *Given the respective program P, the operator* $\mathbf{T}_P$ *should be applied* 5 *times to make a true.* ■

4.2 Negative Conditions

Negative body conditions form the missing pieces when it comes to fully covering WCPs with level ranking constraints as embedded in tight ordered completion. To this end, our strategy is based on rewriting and ideas used in [7] where the correctness of normalization is first shown for positive programs and then generalized for programs with negation. In a nutshell, negative literals in (4) can be replaced by new atoms $\overline{c_1},\ldots,\overline{c_m}$ that respectively denote that $c_1,\ldots,c_m$ cannot be derived. These atoms are subsequently defined by (atomic) normal rules $\overline{c_1} \leftarrow {\sim}c_1.\quad \ldots\quad \overline{c_m} \leftarrow {\sim}c_m$. The outcome is a set of rules that is visibly strongly equivalent with the original weight rule (4). The completions of $\overline{c_1},\ldots,\overline{c_m}$ are $\overline{c_1} \leftrightarrow \neg c_1,\ldots,\overline{c_m} \leftrightarrow \neg c_m$, enabling the substitution of $\overline{c_1},\ldots,\overline{c_m}$ by $\neg c_1,\ldots,\neg c_m$ in any formulas of interest. In this way, $\overline{c_1},\ldots,\overline{c_m}$ can be readily forgotten under classical semantics. The transformation described above leaves the SCCs of the program intact, because positive dependencies are not affected. Thus, besides taking care of negative body conditions, our next rewriting step recalls the scope $S \subseteq \mathrm{At}(P)$ from (10)–(12): we present TOC formulas to cover WCPs split into modules based on SCCs. We say that a WCP is *pure* if it contains weight rules (4) only.

Definition 3. *Let P be a pure WCP and $S \subseteq \mathrm{At}(P)$ an SCC of P used as the scope of completion. The* tight ordered completion *of P relative to S, denoted $\mathrm{TOC}^S(P)$, consists of the formulas listed below:*

- *If $|S| > 1$, then for each $a \in S$:*
  1. *the formulas* (6)*;*
  2. *the formulas* (7) *and* (8) *for each $b \in S$ such that $a \succeq_P b$; plus*
  3. *the following formulas based on the definition $\mathrm{Def}_P(a) = \{r_1,\ldots,r_k\}$ in the program P:*

$$
\begin{aligned}
a &\leftrightarrow \mathrm{app}^1(a) \vee \cdots \vee \mathrm{app}^k(a), && (28)\\
\mathrm{app}^i(a) &\leftrightarrow \mathrm{int}^i(a) \vee \mathrm{ext}^i(a), && (29)\\
\mathrm{int}^i(a) &\leftrightarrow (\textstyle\sum_{b\in \mathrm{B}^+(r_i)\cap S}(w_b \times \mathrm{dep}(a,b)) + \sum_{b\in \mathrm{B}^+(r_i)\setminus S}(w_b \times b) &&\\
&\qquad - \textstyle\sum_{c\in \mathrm{B}^-(r_i)}(w_c \times c) \geq w_{r_i} - w_i), && (30)\\
\mathrm{int}^i(a) &\rightarrow (\textstyle\sum_{b\in \mathrm{B}^+(r_i)\cap S}(w_b \times \mathrm{gap}(a,b)) + \sum_{b\in \mathrm{B}^+(r_i)\setminus S}(w_b \times b) &&\\
&\qquad - \textstyle\sum_{c\in \mathrm{B}^-(r_i)}(w_c \times c) < w_{r_i} - w_i) \vee \mathrm{ext}^i(a), && (31)\\
\mathrm{ext}^i(a) &\leftrightarrow (\textstyle\sum_{b\in \mathrm{B}^+(r_i)\setminus S}(w_b \times b) - \sum_{c\in \mathrm{B}^-(r_i)}(w_c \times c) &&\\
&\qquad \geq w_{r_i} - w_i), && (32)\\
\mathrm{ext}^i(a) &\rightarrow (x_a \leq 1), && (33)
\end{aligned}
$$

  where $w_i = \sum_{c\in \mathrm{B}^-(r_i)} w_c$ is the adjustment to the bound w_{r_i} of r_i.

- *If $|S| = 1$ and $S = \{a\} = \mathrm{SCC}(a)$, then* (28)–(32) *are replaced by the standard completion*

$$
\mathrm{app}^i(a) \leftrightarrow (\sum_{b\in \mathrm{B}^+(r_i)}(w_b \times b) - \sum_{c\in \mathrm{B}^-(r_i)}(w_c \times c) \geq w_{r_i} - w_i). \tag{34}
$$

In the equations of Definition 3, the treatment of negative literals occurring in a defining rule r_i is justified by their contribution $\sum_{c\in \mathrm{B}^+(r_i)}(w_c \times (1-c)) = w_i - \sum_{c\in \mathrm{B}^+(r_i)}(w_c \times c)$ toward the bound w_{r_i}. Weight rules can also create external support in more flexible ways, i.e., if the bound w_{r_i} can be reached by satisfying any positive body conditions outside the SCC S in question or any negative body conditions. Moreover, a single weight rule r_i may justify the head a either internally or externally as formalized by (29), which is different from the case of normal rules. Formulas (30) and (32) capture this distinction.

Note that $\mathrm{ext}^i(a)$ implies $\mathrm{int}^i(a)$. The constraint (31) generalizes (11) while (33) is the analog of (12). The consequent of (31) is weakened by the condition $\mathrm{ext}^i(a)$, since the other disjunct is falsified if r_i provides external support, $x_a = 1$ holds, and all atoms $\mathrm{dep}(a,\cdot)$ and $\mathrm{gap}(a,\cdot)$ associated with a are falsified. The reader may have noticed that the formulas concerning $\mathrm{int}^i(a)$ and $\mathrm{ext}^i(a)$ share a potentially large subexpression $s_i = \sum_{b \in \mathrm{B}^+(r_i)\setminus S}(w_b \times b) - \sum_{c \in \mathrm{B}^-(r_i)}(w_c \times c)$. Certain back-end formalisms, such as DL and MIP, enable the representation of this expression only once using an integer variable.

The following theorem states the correctness of $\mathrm{TOC}(P)$ obtained as the union of $\mathrm{TOC}^S(P)$ for the SCCs S of P. The claimed one-to-one correspondence, however, must take into account the fact that a satisfying assignment $\langle M, \tau\rangle$ in DL can be replicated into infinitely many copies by substituting τ in $\langle M, \tau\rangle$ by a function $\tau'(x) = \tau(x) + k$ for any $k \in \mathbb{Z}$. The remedy is to introduce a special variable z which is assumed to hold 0 as its value and the values of the other variables are set relative to the value of z. The current difference constraints mentioning only one variable must be rewritten using z. For instance, $1 \leq x_a \leq |S| + 1$ in (6) is expressed by the conjunction of $z - x_a \leq -1$ and $x_a - z \leq |S| + 1$, and this is how the variable z gets introduced.

Theorem 2. *Let P be a WCP with SCCs $S_1, \ldots, S_q$ and $P_{S_1}, \ldots, P_{S_q}$ the respective modules of P. Then P and the set of formulas $F = \bigcup_{j=1}^{q} \mathrm{TOC}^{S_j}(P)$ are visibly equivalent (up to assigning $z = 0$).*

5 Generalizations Toward Convex Aggregates

Weight rules (4) can be generalized by introducing upper bounds u besides lower bounds l:

$$a \leftarrow l \leq \{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}, {\sim}c_1 = w_{c_1}, \ldots, {\sim}c_m = w_{c_m}\} \leq u. \tag{35}$$

This gives a rise to a *convex* condition which is easier to explain for positive rules ($m = 0$). If the condition can be satisfied by setting the atoms of $B_1 \subseteq \{b_1, \ldots, b_n\}$ true and the same holds for a superset $B_2 \subseteq \{b_1, \ldots, b_n\}$ of B_1, then every intermediate set B' such that $B_1 \subseteq B' \subseteq B_2$ satisfies the condition, too. It is easy to check that this is a property of (35), since $B_1 \subseteq B_2$ implies $\mathrm{WS}_{B_1}(\{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}\}) \leq \mathrm{WS}_{B_2}(\{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}\})$ in general. However, the bounds play a role here: upper bounds jeopardize monotonicity in general but convexity is still guaranteed. Thus, we use WCPs based on (35) to understand the role of level ranking constraints in the context of convex aggregates. The effect of negative literals is anti-monotonic, but their semantics is determined by the reduct as usual. Simons et al. [30] present a transformation that checks the upper bound of (35) with another weight rule. The rules below adopt this idea but using a constraint and new atoms that are in line with (34) and $r_i \in \mathrm{Def}_P(a)$:

$$\mathrm{app}^i(a) \leftarrow l \leq \{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}, {\sim}c_1 = w_{c_1}, \ldots, {\sim}c_m = w_{c_m}\}. \tag{36}$$

$$\mathrm{vub}^i(a) \leftarrow u + 1 \leq \{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}, {\sim}c_1 = w_{c_1}, \ldots, {\sim}c_m = w_{c_m}\}. \tag{37}$$

$$\leftarrow \mathrm{app}^i(a), \mathrm{vub}^i(a). \tag{38}$$

For the moment, this relaxes the notion of applicability for r_i but the constraint (38) makes sure that the upper bound is not *violated*. Since no atom depends positively on $\mathrm{vub}^i(a)$, the resulting TOC formula is analogous to (34) with $\mathrm{vub}^i(a)$ as its head. The constraint (38) can be expressed as $\neg(\mathrm{app}^i(a) \land \mathrm{vub}^i(a))$. The net effect is that if $\mathrm{app}^i(a)$ is true, then $(\sum_{b \in \mathrm{B}^+(r_i)}(w_b \times b) - \sum_{c \in \mathrm{B}^-(r_i)}(w_c \times c) \leq u - w_i)$ must hold. Thus (30)–(32) can be revised for (35) by replacing the previous lower bound w_{r_i} with l and by incorporating $u - w_i$ into (30) and (32) as upper bounds; either as two pseudo-Boolean constraints or a combined one with two bounds. The respective upper bound does not play a role in (31) that concerns the criticality of the lower bound and this check does not interfere with the satisfaction of the upper bound due to convexity.

5.1 Abstract Constraint Atoms

Based on the preceding analysis of weight rules, we will rephrase our approach for an arbitrary convex aggregate $\mathrm{Aggr}(B)$ that takes a set of (body) atoms B as input and accepts a certain subset $\mathscr{S}$ of the powerset $\mathbf{2}^B$ by evaluating to true. This set must satisfy the convexity condition, i.e., if $S_1, S_2 \in \mathscr{S}$ then $S \in \mathscr{S}$ for each intermediate set $S_1 \subseteq S \subseteq S_2$, too. Moreover, we let $\mathrm{Aggr}^*(B)$ stand for the *monotonic* (upward) *closure* of $\mathrm{Aggr}(B)$ based on the signature B. The set of satisfiers $\mathscr{S} \subseteq \mathbf{2}^B$ of the latter is extended to a set of satisfiers $\mathscr{S}^* = \{S \mid \exists S' \in \mathscr{S},\ S' \subseteq S \subseteq B\}$ for the former. The syntax of logic programs can be extended by introducing aggregated conditions as rule bodies in analogy to (3) and (4):

$$a \leftarrow \mathrm{Aggr}(b_1, \ldots, b_n, {\sim}c_1, \ldots, {\sim}c_m). \tag{39}$$

Let P be a logic program consisting of rules of the form (39) and $M \subseteq \mathrm{At}(P)$ a *model* of P that satisfies all rules (39) in the standard sense, i.e., if the body is satisfied by M, then the head $a \in M$. The reduct P^M can be formed by including a positive rule $a \leftarrow \mathrm{Aggr}^*(b_1, \ldots, b_m, c_1 \in M\,?\,\bot : \top, \ldots, c_m \in M\,?\,\bot : \top)$ for each (39) such that $M \models \mathrm{Aggr}(b_1, \ldots, b_n, {\sim}c_1, \ldots, {\sim}c_m)$. In the above, we exploit *conditional substitutions* $c\,?\,v : u$ by the value v if the condition c is true and the value u otherwise. Thus, given a set of input atoms $I \subseteq \mathrm{At}(P) \setminus \mathrm{H}(P)$, we can calculate the least model of P^M and assign level ranks $\#a = i$ of atoms $a \in \mathrm{H}(P)$ based on the membership $a \in \mathbf{T}_{P^M} \uparrow^i (I) \setminus \mathbf{T}_{P^M} \uparrow^{i-1} (I)$ as earlier. Assuming a defining rule $r_i \in \mathrm{Def}_P(a)$ for an atom $a \in \mathrm{H}(P)$ and the scope $S = \mathrm{SCC}(a)$, we generalize (30)–(32) as follows:

$$\begin{aligned}
\mathrm{int}^i(a) &\leftrightarrow \mathrm{Aggr}(b_1 \in S\,?\,\mathrm{dep}(a,b_1) : b_1, \ldots, b_n \in S\,?\,\mathrm{dep}(a,b_n) : b_n, \neg c_1, \ldots, \neg c_m), && (40)\\
\mathrm{int}^i(a) &\rightarrow \neg\mathrm{Aggr}^*(b_1 \in S\,?\,\mathrm{gap}(a,b_1) : b_1, \ldots, b_n \in S\,?\,\mathrm{gap}(a,b_n) : b_n, \neg c_1, \ldots, \neg c_m) &&\\
&\qquad \vee \mathrm{ext}^i(a), && (41)\\
\mathrm{ext}^i(a) &\leftrightarrow \mathrm{Aggr}(b_1 \in S\,?\,\bot : b_1, \ldots, b_n \in S\,?\,\bot : b_n, \neg c_1, \ldots, \neg c_m). && (42)
\end{aligned}$$

The formulas above assume that the aggregate $\mathrm{Aggr}(\cdot)$ can be expressed as a propositional formula, or the target language has sufficient syntactic primitives available such as pseudo-Boolean constraints. Conditional substitutions are necessary, because we have not made any assumptions about ordering the arguments to $\mathrm{Aggr}(\cdot)$. For instance, if the weight constraint from Example 7 were abstracted as $\mathrm{Aggr}(b_1, b_2, b_3, b_4, b_5)$, the meaning of $\mathrm{Aggr}(b_5, b_4, b_3, b_2, b_1)$ would become different due to asymmetric weights. As in the case of pure WCPs, the TOC formula (40) takes care of the (potential) internal support delivered by r_i and the support is ordered by the conjoined atoms $\mathrm{dep}(a, b_i)$ so that the stability of models can be guaranteed. The formula (42) for external support is analogous except that the contribution of positive body conditions from the same component is hindered by substituting them by $\bot$. By the side of these formulas, the strong ranking constraint (41) ensures that $\mathrm{Aggr}(\cdot)$ could not be satisfied with arguments derived way before a.

6 Conclusions

In this work, we investigate potential generalizations of *level ranking constraints* [27] for extended rule types involving aggregates. It turns out that the structure of aggregates can be preserved if the back-end language offers analogous syntactic primitives for expressing such conditions. As a consequence, it is not necessary to normalize rules before the introduction of ranking constraints in a form or another, depending on the target formalism and the back-end solver to be used for actual computations. The particular novelty of our approach lies in the generalization of *strong* level ranking constraints for WCPs and other monotone/convex aggregates. Using them level rankings can be made unique which is desirable, e.g., when counting answer sets. A further by-product is that any WCP can be translated into a *tight* WCP if the formulas in $\mathrm{TOC}(P)$ are expressed with rules rather than formulas, also justifying "*tight*" as part of TOC.

Although the results of this article are theoretical by nature, they enable new kinds of strategies when it comes to implementing the search of stable models using existing solver technology for SAT, SMT, and MIP. E.g., the presented TOC formulas offer a common ground for the translators in the LP2* family [19]. We leave *non-convex* aggregates as future work for two main reasons. First, there is no consensus about their semantics when recursive definitions are enabled [2]. The ASP-core-2 language standard assumes the *stratified* setting only whereas the Clingo system implements one particular semantics for recursive non-convex aggregates [13]. Second, there is also evidence [1] that the removal of non-convex aggregates tends to produce disjunctive rules which go beyond level rankings in the first place. One potential solution is provided by the *decomposition* of non-convex aggregates into their maximal convex regions, cf. [18, 24].

Acknowledgments This research is partially supported by the Academy of Finland projects AI-ROT (#335718), XAILOG (#345633), and ETAIROS (#352441).

References

[1] Mario Alviano, Wolfgang Faber & Martin Gebser (2015): *Rewriting recursive aggregates in answer set programming: back to monotonicity*. Theory Pract. Log. Program. 15(4-5), pp. 559–573, doi:10.1017/S1471068415000228.

[2] Mario Alviano, Wolfgang Faber & Martin Gebser (2023): *Aggregate Semantics for Propositional Answer Set Programs*. Theory Pract. Log. Program. 23(1), pp. 157–194, doi:10.1017/S1471068422000047.

[3] Vernon Asuncion, Yin Chen, Yan Zhang & Yi Zhou (2015): *Ordered completion for logic programs with aggregates*. Artif. Intell. 224, pp. 72–102, doi:10.1016/j.artint.2015.03.007.

[4] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia & Cesare Tinelli (2021): *Satisfiability Modulo Theories*. In: Handbook of Satisfiability – 2nd Ed., IOS Press, pp. 1267–1329, doi:10.3233/FAIA201017.

[5] Armin Biere, Marijn Heule, Hans van Maaren & Toby Walsh (2021): *Handbook of Satisfiability – 2nd Ed.* FAIA 336, IOS Press, doi:10.3233/FAIA336.

[6] Jori Bomanson, Martin Gebser, Tomi Janhunen, Benjamin Kaufmann & Torsten Schaub (2016): *Answer Set Programming Modulo Acyclicity*. Fundam. Informaticae 147(1), pp. 63–91, doi:10.3233/FI-2016-1398.

[7] Jori Bomanson, Tomi Janhunen & Ilkka Niemelä (2020): *Applying Visible Strong Equivalence in Answer-Set Program Transformations*. ACM Trans. Comput. Log. 21(4), pp. 33:1–33:41, doi:10.1145/3412854.

[8] Gerhard Brewka, Thomas Eiter & Mirek Truszczynski (2011): *Answer set programming at a glance*. Communications of the ACM 54(12), pp. 92–103, doi:10.1145/2043174.2043195.

[9] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca & Torsten Schaub (2012): ASP-CORE-2 *Input Language Format*. Available at `https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03b.pdf`.

[10] Keith Clark (1978): *Negation as Failure*. In: Logic and Data Bases, Plenum Press, pp. 293–322, doi:10.1007/978-1-4684-3384-5_11.

[11] Francois Fages (1994): *Consistency of Clark's completion and the existence of stable models*. Journal of Methods of Logic in Computer Science 1, pp. 51–60.

[12] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Sht. Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens & Luc De Raedt (2015): *Inference and learning in probabilistic logic*

programs using weighted Boolean formulas. Theory Pract. Log. Program. 15(3), pp. 358–401, doi:10.1017/S1471068414000076.

[13] Martin Gebser, Roland Kaminski & Torsten Schaub (2016): *Grounding Recursive Aggregates: Preliminary Report*. CoRR abs/1603.03884, p. 21 pages, doi:10.48550/arXiv.1603.03884.

[14] Michael Gelfond & Vladimir Lifschitz (1988): *The Stable Model Semantics for Logic Programming*. In: Proceedings of the 6th International Conference on Logic Programming (ICLP'88), MIT Press, pp. 1070–1080.

[15] Michael Gelfond & Vladimir Lifschitz (1991): *Classical Negation in Logic Programs and Disjunctive Databases*. New Generation Computing 9(3/4), pp. 365–385, doi:10.1007/BF03037169.

[16] Jinbo Huang (2008): *Universal Booleanization of Constraint Models*. In: CP 2008, Springer, pp. 144–158, doi:10.1007/978-3-540-85958-1_10.

[17] Tomi Janhunen (2006): *Some (In)translatability Results for Normal Logic Programs and Propositional Theories*. Journal of Applied Non-Classical Logics 16(1–2), pp. 35–86, doi:10.3166/jancl.16.35-86.

[18] Tomi Janhunen (2010): *Sampler Programs: The Stable Model Semantics of Abstract Constraint Programs Revisited*. In: ICLP 2010, LIPIcs 7, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 94–103, doi:10.4230/LIPIcs.ICLP.2010.94.

[19] Tomi Janhunen (2018): *Cross-Translating Answer Set Programs Using the ASPTOOLS Collection*. Künstliche Intell. 32(2-3), pp. 183–184, doi:10.1007/s13218-018-0529-9.

[20] Tomi Janhunen, Ilkka Niemelä & Mark Sevalnev (2009): *Computing Stable Models via Reductions to Difference Logic*. In: LPNMR 2009, Springer, pp. 142–154, doi:10.1007/978-3-642-04238-6_14.

[21] Vladimir Lifschitz, David Pearce & Augustin Valverde (2001): *Strongly Equivalent Logic Programs*. ACM Transactions on Computational Logic 2(4), pp. 526–541, doi:10.1145/383779.383783.

[22] Vladimir Lifschitz & Alexander Razborov (2006): *Why are there so many loop formulas?* ACM Trans. Comput. Log. 7(2), pp. 261–268, doi:10.1145/1131313.1131316.

[23] Fangzhen Lin & Yuting Zhao (2004): *ASSAT: computing answer sets of a logic program by SAT solvers*. Artif. Intell. 157(1-2), pp. 115–137, doi:10.1016/j.artint.2004.04.004.

[24] Guohua Liu, Randy Goebel, Tomi Janhunen, Ilkka Niemelä & Jia-Huai You (2011): *Strong Equivalence of Logic Programs with Abstract Constraint Atoms*. In: LPNMR 2011, Springer, pp. 161–173, doi:10.1007/978-3-642-20895-9_15.

[25] Guohua Liu, Tomi Janhunen & Ilkka Niemelä (2012): *Answer Set Programming via Mixed Integer Programming*. In: KR 2012, AAAI Press, pp. 32–42. Available at `http://www.aaai.org/ocs/index.php/KR/KR12/paper/view/4516`.

[26] Mai Nguyen, Tomi Janhunen & Ilkka Niemelä (2011): *Translating Answer-Set Programs into Bit-Vector Logic*. In: INAP 2011, Springer, pp. 95–113, doi:10.1007/978-3-642-41524-1_6.

[27] Ilkka Niemelä (2008): *Stable models and difference logic*. Ann. Math. Artif. Intell. 53(1-4), pp. 313–329, doi:10.1007/s10472-009-9118-9.

[28] Robert Nieuwenhuis, Albert Oliveras & Cesare Tinelli (2006): *Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)*. J. ACM 53(6), pp. 937–977, doi:10.1145/1217856.1217859.

[29] Emilia Oikarinen & Tomi Janhunen (2008): *Achieving compositionality of the stable model semantics for smodels programs*. Theory Pract. Log. Program. 8(5-6), pp. 717–761, doi:10.1017/S147106840800358X.

[30] Patrik Simons, Ilkka Niemelä & Timo Soininen (2002): *Extending and Implementing the Stable Model Semantics*. Artificial Intelligence 138(1–2), pp. 181–234, doi:10.1016/S0004-3702(02)00187-X.

[31] Hudson Turner (2003): *Strong equivalence made easy: nested expressions and weight constraints*. Theory Pract. Log. Program. 3(4-5), pp. 609–622, doi:10.1017/S1471068403001819.

[32] Laurence A. Wolsey (2008): *Mixed Integer Programming*. John Wiley & Sons, Inc., doi:10.1002/9780470050118.ecse244.

Sorting Strategies for Interactive Conflict Resolution in ASP

Andre Thevapalan
Technische Universität Dortmund, Dortmund, Germany
andre.thevapalan@tu-dortmund.de

Gabriele Kern-Isberner
Technische Universität Dortmund, Dortmund, Germany
gabriele.kern-isberner@cs.tu-dortmund.de

Answer set programs in practice are often subject to change. This can lead to inconsistencies in the modified program due to conflicts between rules which are the results of the derivation of strongly complementary literals. To facilitate the maintenance of consistency in answer set programs, in this paper we continue work on a recently presented framework that implements interactive conflict resolution by extending the bodies of conflicting rules by suitable literals, so-called λ-*extensions*. More precisely, we present strategies to choose λ-extensions that allow for resolving several conflicts at a time in an order that aims at minimizing (cognitive) efforts. In particular, we present a graphical representation of connections between conflicts and their possible solutions. Such a representation can be utilized to efficiently guide the user through the conflict resolution process by displaying conflicts and suggesting solutions in a suitable order.

1 Introduction

Answer set programming provides valuable features for usage in real-world applications where complex decisions have to be made, thanks to its declarative nature and the availability of both strong negation and default negation. Such programs, though, are often subject to change. Adding rules to an answer set program can however potentially lead to inconsistency due the activation of conflicting rules, i. e., rules with complementary literals in their heads.The approach in [9] deals with inconsistency caused by the derivation of strongly complementary literals. Herefore, the notion of λ-extensions for conflicting rules has been introduced that enables an interactive conflict resolution process in which a knowledge expert can help restore the consistency of an updated program in a professionally adequate way. A λ-extension is a set of (default) literals with which the body of a conflicting rule can be extended to constrain its applicability and thus resolves the conflict. However, the paper [9] focuses on conflicts between two rules only. But conflicts can involve more than two rules, and solutions to one conflict can affect solutions to other conflicts. It is clear that no new conflicts may arise by extending rule bodies. So we might expect even synergetic positive effects when considering all interactions between conflicts in a program at the same time and finding a clever order to solve conflicts. This is exactly the topic here.

In this work, we extend the approach of [9] by first defining a graph that shows the connections between the conflicts and their solutions, thereby embedding the conflicts of a program and their possible solutions within the overall context of the program's conflict resolution. This graphical representation of possible solutions is then utilized to define suitable orders over conflicts and the respective λ-extensions so that one solution can help solving subsequent other conflicts. In particular, the strategies presented in this paper can also help shrink the (sometimes large) sets of possible λ-extensions by choosing such extensions that are involved in more than one conflict. In this way, also the cognitive burden for knowledge experts during the conflict resolution process can be reduced.

The main contributions of this paper can be summarized as follows:

- We introduce a suitable graph structure named λ-graphs for λ-extensions to display the relationships between conflicts w.r.t. their possible solutions.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 116–128, doi:10.4204/EPTCS.385.13

- We show how λ-graphs provide the necessary, syntax-based information to find suitable orders over conflicts.
- Utilizing λ-graphs, we furthermore explain how one can obtain an order over the possible solutions of a conflict.
- Based on these results, we present an explicit sorting strategy that defines an order over conflicts and their solutions for the application in a conflict resolution framework as proposed in [8, 9].

This paper begins by laying out the necessary preliminaries in Section 2. Section 3 provides further terminology regarding conflict resolution which is then used in Section 4 to construct λ-graphs that provide crucial information regarding the connections between conflicts and λ-extensions. Based on these results in Section 5.1, a sorting strategy is proposed that defines an order over conflicts and their possible solutions by taking their relationships among each other into account. Section 6 gives a brief overview of related work. We conclude this paper by summarizing our findings in Section 7 and briefly outlining possible future work.

2 Preliminaries

In this paper, we look at non-disjunctive *extended logic programs* (ELPs) [4]. An ELP is a finite set of rules over a set $\mathscr{A}$ of propositional atoms. A literal L is either an atom A (*positive literal*) or a negated atom $\overline{A}$ (*negative literal*). For a literal L, the *strongly complementary* literal $\overline{L}$ is $\overline{A}$ if $L = A$ and A otherwise. A *default-negated literal* L, called *default literal*, is written as $\sim L$. Given a set S of literals, we say a literal L *is true in* S (symbolically $S \models L$) iff $L \in S$ and $\sim L$ is true in S (symbolically $S \models \sim L$) iff $L \notin S$. A set of literals is *inconsistent* if it contains strongly complementary literals.

We are now ready to specify the form of ELPs.

A *rule* r is of the form

$$L_0 \leftarrow L_1, \ldots, L_m, \sim L_{m+1}, \ldots, \sim L_n., \tag{1}$$

with literals $L_0, \ldots, L_n$ and $0 \leq m \leq n$. The literal L_0 is the *head* of r, denoted by $H(r)$, and $\{L_1, \ldots L_m, \sim L_{m+1}, \ldots \sim L_n\}$ is the *body* of r, denoted by $B(r)$. Furthermore, $\{L_1, \ldots, L_m\}$ is denoted by $B^+(r)$ and $\{L_{m+1}, \ldots, L_n\}$ by $B^-(r)$. An *extended logic program (ELP)* is a set of rules of the form (1). Given a set S of literals, a rule r *is true in* S (symbolically $S \models r$) iff $H(r)$ is true in S whenever $B(r)$ is true in S. In this case we also say that S *satisfies* r. Given an ELP $\mathscr{P}$ over $\mathscr{A}$ without default negation and $Lit_{\mathscr{A}} = \{\mathscr{A} \cup \{\overline{A} \mid A \in \mathscr{A}\}\}$, the *answer set* of $\mathscr{P}$ is either (a) the smallest set $S \subseteq Lit_{\mathscr{A}}$ such that S is consistent and $S \models r$ for every rule $r \in \mathscr{P}$, or (b) the set $Lit_{\mathscr{A}}$ of literals in case all such sets are inconsistent. Note that, similar to Horn logic programs, each such ELP without default negation has exactly one minimal model which might, however, be inconsistent.

In general, an *answer set of an ELP* $\mathscr{P}$ is determined by its reduct. The *reduct* $\mathscr{P}^S$ of a program $\mathscr{P}$ relative to a set S of literals is defined by

$$\mathscr{P}^S = \{H(r) \leftarrow B^+(r). \mid r \in \mathscr{P}, B^-(r) \cap S = \emptyset\}.$$

A set S of literals is an *answer set* of $\mathscr{P}$ if it is the answer set of $\mathscr{P}^S$ [4].

3 Conflicts and λ-extensions

In [8, 9], a framework is outlined that supports knowledge experts in restoring consistency interactively in answer set programs. To this aim, the bodies of rules involved in a conflict are extended suitably by known literals, so-called λ-extensions. In this way, conflicts that cause inconsistency can be resolved. In particular, a method is provided that for every conflict generates all possible λ-extensions, from which the expert can then choose the most adequate ones.

We briefly recall the basic techniques from [8, 9], in particular the terms *conflicts* and *λ-extensions*, and illustrate them with the following (running) example.

Example 1. *Let $\mathscr{P}_{ex}$ be specified by the following rules:*

$$\begin{array}{llll}
r_1\colon a \leftarrow b, \sim c. & r_2\colon \overline{a} \leftarrow b. & r_3\colon x \leftarrow d, e, f, \sim c. & r_4\colon \overline{x} \leftarrow d, e. \\
r_5\colon y \leftarrow g, h, f. & r_6\colon \overline{y} \leftarrow g. & r_7\colon z \leftarrow j, k, \sim l. & r_8\colon \overline{z} \leftarrow j, \sim l. \\
r_9\colon w \leftarrow f, m, n. & r_{10}\colon \overline{w} \leftarrow m. & r_{11}\colon \overline{w} \leftarrow n. & r_{12}\colon p \leftarrow o, h, f, \sim q. \\
r_{13}\colon \overline{p} \leftarrow o, \sim q. & r_{14}\colon u \leftarrow s. & r_{15}\colon \overline{u} \leftarrow s, \overline{t}, h. & r_{16}\colon \overline{u} \leftarrow s, t, h.
\end{array}$$

Note that $\mathscr{P}_{ex}$ in Example 1 is trivially consistent because it is a so-called *program core* [9], i.e., it has no facts. Program cores are usable with different instances by expanding the program by a corresponding set $\mathscr{F}$ of facts (instance data). A typical example would be a medical expert system where the instance data are provided by the patients and so cannot be part of the generic program.

Example 2. *Suppose the following program $\mathscr{P}$ describing the symptoms of and treatments for two diseases disA and disB:*

$$disA \leftarrow sympM, sympN. \qquad disB \leftarrow sympM, sympO. \qquad treatX \leftarrow disA. \qquad treatY \leftarrow disB.$$

Program $\mathscr{P}$ is a program core. Adding the instance data $\mathscr{F} = \{sympM, sympO\}$ as patient data to $\mathscr{P}$ yields the unique answer set $\mathscr{F} \cup \{disB, treatY\}$ describing that the corresponding patient has condition disB and should be treated with treatment treatY.

However, it is easy to see that there exist multiple instance data $\mathscr{F}$ for $\mathscr{P}_{ex}$ that would yield an inconsistent program $\mathscr{P}_{ex} \cup \mathscr{F}$ because of rules in $\mathscr{P}_{ex}$ that, if their bodies are satisfied simultaneously, derive strongly complementary literals.

Example 3 (Example 1 contd.)**.** *Let $\mathscr{F}_{ex}$ be a set of instance data for $\mathscr{P}_{ex}$ such that $b \in \mathscr{F}_{ex}$ and $c \notin \mathscr{F}_{ex}$. Then in $\mathscr{P}_{ex} \cup \mathscr{F}_{ex}$, both r_1 and r_2 are simultaneously satisfied. Hence, $\mathscr{P}_{ex} \cup \mathscr{F}_{ex}$ is inconsistent as both a and $\overline{a}$ are derivable.*

Rule sets like $\{r_1, r_2\}$ of the previous example are called *conflicting rules* or *conflicts*.

Definition 1 (Conflict and conflict group (cf. [8]))**.** *In a logic program $\mathscr{P}$, two rules r, r' are* conflicting *iff there exists a set S of literals such that S satisfies $B(r)$ and $B(r')$ simultaneously and the head literals $H(r)$ and $H(r')$ are strongly complementary. A* conflict *is a set $\gamma = \{r, r'\}$ of two conflicting rules r, r'. For a rule r, the corresponding* conflict group *$\Gamma(r)$ is the set of all conflicts γ in $\mathscr{P}$ with $r \in \gamma$. The* size of a conflict group *is the number of different conflicts in a conflict group.*

Intuitively, two rules are conflicting if their bodies can be true simultaneously and their head literals are contradictory. The following example illustrates conflict groups and their sizes in $\mathscr{P}_{ex}$.

Example 4 (Example 1 contd.)**.** *The conflict groups for r_1 and r_4 are the sets $\Gamma(r_1) = \{\{r_1, r_2\}\}$ and $\Gamma(r_4) = \{\{r_3, r_4\}\}$, respectively. Both groups have each size 1 while conflict group $\Gamma(r_{14}) = \{\{r_{14}, r_{15}\}, \{r_{14}, r_{16}\}\}$ has size 2.*

Table 1: Components of λ-graph

Group	*Representative*	*Conflicts*	*λ-Ext.*	*Node*	*Cliques*
$\Gamma(r_2)$	r_2	$\{r_1, r_2\}$	$\{c\}$	$(\Gamma(r_2),1)$	Q_3
$\Gamma(r_4)$	r_4	$\{r_3, r_4\}$	$\{\sim f\}, \{c\}$	$(\Gamma(r_4),1)$	Q_2,Q_3
$\Gamma(r_6)$	r_6	$\{r_5, r_6\}$	$\{\sim h\}, \{\sim f\}$	$(\Gamma(r_6),1)$	Q_1,Q_2
$\Gamma(r_8)$	r_8	$\{r_7, r_8\}$	$\{\sim k\}$	$(\Gamma(r_8),1)$	Q_4
$\Gamma(r_{10})$	r_{10}	$\{r_9, r_{10}\}$	$\{\sim f\}$	$(\Gamma(r_{10}),1)$	Q_1
$\Gamma(r_{11})$	r_{11}	$\{r_9, r_{11}\}$	$\{\sim f\}$	$(\Gamma(r_{11}),1)$	Q_1
$\Gamma(r_{13})$	r_{13}	$\{r_{12}, r_{13}\}$	$\{\sim h\}, \{\sim f\}$	$(\Gamma(r_{13}),1)$	Q_1,Q_2
$\Gamma(r_{14})$	r_{14}	$\{r_{14}, r_{15}\}, \{r_{14}, r_{16}\}$	$\{\sim h\}, \{\sim \overline{t}, \sim t\}$	$(\Gamma(r_{14}),2)$	Q_2,Q_5

If both rules of a conflict in a program core are simultaneously satisfiable, a program is potentially inconsistent [9]. In the rest of this paper, we deal with programs that are inconsistent due to *conflicts* and we require a program to be coherent, that is, each given program has at least one answer set (cf. [1]).

In order to guarantee that a program has an answer set whenever it is extended by consistent instance data, all conflicts have to be resolved. For that, we use an approach called *λ-extensions*. In the following, we will present the main aspects of λ-extensions and properties that are relevant to this work. For more details, we refer the reader to [9].

Definition 2 (λ-extensions, cf. [9])**.** *Suppose a program $\mathcal{P}$ and a rule $r \in \mathcal{P}$ that is in conflict with a non-empty set of rules $R \subset \mathcal{P}$. Let X be a set of (default) literals built from atoms occurring in R, and let r'' be a rule obtained from r where $B(r)$ is extended by X, viz., r'': $H(r) \leftarrow B(r) \cup X$.. This set X is a* (conflict-resolving) λ-extension for rule *r if for every rule $r' \in R$ it holds that r' and r'' are no longer conflicting. Such a rule r'' is called a* λ-extended rule w.r.t. X*. A λ-extension X for r is* minimal *iff there exists no set $X' \subset X$ such that X' is also a λ-extension for r. A conflict group $\Gamma(r)$ is called* resolvable *if there exists a rule r' and a λ-extension X for r' such that replacing r' in every conflict of $\Gamma(r)$ by the corresponding λ-extended rule w.r.t. X leads to all conflicts in $\Gamma(r)$ being resolved. Rule r' is then called the* representative of $\Gamma(r)$.

Intuitively, a λ-extension X for a representative r of a resolvable conflict group $\Gamma(r)$ is a set of (default) literals such that if the body of r is expanded by X, any previous conflicts of r are resolved. In [9], the authors show that a conflict $\{r, r'\}$ can only be resolved iff there exists at least one atom in $B(r')$ that is not in $B(r)$ or vice versa. They also demonstrate that resolving each conflict in a program core $\mathcal{P}$ using λ-extensions yields a *uniformly non-contradictory* program core $\mathcal{P}^{\star}$, meaning, the program core can be used with any set of consistent instance data that consists of literals that only appear in rule bodies of $\mathcal{P}$.

We will illustrate the workings of λ-extensions in the following example.

Example 5 (Example 4 contd.)**.** *Consider conflict groups $\Gamma(r_{14})$ in Example 1 which consists of the conflicts $\{r_{14}, r_{15}\}$ and $\{r_{14}, r_{16}\}$. For these conflicts, we get the λ-extensions $\{\sim h\}$, $\{\overline{h}\}$ and $\{\sim t, \sim \overline{t}\}$ as possible extensions for the rule body of r_{14}. This means both conflicts of r_{14} can be solved by replacing r_{14} in $\mathcal{P}_{ex}$ by one of the following rules:*

$$r'_{14}\colon u \leftarrow s, \sim h. \qquad r''_{14}\colon u \leftarrow s, \overline{h}. \qquad r'''_{14}\colon u \leftarrow s, \sim \overline{t}, \sim t.$$

Note that no additional conflicts are introduced as the premise for r is solely extended yielding a more specific condition. However, not every conflict group of a program is resolvable. The following example shows that in order to utilize conflict groups for the resolution of conflicts, their proper selection is a crucial step.

Example 6 (Example 4 contd.)**.** *For rules* r_1 *and* r_2 *it holds that* $\Gamma(r_1) = \Gamma(r_2) = \{\{r_1, r_2\}\}$*. As* $B(r_2) \subseteq B(r_1)$ *holds,* r_1 *can not be picked as a representative. However,* $B(r_1) \backslash B(r_2) = \{\sim c\}$ *holds and thus the* λ*-extension* $\{c\}$ *resolves the conflict of both conflict groups. Similarly, regarding* $\Gamma(r_4)$*, we get the* λ*-extensions* $\{\overline{f}\}$*,* $\{\sim f\}$*, and* $\{c\}$ *for* r_4*. Note that conflict group* $\Gamma(r_9)$ *has no possible representative as there does not exist any* λ*-extension for* r_9 *that solves all conflicts in* $\Gamma(r_9)$*. It is therefore necessary to pick the conflict groups* $\Gamma(r_{10})$ *and* $\Gamma(r_{11})$ *which are both resolvable via the representative* r_{10} *and* r_{11} *respectively. Table 1 shows the* λ*-extensions of the different resolvable conflict groups of* $\mathscr{P}_{ex}$*.*

In the rest of this paper for any atom a, we omit the λ-extension that contains $\overline{a}$ whenever $\sim a$ and $\overline{a}$ can both be used because $\sim a$ is more cautious. In Example 6, we henceforth solely state $\{c\}$ and $\{\sim f\}$ as the λ-extensions for r_4.

Furthermore, note that conflicts like $\{a \leftarrow b., \overline{a} \leftarrow b.\}$ require an expansion of both rules bodies by complementary literals in order to be resolved. We argue that suitable expansions for such conflicts can only be determined by the knowledge expert. Thus, conflicts of this type will not be considered in this paper.

4 Relationship between λ-extensions

Naturally, a practical implementation of the interactive framework for conflict resolution, as presented in [8, 9], has to provide a proper workflow to resolve each conflict and suggest solutions in a suitable order. We propose that a syntax-based approach considers the different connections between possible solutions in order to condense the resolution of multiple conflicts. For this reason, we introduce λ*-graphs* and corresponding *clique covers* that can be used to point out such connections. Based on these results in Section 5, we demonstrate how λ-graphs and clique covers can be used to define explicit strategies that specify in which order conflicts and solutions should be presented to the knowledge expert.

For resolving conflicts in a program thoroughly, we have to make sure that each rule that is involved in a conflict must be taken into regard. This idea is formalized by conflict group covers.

Definition 3 (Conflict group cover)**.** *Let* $\mathscr{P}_{cf}$ *be the set of all rules in* $\mathscr{P}$ *that are part of a conflict, and let* $\mathbf{\Gamma}$ *be a set of resolvable conflict groups. Then,* $\mathbf{\Gamma}$ *is a* conflict group cover *of* $\mathscr{P}$ *if each rule in* $\mathscr{P}_{cf}$ *appears in at least one conflict group of* $\mathbf{\Gamma}$*. The set of all inclusion-minimal conflict group covers of a logic program* $\mathscr{P}$ *is denoted by* $CCG(\mathscr{P})$ *.*

A conflict group cover $\mathbf{\Gamma}$ of $\mathscr{P}$ therefore implies via the representative of each conflict group in $\mathbf{\Gamma}$ which rules in $\mathscr{P}$ shall be modified.

Notice that a conflict group cover $\mathbf{\Gamma}$ involves a sufficient set of rules that have to be modified to resolve every conflict.

Proposition 1. *Given a conflict group cover* $\mathbf{\Gamma}$*, expanding the body of each rule r in* $\{r \mid \text{ there is } \Gamma(r') \in \mathbf{\Gamma} \text{ such that } r \text{ is a representative of } \Gamma(r')\}$ *by one of its respective* λ*-extensions yields a consistent program.*

Proof. Let $\mathbf{\Gamma}$ be a conflict group cover of a program $\mathscr{P}$ with conflicts and $\mathscr{P}_{cf}$ the set of conflicting rules in $\mathscr{P}$. By Definition 3, a set $\mathbf{\Gamma} \in CCG(\mathscr{P})$ is a set of conflict groups such that every rule in $\mathscr{P}_{cf}$ appears in at least one conflict group $\Gamma \in \mathbf{\Gamma}$. Since every conflict group in $\mathbf{\Gamma}$ is resolvable by Definition 3, there exists at least one λ-extension for the representative of each conflict group in $\mathbf{\Gamma}$. This in turn means that applying a respective λ-extension to the representative of each conflict group in $\mathbf{\Gamma}$ resolves every conflict in $\mathscr{P}$, thus, yielding a conflict-free program $\mathscr{P}'$. □

This result implies that regarding conflicting rules, it is mandatory for the knowledge expert to initially decide which rules are allowed to be modified and which rules must stay unaffected. This allows to determine all appropriate conflict groups and consequently all appropriate conflict groups covers. Choosing the most suitable cover then ensures that a sufficient and moreover the most suitable set of rules can be modified using λ-extensions to obtain a uniformly non-contradictory program.

We can now define λ-graphs w.r.t. a conflict group cover $\mathbf{\Gamma}$ by making use of weights for the nodes, and labels for the edges to store information which is crucial for the resolution process.

Definition 4 (λ-graph)**.** *Given a conflict group cover $\mathbf{\Gamma} \in CCG(\mathscr{P})$ of a logic program $\mathscr{P}$, the* $\mathbf{\Gamma}$-induced λ-graph $G(\mathbf{\Gamma})$ *is a tuple (V,E) of weighted nodes V and labeled edges E where V contains a weighted node (Γ, w_Γ) for each conflict group Γ in $\mathbf{\Gamma}$ with size w_Γ, and E contains labeled edges (Γ,Γ',λ) whenever the representatives of two different conflict groups Γ and Γ' have a common λ-extension λ, and for any λ-extension λ that is an extension for a representative of only one conflict group Γ, there exists a self-loop (Γ,Γ,λ).*

Since every conflict group in a program $\mathscr{P}$ is represented by a weighted node and each node has either an edge to itself or to another node that shares a common λ-extension, every conflict is considered in a λ-graph.

We now illustrate $\mathbf{\Gamma}$-induced λ-graphs using the running example.

Example 7 (Example 1 contd.)**.** *Suppose $\mathscr{P}_{ex} = \mathscr{P}_{cf}$ and the conflict group cover $\mathbf{\Gamma}_{ex} = \{\Gamma(r_2), \Gamma(r_4), \Gamma(r_6), \Gamma(r_8), \Gamma(r_{10}), \Gamma(r_{11}), \Gamma(r_{13}), \Gamma(r_{14})\} \in CCG(\mathscr{P}_{ex})$ are given. Table 1 shows the conflict groups in $\mathbf{\Gamma}_{ex}$ and the λ-extensions of the corresponding representatives as stated in Table 1. The $\mathbf{\Gamma}_{ex}$-induced λ-graph $G(\mathbf{\Gamma}_{ex}) = (V,E)$ is obtained in the following way: for each conflict group Γ and its size w_Γ, we define (Γ, w_Γ) as the corresponding weighted node. Then, V consists of all such weighted nodes, viz., $V = \{(\Gamma(r_2),1), (\Gamma(r_4),1), (\Gamma(r_6),1), (\Gamma(r_8),1), (\Gamma(r_{10}),1), (\Gamma(r_{11}),1), (\Gamma(r_{13}),1), (\Gamma(r_{14}),2)\}$. The set E consists of all labeled edges (Γ,Γ',lb) such that Γ and Γ' are pairs of different nodes in V that share the common label lb, or pairs of identical nodes Γ if the extension corresponding to lb is only a solution for Γ, viz., $E = \{\ (\Gamma(r_2),\Gamma(r_4),\{c\}),\ (\Gamma(r_8),\Gamma(r_8),\{{\sim}k\}),\ (\Gamma(r_{14}),\Gamma(r_{14}),\{{\sim}\bar{t},{\sim}t\})\ \} \cup F \cup H$ where $F = \{(\Gamma(r),\Gamma(r'),\{{\sim}f\}) \mid r,r' \in \{r_4,r_6,r_{10},r_{11},r_{13}\}, r \neq r'\}$ and $H = \{(\Gamma(r),\Gamma(r'),\{{\sim}h\}) \mid r,r' \in \{r_6,r_{13},r_{14}\}, r \neq r'\}$.*

The graphical representation of the resulting λ-graph $G(\mathbf{\Gamma}_{ex})$ is displayed in Figure 1.

The graph illustrates several complete subgraphs which are sets of nodes where all nodes are connected to each other by an edge with the same label. Such subgraphs we call *λ-cliques*.

Definition 5 (λ-clique)**.** *Suppose a logic program $\mathscr{P}$ and a conflict group cover $\mathbf{\Gamma} \in CCG(\mathscr{P})$. A* λ-clique *w.r.t. a label lb in a λ-graph $G(\mathbf{\Gamma}) = (V,E)$ is a maximal subgraph $G(\mathbf{\Gamma},lb) = (V',E')$ of $G(\mathbf{\Gamma})$ where (1) $E' \subseteq E$ contains all edges in E with label lb, and (2) $V' \subseteq V$ contains every node that is connected to an edge in E'. We define the* weight of a λ-clique *as the sum of the weights of all nodes that occur in the λ-clique. The set of all λ-cliques in a λ-graph $G(\mathbf{\Gamma})$ is denoted by $CLQ(\mathbf{\Gamma})$.*

In the following, given a set of edges E, its subset of all edges with label lb is denoted by E^{lb}.

Since a λ-clique $G(\mathbf{\Gamma},lb)$ contains all edges of E^{lb} and all their connected nodes, every λ-clique is a complete graph.

Example 8 (Example 7 contd.)**.** *$G(\mathbf{\Gamma}_{ex}) = (V,E)$ contains the following five λ-cliques:*

$Q_1 = (\{(\Gamma(r_6),1),(\Gamma(r_{13}),1),(\Gamma(r_{14}),2)\},E^{\sim h},\{{\sim}h\})$ *with weight 4*

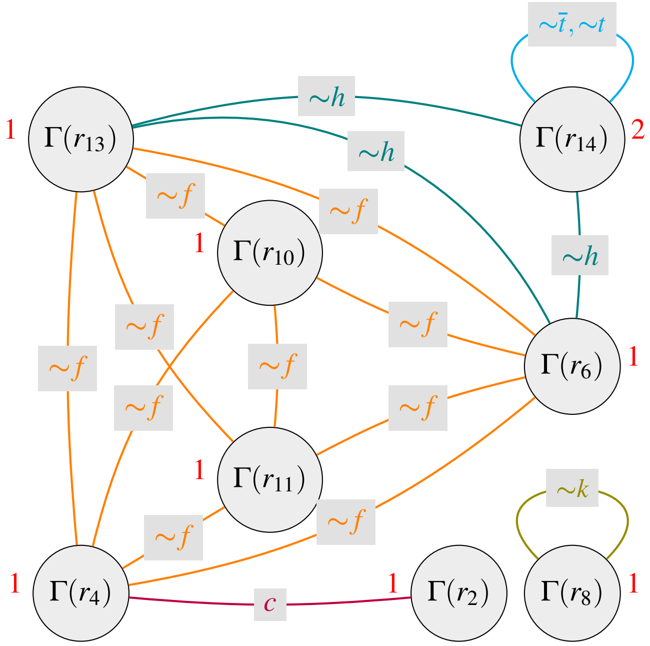


Figure 1: λ-graph $G(\mathbf{\Gamma}_{ex})$

$Q_2 = (\{(\Gamma(r_4),1),(\Gamma(r_6),1),(\Gamma(r_{10}),1),(\Gamma(r_{11}),1),(\Gamma(r_{13}),1)\},E^{\sim f},\{\sim f\})$ *with weight 5*

$Q_3 = (\{(\Gamma(r_2),1),(\Gamma(r_4),1)\},E^{\sim c},\{c\})$ *with weight 2*

$Q_4 = (\{(\Gamma(r_8),1)\},E^{\sim k},\{\sim k\})$ *with weight 1*

$Q_5 = (\{(\Gamma(r_{14}),2)\},E^{\sim t,\sim\bar{t}},\{\sim t,\sim\bar{t}\})$ *with weight 2*

Table 1 shows which conflicts are involved in the different cliques

As all conflicts represented in a λ-clique (V',E') share a λ-extension λ, its weight ω indicates that ω different conflicts can be solved by extending $B(r)$ of each representative rule r in $(\Gamma(r),w) \in V'$ by lb.

To find sets of cliques such that every conflict in a program is considered, we introduce *clique covers* for λ-graphs.

Definition 6 (Clique cover)**.** *Suppose a logic program $\mathscr{P}$ and a set $\mathbf{Q} \subseteq CLQ(\mathbf{\Gamma})$ of λ-cliques in a graph $G(\mathbf{\Gamma}) = (V,E)$ are given. We say $\mathbf{Q}$ is a* clique cover *for $G(\mathbf{\Gamma})$ if every node in V appears in at least one clique of $\mathbf{Q}$. Moreover, $\mathbf{Q}$ is the* minimal *clique cover if there is no clique cover $\mathbf{Q}'$ for $G(\mathbf{\Gamma})$ s. t. $|\mathbf{Q}'| \lneq |\mathbf{Q}|$.*

A minimal clique cover for a graph $G(\mathbf{\Gamma})$, therefore, provides us with minimal compositions of cliques where every conflict is considered. For this reason, our approach uses clique covers to determine which conflicts can be solved by the same λ-extensions which in turn can be used to find a suitable order in which conflicts and their solutions can be suggested to the user. Herewith, we arrive at the following result which will be useful in the following.

Proposition 2. *Given a program $\mathscr{P}$ with conflicts and a λ-clique $\mathbf{\Gamma} \in CCG(\mathscr{P})$, a minimal clique cover for $G(\mathbf{\Gamma})$ provides a minimal set of λ-extensions $\mathbf{L} = \{lb \mid (V',lb) \in \mathbf{Q}\}$ that is required to obtain a program without conflicts.*

Proof. Let $\mathbf{\Gamma} \in CCG(\mathscr{P})$ be a λ-clique of a program $\mathscr{P} = (V,E)$ with conflicts, and $\mathbf{Q} \subseteq CLQ(\mathbf{\Gamma})$ a clique cover for $G(\mathbf{\Gamma})$. By Definition 4, V contains a node for each conflict group of a conflict group cover $\mathbf{\Gamma}$. By Proposition 1, $\mathbf{\Gamma}$ considers all conflicting rules of $\mathscr{P}$. Thus, every conflict in $\mathscr{P}$ is implicitly represented by at least one node in V. By Definition 5, a λ-clique (V', E^{lb}, lb) in $G(\mathbf{\Gamma})$ condenses conflict groups that share a common λ-extension lb, and by Definition 6, each node in V appears in at least one clique of $\mathbf{Q}$. Hence, $\mathbf{Q}$ implies a set of λ-extensions $\mathbf{L} = \{lb \mid (V', lb) \in \mathbf{Q}\}$ that are sufficient to resolve all conflicts in $\mathscr{P}$, by extending the body of representative rules in the conflict groups suitably. Likewise if such a clique cover is minimal, $\mathbf{\Gamma}$ implies the smallest set of solutions for $\mathscr{P}$. ☐

The notion of minimal cover is illustrated in the following example.

Example 9 (Example 7 contd.)**.** *The minimal clique cover in* $G(\mathbf{\Gamma}_{ex})$ *is the set* $\{Q_1, Q_2, Q_3, Q_4\}$. *Therefore, the four* λ*-extensions* $\{c\}$, $\{\sim f\}$, $\{\sim h\}$, *and* $\{\sim k\}$ *suffice to obtain a program without conflicts.*

Now we are ready to show how the results provide the crucial basis to define suitable orders over conflicts and λ-extensions for their usage in conflict resolution frameworks.

5 Sorting conflicts and λ-extensions

In this section we show how λ-graphs and clique covers can be utilized to define strategies that compute (1) an order over all conflict groups of a program, and (2) an order over the λ-solutions of each conflict group. These orders can then be used to define in which sequence they should be presented to the knowledge expert. The goal is to improve the efficiency of the interactive conflict resolution process with the expert and to facilitate the resolution process overall by prioritizing those conflict groups whose solutions can be used for other conflict groups and thereby solve the most amount of conflicts simultaneously. For that reason in this section, we provide an example of such a strategy that makes use of the technical notions introduced in the preceding sections.

We begin by introducing the notion of *relationships*: We say a λ-clique Q *is related to another* λ*-clique* Q' iff Q and Q' share a common node. Moreover, we say that a conflict group Γ is *part of a* λ*-clique* Q if the node that corresponds to Γ is in Q. With these conventions, we are now able to define a strategy to sort conflicts and their λ-extensions such that the user can resolve all conflicts in a more suitable way.

5.1 Order strategy

The goal of an order strategy is to establish an order in which the conflict groups are presented to the user and to additionally obtain an order for each conflict group that specifies how the respective λ-extensions are suggested. The primary objective is to assist the knowledge expert to efficiently find the correct resolution for each conflict by preferring solutions that solve the most amount of conflicts simultaneously.

Recall that a clique cover of $G(\mathbf{\Gamma})$ contains a set of conflict groups that considers all conflicts in $\mathscr{P}$ and, by definition, also implicitly provides possible solutions for every conflict. The clique cover furthermore specifies explicitly which rules of $\mathscr{P}$ will be modified. The following strategy will illustrate how these properties can be used to obtain an order over conflict groups and λ-extensions.

The first step employs two sorting criteria in lexicographical order. First, a general order over the conflict groups in $\mathbf{\Gamma}$ is determined by viewing the number of edges of each node in $G(\mathbf{\Gamma})$ in order to prefer conflict groups with less possible solutions. Conflicts with only one possible solution can hereby be dealt

Algorithm 1 Interactive conflict resolution

Input: Logic program $\mathscr{P}$ with conflicts

Output: Uniformly non-contraditory logic program $\mathscr{P}^\star$

```
Let $\mathscr{P}^\star = \mathscr{P}$.
while $\mathscr{P}^\star$ has conflicts do
    Choose a suitable conflict group cover $\mathbf{\Gamma} \in CCG(\mathscr{P}^\star)$.
    Generate corresponding $\lambda$-graph $G(\mathbf{\Gamma})$.
    Choose a suitable clique cover $\mathbf{Q} \subseteq CLQ(\mathbf{\Gamma})$.
    Compute $>_{\mathbf{\Gamma}}$ : Sort conflict groups by the number of related $\lambda$-cliques in ascending order.
    for Element $e$ in $>_{\mathbf{\Gamma}}$ do
        if $e$ contains more than one conflict group then
            Extend order $>_{\mathbf{\Gamma}}$ : sort conflict groups in $e$ by their weight in descending order.
            Sort conflict groups with the same weight in alphanumerical order.
        end if
    end for
    for Element $\Gamma(r)$ in $>_{\mathbf{\Gamma}}$ do
        Compute $\succ_{\Gamma(r)}$: Sort $\lambda$-extensions of the representative of $\Gamma(r)$ by their $\lambda$-clique's weight.
        for Element $e$ in $\succ_{\Gamma(r)}$ do
            if $e$ contains multiple $\lambda$-extensions then
                Extend $\succ_{\Gamma(r)}$: Sort $\lambda$-extensions in $e$ in alphanumerical order.
            end if
        end for
    end for
    Present expert the conflict groups and the $\lambda$-extensions of the representatives in their respective
    order.
    if Expert chooses a $\lambda$-extension for a conflict group $\Gamma(r)$ then
        In $\mathscr{P}^\star$, replace $r$ by the corresponding $\lambda$-extended rule of $r$.
    end if
end while
return $\mathscr{P}^\star$
```

with first which can potentially reduce the complexity of subsequent conflict resolutions. However, since many conflict groups can have the same amount of solutions, we refine this order in a subsequent action by taking the corresponding clique weights into account.

The second step of the strategy determines an order over λ-extensions for each conflict group. Here again, clique weights are utilized. Consequently, this strategy defines an order over conflict groups and possible solutions where those groups with the least possible solutions are presented first and for each conflict group those solutions are preferred that can be used to resolve related conflicts in parallel.

We now define these steps in more detail. Each step is explained by means of our running example.

Step 1a: In the first step, the conflict groups are ordered by the number of cliques they are part of. The user is shown those conflict groups first that are related to the least amount of cliques. By this, the knowledge expert is being presented with as few choices at a time as possible.

Example 10 (Example 9 contd.). *The respective representatives of conflict groups* $\Gamma(r_2)$, $\Gamma(r_8)$, $\Gamma(r_{10})$, *and* $\Gamma(r_{11})$ *each have only one possible* λ*-extension. The representatives of all remaining conflict groups* *have two. For $G(\mathbf{\Gamma}_{ex})$, we thus get the preliminary order of conflict groups*

$$\Gamma(r_2), \Gamma(r_8), \Gamma(r_{10}), \Gamma(r_{11}) >_{\mathbf{\Gamma}} \Gamma(r_4), \Gamma(r_6), \Gamma(r_{13}), \Gamma(r_{14})$$

that says that all conflict groups on the left side should be presented before those on the right.

It is easy to see that even in smaller programs, this kind of order can be too coarse-grained. To order conflict groups that have the same amount of possible solutions, we propose the following subsequent step.

Step 1b: To refine the order obtained in Step 1a, we use the weight of the cliques. As mentioned before, the weight of a clique represents the amount of different conflict groups that can be solved by the λ-extension that is represented by the label of the clique. Therefore, the conflict groups with the same amount of solutions should additionally be arranged by the sum of weights of all cliques they are part of in descending order. Thereby, we provide the knowledge expert with the opportunity to resolve as many conflicts as possible as soon as possible. If there are conflict groups with the same total weight, we simply arrange them in alphanumerical order.

Example 11 (Ex. 10 contd.)**.** *Conflict group $\Gamma(r_2)$ is only part of clique Q_3 that has weight 2. Conflict group $\Gamma(r_8)$ is only part of clique Q_4 with weight 1. Conflict groups $\Gamma(r_{10})$ and $\Gamma(r_{11})$ are only part of clique Q_2 that has weight 5. For conflict group $\Gamma(r_4)$, which is both in Q_2 and Q_3, we get the total weight of 7 as Q_3 has weight 2 and Q_2 has weight 5. Likewise for $\Gamma(r_6)$ and $\Gamma(r_{13})$, we get a total weight of 9, and for $\Gamma(r_{14})$ a total weight of 6. This way, for $G(\mathbf{\Gamma}_{ex})$ we obtain the following, more specific order:*

$$\Gamma(r_{10}) >_{\mathbf{\Gamma}} \Gamma(r_{11}) >_{\mathbf{\Gamma}} \Gamma(r_2) >_{\mathbf{\Gamma}} \Gamma(r_8) >_{\mathbf{\Gamma}} \Gamma(r_6) >_{\mathbf{\Gamma}} \Gamma(r_{13}) >_{\mathbf{\Gamma}} \Gamma(r_4) >_{\mathbf{\Gamma}} \Gamma(r_{14})$$

Step 2: Step 1 provides us with a suitable order over conflict groups. As conflict groups can have multiple λ-extensions (see Example 5), Step 2 defines how one can obtain an order $\succ_\Gamma$ over all λ-extensions for the representative of each conflict group Γ. For that we will use the weight of their respective cliques. That is, the λ-extensions for the representative in each group are ordered by the weight of their respective clique in descending order. If for two extensions the clique weight is identical, again, we sort them in alphanumerical order.

Example 12 (Ex. 11 contd.)**.** *The representative of conflict groups $\Gamma(r_2)$, $\Gamma(r_8)$, $\Gamma(r_{10})$, and $\Gamma(r_{11})$ each have their own unique solution, viz., $\{c\}$, $\{\sim k\}$, and $\{\sim f\}$. According to Step 2 for $\Gamma(r_6)$, $\{\sim f\}$ should be suggested first as its clique has weight 5, and, if the expert does not accept the first extension, $\{\sim h\}$ whose clique has size 4 can be presented as an alternative. The same order also holds for $\Gamma(r_{13})$ as the possible λ-extensions of their representatives are identical. For $\Gamma(r_4)$, $\{\sim f\}$ has also to be suggested first, then the remaining solution $\{c\}$. Similarly, for $\Gamma(r_{14})$, $\{\sim h\}$ has to be presented first and $\{\sim t, \sim\bar{t}\}$ after that. As a result, for those conflict groups with multiple extensions, we gain the following orders:*

$$\{\sim f\} \succ_{\Gamma(r_4)} \{c\}, \qquad \{\sim f\} \succ_{\Gamma(r_6)} \{\sim h\}, \qquad \{\sim f\} \succ_{\Gamma(r_{13})} \{\sim h\}, \qquad \{\sim h\} \succ_{\Gamma(r_{14})} \{\sim t, \sim\bar{t}\}$$

Keep in mind that since the expert can apply a λ-extension on conflicts of multiple conflict groups simultaneously, some of the subsequent conflict groups and solutions can become obsolete. It is therefore necessary to recalculate the sorting of the remaining conflict groups and solutions once the expert accepted a λ-extension to be applied as the input program changes, in other words, after each program modification the remaining conflict groups have to be identified and the order over conflict groups and λ-extensions have to be computed accordingly. The computation of λ-graphs and all its orders after each program modification is however expedient as it provides the means to implement the crucial functionality to postpone the resolution of certain conflicts as the orders of conflict groups and λ-extension are only a recommendation based on the syntactical properties of the program.

Algorithm 1 summarizes the complete workflow. We recommend that in an actual implementation, the choices stated in lines 3 and 5 should be made in interaction with the expert by presenting the conflicts in an appropriate fashion as Example 6 shows that choosing the representatives of conflicts is a crucial step and not straightforward especially if multiple rules of a conflict are eligible for modification by a λ-extension. An explicit implementation should therefore provide the ability to revise the chosen conflict groups and representatives if the final program is not deemed satisfactory.

To conclude this section, we illustrate the workings of this strategy by applying it on the running example. This example will also propose possible ways to provide the knowledge expert with additional information using the λ-graph and simulate a possible line of thought of the expert.

Example 13 (Example 12 contd.). *Assume that a knowledge expert is assigned to resolve all conflicts in $\mathscr{P}_{ex}$. According to the conflict group order obtained in Example 11, conflict group $\Gamma(r_{10})$ is presented first. As stated in Table 1, r_{10} only has the possible solution $\{\sim f\}$. The λ-graph of $G(\boldsymbol{\Gamma}_{ex})$ shows that this solution belongs to λ-clique Q_5 (see Example 8) which states that the representatives of conflict groups $\Gamma(r_4)$, $\Gamma(r_6)$, $\Gamma(r_{11})$, and $\Gamma(r_{13})$ all have $\{\sim f\}$ as a possible solution. By showing these connections, the knowledge expert can decide whether they want to apply the extension $\{\sim f\}$ to to the representatives of other conflict groups of Q_5 as well. Suppose that the expert knows of a connection between the property encoded in f and those encoded in y, w, and p. Regarding atom x and rules r_3 and r_4, the expert sees no immediate connection.They therefore decide to apply the λ-extension $\sim f$ not only to r_{10}, but also to r_6, r_{11}, and r_{13}. This action leaves the set $\boldsymbol{\Gamma}' = \{\Gamma(r_2),\Gamma(r_4),\Gamma(r_8),\Gamma(r_{14})\}$ of unresolved conflict groups with the resulting λ-graph $G(\boldsymbol{\Gamma}') = (\{(\Gamma(r_2),1), (\Gamma(r_4),1), (\Gamma(r_8),1), (\Gamma(r_{14}),2)\}, \{(\Gamma(r_2),\Gamma(r_4),\{c\}), (\Gamma(r_8),\Gamma(r_8),\{\sim k\}), (\Gamma(r_{14}),\Gamma(r_{14}),\{\sim h\}), (\Gamma(r_{14}),\Gamma(r_{14}),\{\sim t,\sim\bar{t}\})\}$ Note that since the conflicts of $\Gamma(r_6)$ and $\Gamma(r_{13})$ are resolved now, there is an additional self-loop $(\Gamma(r_{14}),\Gamma(r_{14}),\{\sim h\})$ for $\Gamma(r_{14})$. This reduced λ-graph therefore leads to the following conflict group order*

$$\Gamma(r_2) >_{\boldsymbol{\Gamma}'} \Gamma(r_4) >_{\boldsymbol{\Gamma}'} \Gamma(r_8) >_{\boldsymbol{\Gamma}'} \Gamma(r_{14}).$$

As before, along with $\Gamma(r_2)$, the expert is also presented with $\Gamma(r_4)$ as they both still build up λ-clique Q_3 and their representatives share the common λ-extension $\{c\}$. Let the knowledge expert apply $\{c\}$ to both representatives, thereby reflecting that $\{c\}$ is a more suitable solution for resolving the conflict in $\Gamma(r_4)$ than $\{\sim f\}$. This leaves the expert with the last conflict group cover $\boldsymbol{\Gamma}'' = \{\Gamma(r_8),\Gamma(r_{14})\}$ and the resulting λ-graph $G(\boldsymbol{\Gamma}'') = (\{(\Gamma(r_8),1), (\Gamma(r_{14}),2)\}, \{(\Gamma(r_8),\Gamma(r_8),\{\sim k\}), (\Gamma(r_{14}),\Gamma(r_{14}),\{\sim h\}), (\Gamma(r_{14}),\Gamma(r_{14}),\{\sim t,\sim\bar{t}\})\})$. For $\Gamma(r_8)$, the expert chooses the only solution $\{\sim k\}$. For the conflicts of $\Gamma(r_{14})$, the expert can lastly choose between $\{\sim h\}$ and $\{\sim t,\sim\bar{t}\}$ as λ-extensions. Originally, the order over these extensions was $\{\sim h\} \succ_{\Gamma(r_{14})} \{\sim t,\sim\bar{t}\}$ (see Example 12) because λ-clique Q_1 has a higher weight than Q_5. Now that all conflicts are resolved except for those of $\Gamma(r_{14})$ and the weight of the λ-clique regarding $\sim h$ decreased from 4 to 2, the order between the two λ-extensions is determined by their alphanumerical order. In this case, the alphanumerical order coincides with the original order. Thus, for $\Gamma(r_{14})$, the expert is first presented with λ-extension $\{\sim h\}$ which they immediately choose as the most fitting solution. This concludes the conflict resolution process that outputs the following conflict-free program:

$r_1\colon a \leftarrow b, \sim c.$
$r_2\colon \overline{a} \leftarrow b, c.$
$r_3\colon x \leftarrow d, e, f, \sim c.$
$r_4\colon \overline{x} \leftarrow d, e, c.$
$r_5\colon y \leftarrow g, h, f.$
$r_6\colon \overline{y} \leftarrow g, \sim f.$
$r_7\colon z \leftarrow j, k, \sim l.$
$r_8\colon \overline{z} \leftarrow j, \sim l, \sim k.$
$r_9\colon w \leftarrow f, m, n.$
$r_{10}\colon \overline{w} \leftarrow m, \sim f.$
$r_{11}\colon \overline{w} \leftarrow n, \sim f.$
$r_{12}\colon p \leftarrow o, h, f, \sim q.$
$r_{13}\colon \overline{p} \leftarrow o, \sim q, \sim f.$
$r_{14}\colon u \leftarrow s, \sim h.$
$r_{15}\colon \overline{u} \leftarrow s, \overline{t}, h.$
$r_{16}\colon \overline{u} \leftarrow s, t, h.$

Note that Example 13 illustrates just one of many possible ways how ordering strategies and λ-graphs can be utilized for the implementation of interactive resolution of conflicts. For instance, instead of completely omitting conflicts during the process once they are resolved, these conflicts can be shown further on, only flagging them as resolved. This would provide the expert with additional information that is otherwise removed once the cliques of resolved conflict groups are removed. Such a functionality could also be extended by the possibility to revert previous modifications.

6 Related work

The method of conflict resolution is closely related to the topic of *ASP debugging*. There we find several approaches that deal with the modification of logic programs [3, 6, 7]. These programs are not necessarily inconsistent. They rather help the user knowledge expert to fix the mismatch between the current program's semantics and the semantics intended by the program's modeller. In [2], the authors utilize the notion of incoherence to implement the debugging of programs. All these approaches, however, require the knowledge expert to provide further information in order to detect and resolve the faulty parts of the program.

In [5], the authors present an approach to resolve inconsistency (by contradictions or incoherence) by finding minimal sets of rules that are causing inconsistency. Similar to [9], it also presents a way to compute possible solutions, which in this case are minimal correction sets of rules whose removal guarantee that the reduced program is consistent. These minimal sets are identical to those found in the presented approach if the inconsistency is caused by contradictory literals. Compared to that work, our approach in this paper helps to preserve information by not removing those rules, but instead exploiting dependencies and subtle differences in conflicting rules so that the knowledge expert is provided with suitable information to sharpen the rules by extending them. In this way, (potential) conflicts are resolved and actually help to make the knowledge expressed by the program more professionally adequate.

In this work, we provide an interactive solution strategy by suggesting an order over the problem causes and an order over the possible solutions. Once suitable solutions are available, both, the approach using λ-extensions as well as debugging approaches like those based on the *meta-programming technique* [3] can be used to obtain a consistent program.

7 Conclusion and future work

In practice, it is often imperative that a program (core) is usable with different instance data. For example, in the medical sector, decision support systems are often required to be usable with different patient data. Resolving conflicts in this manner ensures that program cores can be used in such real-world applications in a safely manner w.r.t. consistency. This paper extends the work in [8, 9] that propose a framework for obtaining uniformly non-contraditory logic programs by enabling knowledge experts to interactively resolve conflicts. Since conflicting rules can have a large amount of possible solutions, methods are necessary that provide a proper order in which the conflicts of a program and their possible solutions should be presented to the knowledge expert. This paper tackled this issue by providing the theoretical groundwork to define such sorting methods. We furthermore provide an explicit strategy for sorting conflicts and λ-extensions to illustrate the results. We have shown that the syntactical structure of answer set programs suffices to acquire preferences over conflicts and their solutions as it provides the necessary information to find relationships between conflicting rules and their λ-extensions.

Further investigations are needed to combine the syntax-based view to sort elements during the conflict resolution process with methods based on the (intended) semantics of the program. For this, proper interaction with the user is needed to obtain the relevant information. In future work we want to extend the approach of λ-extensions to consider other types of conflict groups as they are currently limited to finding solution for a single rule that is in conflict with one or multiple other rules and, for example, not the other way around (see rules r_9-r_{11} in Example 1). Such results offer data to find more connections between rules and solutions and ergo for even more precise sorting strategies. To illustrate the practical relevance and feasibility, we are currently working on an implementation of a conflict resolution framework with the presented capabilities.

References

[1] Stefania Costantini, Benedetto Intrigila & Alessandro Provetti (2003): *Coherence of updates in answer set programming*. In: *In Proc. of the IJCAI-2003 Workshop on Nonmonotonic Reasoning, Action and Change*, pp. 66–72.

[2] Carmine Dodaro, Philip Gasteiger, Kristian Reale, Francesco Ricca & Konstantin Schekotihin (2019): *Debugging Non-ground ASP Programs: Technique and Graphical Tools*. *Theory Pract. Log. Program.* 19(2), pp. 290–316, doi:10.1017/S1471068418000492.

[3] Martin Gebser, Jörg Pührer, Torsten Schaub & Hans Tompits (2008): *A Meta-Programming Technique for Debugging Answer-Set Programs*. In Dieter Fox & Carla P. Gomes, editors: *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, AAAI Press, pp. 448–453.

[4] Michael Gelfond & Vladimir Lifschitz (1991): *Classical Negation in Logic Programs and Disjunctive Databases*. *New Gener. Comput.* 9(3/4), pp. 365–386, doi:10.1007/BF03037169.

[5] Carlos Mencía & João Marques-Silva (2020): *Reasoning About Strong Inconsistency in ASP*. In Luca Pulina & Martina Seidl, editors: *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, *Lecture Notes in Computer Science* 12178, Springer, pp. 332–342, doi:10.1007/978-3-030-51825-7_24.

[6] Johannes Oetsch, Jörg Pührer & Hans Tompits (2011): *Stepping through an Answer-Set Program*. In James P. Delgrande & Wolfgang Faber, editors: *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, *Lecture Notes in Computer Science* 6645, Springer, pp. 134–147, doi:10.1007/978-3-642-20895-9_13.

[7] Kostyantyn M. Shchekotykhin (2015): *Interactive Query-Based Debugging of ASP Programs*. In Blai Bonet & Sven Koenig, editors: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, AAAI Press, pp. 1597–1603.

[8] Andre Thevapalan & Gabriele Kern-Isberner (2020): *Towards Interactive Conflict Resolution in ASP Programs*. In Maria Vanina Martínez & Ivan Varzinczak, editors: *Proceedings of the 18th International Workshop on Non-Monotonic Reasoning, NMR 2020*, pp. 29–36.

[9] Andre Thevapalan & Gabriele Kern-Isberner (2022): *On Establishing Robust Consistency in Answer Set Programs*. *Theory and Practice of Logic Programming*, p. 1–34, doi:10.1017/S1471068422000357.

Inductive Learning of Declarative Domain-Specific Heuristics for ASP[*]

Richard Comploi-Taupe[1]
[1] Siemens AG Österreich, Vienna, Austria,
`richard.taupe@siemens.com`

Domain-specific heuristics are a crucial technique for the efficient solving of problems that are large or computationally hard. Answer Set Programming (ASP) systems support declarative specifications of domain-specific heuristics to improve solving performance. However, such heuristics must be invented manually so far. Inventing domain-specific heuristics for answer-set programs requires expertise with the domain under consideration and familiarity with ASP syntax, semantics, and solving technology. The process of inventing useful heuristics would highly profit from automatic support. This paper presents a novel approach to the automatic learning of such heuristics. We use Inductive Logic Programming (ILP) to learn declarative domain-specific heuristics from examples stemming from (near-)optimal answer sets of small but representative problem instances. Our experimental results indicate that the learned heuristics can improve solving performance and solution quality when solving larger, harder instances of the same problem.

1 Introduction

Answer Set Programming (ASP) [3, 12, 15, 22] is a declarative problem-solving approach applied successfully in many industrial and scientific domains. For large and complex problems, however, domain-specific heuristics may be needed to achieve satisfactory performance [7, 9].

Therefore, state-of-the-art ASP systems offer ways to integrate domain-specific heuristics in the solving process. An extension for WASP [1] facilitates external *procedural* heuristics consulted at specific points during the solving process via an API [7]. *Declarative* specifications of domain-specific heuristics in the form of so-called *heuristic directives* are supported by CLINGO [11, 13, 14] and ALPHA [6, 25].

However, such heuristics must be invented manually so far. Human domain experts and ASP experts are needed to invent suitable domain-specific heuristics. This paper presents a novel approach capable of learning basic declarative heuristics automatically.

Our core idea is to use Inductive Logic Programming (ILP) to learn declarative domain-specific heuristics from examples stemming from (near-)optimal answer sets of small but representative problem instances. These heuristics can then be used to improve solving performance and solution quality for larger, harder problem instances. Our experimental results are promising, indicating that this goal can be achieved.

After covering preliminaries in Section 2, we present our main contribution in Section 3. Section 4 presents experimental results, and Section 5 describes related work. Section 6 concludes the paper by giving an outlook on future work.

---

[*] This paper is a significant revision of a previous workshop paper [5]. Improvements include: Using FASTLAS instead of ILASP; creating the mode bias in a systematic way; using multiple examples in the learning task; learning "soft" in addition to "hard" heuristics; experimentally comparing learned heuristics to CLINGO's built-in heuristics as well as human-made heuristics; updated related work.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 129–140, doi:10.4204/EPTCS.385.14

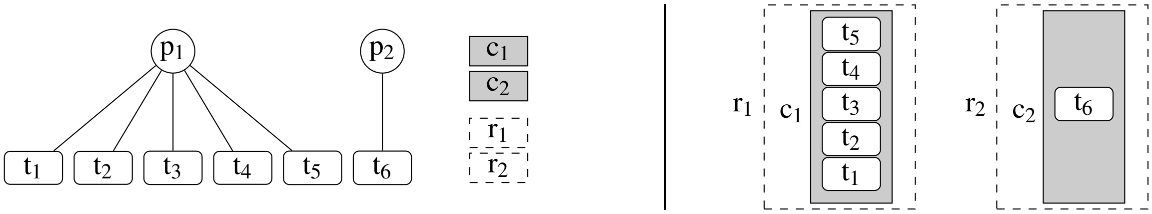


Figure 1: Sample HRP instance (left) and one of its solutions (right) [6]

2 Preliminaries

In this section, we introduce a running example and cover preliminaries on domain-specific heuristics in ASP and inductive learning in ASP. We assume familiarity with ASP and refer to [3, 12, 15, 22] for detailed introductions.

2.1 Running Example: The House Reconfiguration Problem (HRP)

The House Reconfiguration Problem (HRP) [10] is an abstracted version of industrial (re)configuration problems, e.g., rack configuration. A complete description is available from the ASP Challenge 2019,[1] and an encoding is available in Anna Ryabokon's PhD thesis [23].

Formally, HRP is defined as a modification of the House Configuration Problem (HCP) [6].

Definition 1 (HCP). *The input for the* House Configuration Problem (HCP) *is given by four sets of constants P, T, C, and R representing persons, things, cabinets, and rooms, respectively, and an ownership relation $PT \subseteq P \times T$ between persons and things.*

The task is to find an assignment of things to cabinets $TC \subseteq T \times C$ and cabinets to rooms $CR \subseteq C \times R$ such that: (1) each thing is stored in a cabinet; (2) a cabinet contains at most five things; (3) every cabinet is placed in a room; (4) a room contains at most four cabinets; and (5) a room may only contain cabinets storing things of one person.

Definition 2 (HRP). *The input for the* House Reconfiguration Problem (HRP) *is given by an HCP instance $H = \langle P,T,C,R,PT \rangle$, a legacy configuration $\langle TC',CR' \rangle$, and a set of things $T' \subseteq T$ that are defined as "long" (all other things are "short").*

The task is then to find an assignment of things to cabinets $TC \subseteq T \times C$ and cabinets to rooms $CR \subseteq C \times R$, that satisfies all requirements of HCP as well as the following ones: (1) a cabinet is either small or high; (2) a long thing can only be put into a high cabinet; (3) a small cabinet occupies 1 and a high cabinet 2 of 4 slots available in a room; (4) all legacy cabinets are small.

The sample HRP instance shown in Fig. 1 comprises two cabinets, two rooms, five things belonging to person p_1, and one thing belonging to person p_2. A legacy configuration is empty, and all things are small. In a solution, the first person's things are placed in cabinet c_1 in the first room, and the thing of the second person is in cabinet c_2 in the second room. For this sample instance, a solution of HRP corresponds to a solution of HCP [6].

Instances use the following predicates: `cabinetDomainNew/1` and `roomDomainNew/1` define potential cabinet and potential rooms; `thingLong/1` defines which things are long; and `legacyConfig/1` defines all the other data in the legacy configuration, e.g., `legacyConfig(personTOthing(p1,t1))`

---

[1] `https://sites.google.com/view/aspcomp2019/problem-domains`

defines that person `p1` owns thing `t1`, and `legacyConfig(roomTOcabinet(r1,c1))` specifies one tuple in the legacy assignment of cabinets to rooms.

For the present work, unique predicates have been used instead of predicates with function terms, e.g., `legacyConfig_personTOthing(p1,t1)` instead of `legacyConfig(personTOthing(p1,t1))`.

The main two choice rules guessing the assignment of things to cabinets and the assignment of cabinets to rooms look as follows:

```
{ cabinetTOthing(C,T) } :- cabinetDomain(C), thing(T).
{ roomTOcabinet(R,C) }  :- roomDomain(R),  cabinet(C).
```

To define the domains of cabinets and rooms as the union of existing objects and newly available identifiers, the encoding also contains the following rules:

```
cabinetDomain(C) :- cabinetDomainNew(C).
cabinetDomain(C) :- legacyConfig_cabinet(C).
roomDomain(R)    :- roomDomainNew(R).
roomDomain(R)    :- legacyConfig_room(R).
```

Instances may optionally include atoms of various predicates to define *costs* for specific actions such as placing a thing in a cabinet, placing a cabinet in a room, reusing an existing placement of a thing in a cabinet, reusing an existing placement of a cabinet in a room, removing a thing from a cabinet, removing a cabinet from a room, etc. These costs are then determined based on the difference between solution $\langle TC, CR \rangle$ and legacy configuration $\langle TC', CR' \rangle$. A weak constraint in the encoding instructs the solver to minimise the costs.

Each available instance belongs to one of four instance classes [10, 23]: *Empty* (“ec”, empty legacy configuration); *long* (“lt”, some things are long); *new room* (“nr”, some cabinets have to be reallocated to new rooms); and *swap* (“ss”, only one person and a specific pattern of legacy configuration).

2.2 Domain-Specific Heuristics in ASP

To solve large instances of industrial problems, employing an ASP solver out-of-the-box may not be sufficient. Sophisticated encodings or solver tuning methods (such as portfolio solving) are common ways to deal with this issue.

Domain-specific heuristics are another way to speed-up answer set solving. They were even needed to achieve breakthroughs in solving industrial configuration problems with ASP [7].

Several approaches have been proposed to embed heuristic knowledge into the ASP solving process. HWASP [7] extends WASP [1] by facilitating external procedural heuristics consulted at specific points during the solving process via an API.

A declarative approach to formulating domain-specific heuristics in ASP is provided by CLINGO,[2] supporting `#heuristic` directives [11, 14]. Heuristic directives enable the declarative specification of weights determining atom and sign orders in a solver’s internal decision heuristics. An atom’s weight influences the order in which atoms are considered by the solver when making a decision. A sign modifier instructs whether the selected atom must be assigned true or false. Atoms with a higher weight are assigned a value before atoms with a lower weight.

In the syntax for (non-ground) heuristic directives in CLINGO (1), *ha* is an atom, *hB* is a conjunction of literals representing the heuristic body, and *w*, *p*, and *m* are terms [11].

$$\#heuristic\ \ ha : hB. \qquad [w@p, m] \tag{1}$$

---

[2] `https://potassco.org/clingo/`

The optional term p gives a preference between heuristic values for the same atom (preferring those with higher p). The term m specifies the type of heuristic information and can take the following values: `sign`, `level`, `true`, `false`, `init` and `factor`. For instance, heuristics for m=`init` and m=`factor` allow modifying initial and actual atom scores evaluated by the solver's decision heuristics (e.g., VSIDS). The m=`sign` modifier forces the decision heuristics to assign an atom ha a specific sign, i.e., true or false, and m=`level` allows for the definition of an order in which the atoms are assigned—the larger the value of w, the earlier an atom must be assigned. Finally, m=`true` specifies that a should be assigned true with weight w if hB is satisfied, and m=`false` is the analogue heuristics that assigns a false.

A new approach implemented in the lazy-grounding ASP system ALPHA,[3] based on the CLINGO approach, has introduced novel semantics for heuristic directives aimed at non-monotonic heuristics [6].

2.3 Inductive Learning in ASP

Inductive Logic Programming (ILP) is an approach to learning a program that explains a set of examples given some background knowledge. FASTLAS[4] [19, 21] is a system capable of learning Answer Set Programs.[5]

FASTLAS operates on a *learning task*, which consists of three components [19,21]: The *background knowledge B* (an ASP program already known before learning), the *mode bias M* (that expresses which ASP programs can be learned), and the *examples E* (which specify properties the learned program must satisfy). When the properties specified by a particular example in E are satisfied, the example is said to be *covered*.

FASTLAS finds a program (often called a hypothesis) H such that $B \cup H$ covers every example in E (or, if the examples are considered *noisy*, such that the total penalty of non-covered examples is minimised) [21]. H is an element of the search space defined by M.

The syntax used to define learning tasks for FASTLAS follows the one used by ILASP (cf. [16,20]).

2.3.1 Mode Bias

The mode bias consists of a set of *mode declarations* of types `#modeh` and `#modeb`, specifying what the heads and bodies of learned rules may look like, respectively. A *placeholder* is a term `var(t)` or `const(t)` for some constant term `t`. Such placeholders can be replaced by any variable or constant (respectively) of *type* `t` [21].

As a simple example, Listing 1 shows part of the mode bias for a learning task for the HRP.

Listing 1: Part of the mode bias for HRP

```
#modeh(cabinetTOthing(var(cabinetDomain),var(thing))).
#modeb(cabinetDomain(var(cabinetDomain))).
#modeb(thing(var(thing))).
```

The first mode declaration specifies that the binary predicate `cabinetTOthing` can be used in the head of rules and that its terms are of variable types `cabinetDomain` and `thing` (in this order). The other two mode declarations specify which predicates can occur in the bodies of learned rules. Note that the same terms may be used in learned rules wherever the same placeholders are used.

---

[3] `https://github.com/alpha-asp/Alpha`

[4] `https://spike-imperial.github.io/FastLAS/`

[5] FASTLAS is not the only ILP system in the ASP setting. It was chosen for this work because of its speed and recency. ILASP [20], on the other hand, has successfully been used in a previous version of the paper [5].

Thus, the rule space defined by the mode bias given in Listing 1 consists of the following rule:

```
cabinetTOthing(V0,V1) :- cabinetDomain(V0), thing(V1).
```

FastLAS (contrary to ILASP [18, 20]) uses strict types, which need to be defined [19]. Thus, the rules in Listing 2 are needed in the background knowledge to make the mode bias from Listing 1 work.

Listing 2: Definition of some strict types for HRP

```
cabinetDomain(C) :- cabinetDomainNew(C).
cabinetDomain(C) :- legacyConfig_cabinet(C).
thing(T) :- thingLong(T).
thing(T) :- thingShort(T).
thing(T) :- legacyConfig_thing(T).
```

2.3.2 Examples

A positive example is given by a `#pos` statement, and a negative example by a `#neg` statement [16, 20]. Each example consists of several components:[6] an optional example identifier, a set of ground atoms called *inclusions*, a set of ground atoms called *exclusions*, and an optional set of rules (usually just facts) called *context*.

A positive example is covered iff there exists at least one answer set for $B \cup H$ that contains all of the inclusions and none of the exclusions. A negative example states that there must *not* exist an answer set that contains all of its inclusions and none of its exclusions [16, 20].

The *context* is the problem instance to which the inclusions and exclusions refer (considering the usual distinction between unvarying problem encoding and problem instances specified by facts) [16, 20].

Listing 3 shows a simplified example for HRP stating that `cabinetTOthing(1,2)` shall be true for the problem instance in which `cabinetDomainNew(1)` and `thing(2)` are true. The identifier of this example is `ex1`.

Listing 3: A simplified example in a learning task for HRP

```
#pos(ex1,
 { cabinetTOthing(1,2) },            % inclusions
 { },                                % exclusions
 { cabinetDomainNew(1). thing(2). } % context
).
```

3 Inductive Learning of Domain-Specific Heuristics

This section presents the main contribution of this paper—our approach to the inductive learning of domain-specific heuristics for ASP.

The basic idea is to solve one or more small but representative instances of a problem, use the resulting answer sets as positive examples for inductive learning, learn a set of definite rules, and transform the learned rules into declarative heuristic directives in the form of Eq. (1) presented in Section 2.2. These heuristics can then be used to speed up solving larger/harder instances of the same problem.

The learning task for FastLAS is defined in our approach in the way described in the following paragraphs.

---

[6] Currently, we do not use penalties.

Head mode declarations. Predicates that appear in the problem encoding in the heads of choice elements in the heads of choice rules[7] are used to create `#modeh` declarations. The placeholders are created in the following way:

- If the rule body and the choice condition together are (indirectly via rules) completely determined by the problem instance, use the body/condition predicate name as a strict type name and include the defining rules in the background knowledge. For example, from the choice rule in line 1 in Listing 4, the mode bias in line 2 in Listing 5 along with the background knowledge in lines 11–12 in Listing 5 is created.
- If the head predicate depends on another predicate that is used as a `#modeh` predicate itself, use the underlying strict type. Example: Line 2 in Listing 4, lines 3–4 in Listing 5.
- To define placeholders for non-unary predicates, define strict types for each argument in a deterministic way, appending `_arg1` etc. to the predicate name, along with corresponding rules. Example: Line 3 in Listing 4, lines 5–6 and 13–14 in Listing 5.

Body mode declarations. All other predicates appearing in the problem encoding are used to create `#modeb` declarations. Placeholders are directly built from the predicate names for unary predicates, and with suffixes such as `_arg1` etc. as above for non-unary predicates. As an example, cf. Listing 4 and lines 8–9 in Listing 5.

Background knowledge. The background knowledge consists only of rules needed to define strict types as described above.

Examples. As positive examples for learning, answer sets for small but representative problem instances are used (one answer set per problem instance, yielding one example per problem instance). In case the underlying problem is an optimisation problem (like the HRP described in Section 2.1), we propose to use (near-)optimal answer sets for this process. The (yet unproven) hypothesis is that learning from better answer sets yields better heuristics.

We use context-dependent examples; the context is given by the problem instance. The set of inclusions corresponds to the answer set filtered to cover only the predicates appearing in `#modeh`, and the set of exclusions is empty.

3.1 Learning in the House Reconfiguration Problem

Let us re-consider the running example from Section 2.1, the House Reconfiguration Problem (HRP). As representative problem instances, the smallest instance of each of the four instance classes was used. A near-optimal[8] answer set for each of these instances was computed by CLINGO [13].

We built a learning task from the full HRP encoding and the four representative instances as described at the beginning of Section 3. The entire mode bias contains 16 `#modeh` declarations and 32 `#modeb` declarations and is too large to be shown here. Listing 5 shows the parts of the learning task that are created for the choice rules in Listing 4.

---

[7] More formally: The head of a choice rule contains a collection of choice elements, each of the form a: $l_1, \ldots, l_k$ [4]; the predicates used for a are the ones that are used to create `#modeh` declarations.

[8] CLINGO was used to find the best solution that could be found with human-made domain-specific heuristics within 10 minutes; for some instances, optimality could not be proven within several days of search.

Listing 4: Some choice rules from the HRP encoding

```
{ cabinet(C) } :- cabinetDomain(C).
1 = {cabinetHigh(C); cabinetSmall(C)} :- cabinet(C).
1 = {reuse_cabinetTOthing(C,T); delete_cabinetTOthing(C,T)} :-
    legacyConfig_cabinetTOthing(C,T).
```

Listing 5: Part of the learning task for HRP

```
% HEAD MODE DECLARATIONS:
#modeh(cabinet(var(cabinetDomain))).
#modeh(cabinetHigh(var(cabinetDomain))).
#modeh(cabinetSmall(var(cabinetDomain))).
#modeh(reuse_cabinetTOthing(var(legacyConfig_cabinetTOthing_arg1),
    var(legacyConfig_cabinetTOthing_arg2))).
#modeh(delete_cabinetTOthing(var(legacyConfig_cabinetTOthing_arg1),
    var(legacyConfig_cabinetTOthing_arg2))).
% BODY MODE DECLARATIONS:
#modeb(cabinetDomain(var(cabinetDomain))).
#modeb(legacyConfig_cabinetTOthing(var(legacyConfig_cabinetTOthing_arg1),
    var(legacyConfig_cabinetTOthing_arg2))).
% BACKGROUND KNOWLEDGE:
cabinetDomain(C) :- cabinetDomainNew(C).
cabinetDomain(C) :- legacyConfig_cabinet(C).
legacyConfig_cabinetTOthing_arg1(C) :- legacyConfig_cabinetTOthing(C,T).
legacyConfig_cabinetTOthing_arg2(T) :- legacyConfig_cabinetTOthing(C,T).
```

Results. The following rules form the hypothesis learned by FASTLAS:[9]

```
cabinet(V0)        :- cabinetDomain(V0).
cabinetHigh(V0)    :- cabinetDomain(V0).
cabinetSmall(V0)   :- cabinetDomain(V0).
room(V0)           :- roomDomain(V0).
roomTOcabinet(V0,V1)  :- roomDomain(V0), cabinetDomain(V1).
cabinetTOthing(V0,V1) :- cabinetDomain(V0), thing(V1).
reuse_room(V0)        :- legacyConfig_room(V0).
reuse_cabinet(V0)     :- legacyConfig_cabinet(V0).
reuse_personTOroom(V0,V1)    :- legacyConfig_personTOroom(V0,V1).
reuse_roomTOcabinet(V0,V1)   :- legacyConfig_roomTOcabinet(V0,V1).
delete_roomTOcabinet(V0,V1)  :- legacyConfig_roomTOcabinet(V0,V1).
reuse_cabinetTOthing(V0,V1)  :- legacyConfig_cabinetTOthing(V0,V1).
delete_cabinetTOthing(V0,V1) :- legacyConfig_cabinetTOthing(V0,V1).
```

FASTLAS needed approximately 3 minutes to come up with this result on the author's personal computer.

As a next step, we transformed these rules to heuristics in the form of Eq. (1) by using the head of each rule as *ha* and the body as *hB*. Syntactically, this replaces `:-` with `:` in each rule and adds an appropriate annotation at the end of each resulting heuristic directive. The learned heuristics instruct the solver to choose the heads of most–but not all–choice rules.

---

[9] FASTLAS was called with parameter `--force-safety` to enforce learned rules to be save also without artificially added strict type atoms, and those atoms have been stripped from the learned rules in postprocessing where they were redundant. With this learning task, it does not seem to make a difference whether FASTLAS is called with `--opl` or `--nopl`. Rules have been re-ordered manually to improve readability.

For the annotation, we tried two approaches. The first annotation used for all heuristic directives is `[1,true]`. The idea is to instruct the solver to make the heads of the heuristics true, since the positive examples consist of atoms that are true in answer sets. And since we don't have any information on prioritising the heuristics at this point, all get the same weight 1.

The second approach uses the annotation `[2,factor]`. By this, the solver will multiply the atom scores of its domain-independent heuristics by the factor 2 for the atoms indicated by the heuristic directives. The idea here is to gently steer the solver into the right direction without enforcing any decisions.

4 Experimental Results

To test the effects of the learned heuristics, we used CLINGO version 5.6.2 to solve all available HRP instances (94 in number) with and without the learned heuristics. Additionally, other (built-in and human-made) heuristics were used in the experiments for comparison. The HRP instances stem from previous experiments [6]. These instances were generated in the pattern of the original instances [10]. This pattern represents four different reconfiguration scenarios encountered in practice, and the instances are abstracted real-world instances. Our instances are considerably larger than the original ones, though (ranging up to 800 things, while the original instances used at most 280 things).

Each of the machines used to run the experiments ran Ubuntu 22.04.2 LTS Linux and was equipped with two Intel® Xeon® E5-2650 v4 @ 2.20GHz CPUs with 12 cores. Hyperthreading was disabled and the maximum CPU frequency was set to 2.90GHz. Scheduling of benchmarks was done with Slurm[10] version 21.08.5. Runsolver[11] v3.4.1 was used to limit time consumption to 10 minutes per instance and memory to 20 GiB. Care was taken to avoid side effects between CPUs, e.g., by requesting exclusive access to an entire machine for each benchmark from Slurm.

CLINGO was instructed to search for the optimal answer set in its default configuration, given an encoding including a weak constraint. After 10 minutes per instance, search was aborted and the optimisation value of the best solution found so far was recorded. The solver was used in the following five configurations:

plain: the plain encoding without any domain-specific heuristics

learned (hard): the learned heuristics with annotation `[1,true]`, i.e., first assigning true on all atoms determined by the learned heuristics

learned (soft): the learned heuristics with annotation `[2,factor]`, i.e., modifying the atom scores of the solver-internal heuristics by the factor 2 for all atoms determined by the learned heuristics

built-in: CLINGO's built-in heuristics `--dom-mod=false,opt`, i.e., preferring atoms being optimised

human-made: the human-made heuristics introduced by Comploi-Taupe et al. [6].

Table 1 shows the achieved optimisation values and the relative improvement when using the learned heuristics for all 30 instances that could be solved in any solver configuration. For the other 64 instances, no answer set could be found in any configuration; therefore, they are not included in the table.

The first column shows the instance identifier. The first two characters of each identifier refer to one of the four instance classes of the HRP (cf. Section 2.1). The numeric part of the identifier increases with increasing instance size.

---

[10] https://slurm.schedmd.com/

[11] https://github.com/utpalbora/runsolver

Table 1: Experimental results: Achieved optimisation values without and with various heuristics, and relative improvement (positive percentage, boldface) or deterioration (negative percentage, italics)

Instance	**plain**	**learned (hard)**	**learned (soft)**	**built-in**	**human-made**
ec-0001	100	*125 (-25%)*	100 (0%)	*∞ (−∞)*	100 (0%)
ec-0002	125	*195 (-56%)*	125 (0%)	*∞ (−∞)*	125 (0%)
ec-0003	155	*230 (-48%)*	**150 (3%)**	*∞ (−∞)*	**150 (3%)**
ec-0004	180	*290 (-61%)*	**175 (3%)**	*∞ (−∞)*	**175 (3%)**
ec-0005	230	*335 (-46%)*	**200 (13%)**	*∞ (−∞)*	**200 (13%)**
ec-0006	270	*675 (-150%)*	**225 (17%)**	*∞ (−∞)*	**225 (17%)**
ec-0007	310	*550 (-77%)*	**250 (19%)**	*∞ (−∞)*	**250 (19%)**
ec-0008	350	*1220 (-249%)*	**275 (21%)**	*∞ (−∞)*	**275 (21%)**
ec-0009	385	*1250 (-225%)*	*505 (-31%)*	*∞ (−∞)*	**300 (22%)**
ec-0010	1045	*1585 (-52%)*	1045 (0%)	*∞ (−∞)*	**325 (69%)**
ec-0011	1340	*1750 (-31%)*	**1325 (1%)**	*∞ (−∞)*	**350 (74%)**
lt-0001	337	*980 (-191%)*	*460 (-36%)*	*∞ (−∞)*	**192 (43%)**
lt-0002	1347	**1138 (16%)**	**1330 (1%)**	*∞ (−∞)*	**224 (83%)**
lt-0003	1589	**1302 (18%)**	**1527 (4%)**	*∞ (−∞)*	**256 (84%)**
lt-0004	1848	*∞ (−∞)*	**1798 (3%)**	*∞ (−∞)*	**288 (84%)**
lt-0005	2103	*∞ (−∞)*	**1979 (6%)**	*∞ (−∞)*	**320 (85%)**
lt-0006	2382	*∞ (−∞)*	**2357 (1%)**	*∞ (−∞)*	**352 (85%)**
nr-0001	1297	**1064 (18%)**	**1283 (1%)**	*∞ (−∞)*	**256 (80%)**
nr-0002	1124	*1194 (-6%)*	*1403 (-25%)*	*∞ (−∞)*	**288 (74%)**
nr-0003	1711	**1324 (23%)**	**1550 (9%)**	*∞ (−∞)*	**320 (81%)**
nr-0004	1023	*∞ (−∞)*	*1739 (-70%)*	*∞ (−∞)*	**352 (66%)**
nr-0005	2008	*∞ (−∞)*	**1870 (7%)**	*∞ (−∞)*	**384 (81%)**
nr-0006	∞	∞	**2173 (100%)**	∞	**416 (100%)**
nr-0007	∞	∞	∞	∞	**448 (100%)**
ss-0001	46	*176 (-283%)*	**4 (91%)**	**4 (91%)**	**4 (91%)**
ss-0002	209	**207 (1%)**	**8 (96%)**	**4 (98%)**	**4 (98%)**
ss-0003	726	**242 (67%)**	*843 (-16%)*	**4 (99%)**	**4 (99%)**
ss-0004	1580	**707 (55%)**	*1649 (-4%)*	**4 (100%)**	**4 (100%)**
ss-0005	1785	**840 (53%)**	**1655 (7%)**	**4 (100%)**	**4 (100%)**
ss-0006	2313	**1272 (45%)**	**2124 (8%)**	**4 (100%)**	**4 (100%)**

The second column contains the achieved optimisation values in the "plain" solver configuration, i.e., without domain-specific heuristics. The remaining columns contain the achieved optimisation values using the other solver configurations ("learned (hard)", "learned (soft)", "built-in", and "human-made"). In all columns, the symbol ∞ is used instead of an optimisation value when no answer set could be found within the time limit of 10 minutes.

The columns for non-plain solver configurations additionally show, within parentheses, the respective change in the optimisation value when using heuristics relative to solving without domain-specific heuristics. A positive percentage signifies an improvement (printed in boldface), and negative values indicate a deterioration (printed in italics). The value 100% is used in cases where an answer set could be found only when using one of the heuristics and the value $-\infty$ is used when an answer set could be found only without heuristics.

The learned heuristics seem to have positive effects even though they are (still) straightforward. Positive effects could be seen especially when the heuristics were applied in a "soft" way, weighting but not replacing the domain-independent search heuristics, leading to improved solution quality in 20 cases and to a deterioration in 6 cases. However, improvements varied strongly between different instances. Furthermore, results are sensitive to the chosen time-out. For example, we observed stronger improvements (on fewer solved instances) when experimenting with a time-out of three minutes instead of ten.

Surprisingly, the solver benefited only rarely from learned "hard" heuristics as well as the built-in heuristics preferring atoms being optimised. Human-made heuristics still outperformed learned heuristics by a great degree, which could show untapped potential for our approach.

Besides our experiments with CLINGO, we also experimented with the lazy-grounding ASP system ALPHA [25]. This system accepts heuristic directives in a slightly different syntax [6]. Without domain-specific heuristics, ALPHA could solve none of the HRP instances under consideration. The heuristics learned by our approach enabled ALPHA to solve three instances (without optimisation, which is not yet supported by ALPHA). Human-made heuristics (cf. [6]) enabled ALPHA to solve 58 of these instances on the same benchmarking infrastructure.

5 Related Work

Balduccini [2] has also presented an approach to learning domain-specific heuristics offline from representative instances. The basic idea, which is very different to our approach, is to record which choices are made in the path of a search tree that led to a solution and to use this information to compute probabilities for decisions on ground atoms. These probabilities are then used while solving other problem instances to reduce the likelihood of backtracks. The approach is restricted to DPLL-style solvers like SMODELS [24], and extending it to CDCL-based systems like CLINGO is mentioned as future work.

A similar approach is aimed at configuration problems encoded as constraint satisfaction problems (CSPs) [17].

Dodaro et al. [8] use deep learning to learn domain-specific heuristics for the graph colouring problem to be used by the solver WASP. As usual with deep learning, this approach requires huge numbers of training instances (in reported experiments, 210,000 instances were used for training, 60% of which have been randomly chosen to build the training set). The approach, including the integration in WASP's solving algorithm, is specific to the graph colouring problem. In contrast to the work presented in this paper, learned heuristics apply to ground ASP and don't offer a concise, declarative representation readable by humans.

6 Conclusions and Future Work

We have proposed a novel approach to inductively learning declarative specifications of domain-specific heuristics for ASP from answer sets of small but representative instances. Our approach employs the inductive learning system FASTLAS. Utilising an example representing a significant real-world configuration problem, we have demonstrated that simple heuristics can easily be learned.

Experimental results are promising, since the learned heuristics led to improved solution quality in many cases. The fact that so far, we have only learned very simple heuristics and those already led to significant improvements is encouraging. Future work will show whether our method can be extended to learn more complex heuristics that can improve solving performance and solution quality even further.

Acknowledgements. The author is grateful to the participants of the two workshops HYDRA 2022[12] and TAASP 2022[13] as well as the ICLP 2023 reviewers for their valuable suggestions, to the TU Wien for providing access to computational resources for running the experiments and to Tobias Geibinger for helping with using this infrastructure, and to Mark Law for helping with using FASTLAS.

References

[1] M. Alviano, G. Amendola, C. Dodaro, N. Leone, M. Maratea & F. Ricca (2019): *Evaluation of Disjunctive Programs in WASP*. In M. Balduccini, Y. Lierler & S. Woltran, editors: *Logic Programming and Nonmonotonic Reasoning - 15th International Conference, Philadelphia, PA, USA, June 3-7, 2019, Proceedings*, *Lecture Notes in Computer Science* 11481, Springer, pp. 241–255, doi:`10.1007/978-3-030-20528-7_18`.

[2] M. Balduccini (2011): *Learning and using domain-specific heuristics in ASP solvers*. *AI Commun.* 24(2), pp. 147–164, doi:`10.3233/AIC-2011-0493`.

[3] C. Baral (2003): *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, doi:`10.1017/CBO9780511543357`.

[4] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca & T. Schaub (2020): *ASP-Core-2 Input Language Format*. *Theory Pract. Log. Program.* 20(2), pp. 294–309, doi:`10.1017/S1471068419000450`.

[5] R. Comploi-Taupe (2022): *Towards Inductive Learning of Domain-Specific Heuristics for ASP*. In: *Joint Proceedings of the 1st International Workshop on HYbrid Models for Coupling Deductive and Inductive ReAsoning (HYDRA 2022) and the 29th RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion (RCRA 2022) co-located with LPNMR 2022*, *CEUR Workshop Proceedings* 3281, pp. 21–33. Available at `https://ceur-ws.org/Vol-3281/paper3.pdf`.

[6] R. Comploi-Taupe, G. Friedrich, K. Schekotihin & A. Weinzierl (2023): *Domain-Specific Heuristics in Answer Set Programming: A Declarative Non-Monotonic Approach*. *J. Artif. Intell. Res.* 76, pp. 59–114, doi:`10.1613/jair.1.14091`.

[7] C. Dodaro, P. Gasteiger, N. Leone, B. Musitsch, F. Ricca & K. Schekotihin (2016): *Combining Answer Set Programming and domain heuristics for solving hard industrial problems (Application Paper)*. *Theory Pract. Log. Program.* 16(5-6), pp. 653–669, doi:`10.1017/S1471068416000284`.

[8] C. Dodaro, D. Ilardi, L. Oneto & F. Ricca (2022): *Deep Learning for the Generation of Heuristics in Answer Set Programming: A Case Study of Graph Coloring*. In G. Gottlob, D. Inclezan & M. Maratea, editors: *Logic Programming and Nonmonotonic Reasoning - 16th International Conference, Genova, Italy, September 5-9, 2022, Proceedings*, *Lecture Notes in Computer Science* 13416, Springer, pp. 145–158, doi:`10.1007/978-3-031-15707-3_12`.

---

[12] `https://hydra2022.demacs.unical.it/`

[13] `http://www.kr.tuwien.ac.at/events/taasp22/`

[9] A.A. Falkner, G. Friedrich, K. Schekotihin, R. Taupe & E.C. Teppan (2018): *Industrial Applications of Answer Set Programming*. Künstliche Intell. 32(2-3), pp. 165–176, doi:`10.1007/s13218-018-0548-6`.

[10] G. Friedrich, A. Ryabokon, A.A. Falkner, A. Haselböck, G. Schenner & H. Schreiner (2011): *(Re)configuration using Answer Set Programming*. In K.M. Shchekotykhin, D. Jannach & M. Zanker, editors: Proceedings of the IJCAI 2011 Workshop on Configuration, Barcelona, Spain, July 16, 2011, CEUR Workshop Proceedings 755, CEUR-WS.org, pp. 17–24. Available at `http://ceur-ws.org/Vol-755/paper03.pdf`.

[11] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, S. Thiele & P. Wanko (2019): *Potassco guide version 2.2.0*. Available at `https://github.com/potassco/guide/releases/tag/v2.2.0`.

[12] M. Gebser, R. Kaminski, B. Kaufmann & T. Schaub (2012): *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers.

[13] M. Gebser, R. Kaminski, B. Kaufmann & T. Schaub (2019): *Multi-shot ASP solving with clingo*. Theory Pract. Log. Program. 19(1), pp. 27–82, doi:`10.1017/S1471068418000054`.

[14] M. Gebser, B. Kaufmann, J. Romero, R. Otero, T. Schaub & P. Wanko (2013): *Domain-Specific Heuristics in Answer Set Programming*. In M. desJardins & M.L. Littman, editors: Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA, AAAI Press, pp. 350–356, doi:`10.1609/aaai.v27i1.8585`.

[15] M. Gelfond & Y. Kahl (2014): *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. Cambridge University Press, New York, NY, USA, doi:`10.1017/CBO9781139342124`.

[16] ILASP Limited (2022): *The ILASP manual*. Available at `https://doc.ilasp.com/`.

[17] D. Jannach (2013): *Toward Automatically Learned Search Heuristics for CSP-encoded Configuration Problems – Results from an Initial Experimental Analysis*. In: Configuration Workshop, pp. 9–13.

[18] M. Law (2022): *Conflict-Driven Inductive Logic Programming*. Theory and Practice of Logic Programming, doi:`10.1017/S1471068422000011`.

[19] M. Law, A. Russo, E. Bertino, K. Broda & J. Lobo (2020): *FastLAS: Scalable Inductive Logic Programming Incorporating Domain-Specific Optimisation Criteria*. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence, New York, NY, USA, February 7-12, 2020, AAAI Press, pp. 2877–2885, doi:`10.1609/aaai.v34i03.5678`.

[20] M. Law, A. Russo & K. Broda (2020): *The ILASP system for Inductive Learning of Answer Set Programs*. CoRR abs/2005.00904, doi:`10.48550/arXiv.2005.00904`. arXiv:`2005.00904`.

[21] M. Law, A. Russo, K. Broda & E. Bertino (2021): *Scalable Non-observational Predicate Learning in ASP*. In Z. Zhou, editor: Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021, ijcai.org, pp. 1936–1943, doi:`10.24963/ijcai.2021/267`.

[22] V. Lifschitz (2019): *Answer Set Programming*. Springer, doi:`10.1007/978-3-030-24658-7`.

[23] A. Ryabokon (2015): *Knowledge-based (Re)configuration of Complex Products and Services*. Ph.D. thesis, Alpen-Adria-Universität Klagenfurt. Available at `http://netlibrary.aau.at/urn:nbn:at:at-ubk:1-26431`.

[24] T. Syrjänen & I. Niemelä (2001): *The Smodels System*. In T. Eiter, W. Faber & M. Truszczynski, editors: Logic Programming and Nonmonotonic Reasoning, 6th International Conference, Vienna, Austria, September 17-19, 2001, Proceedings, Lecture Notes in Computer Science 2173, Springer, pp. 434–438, doi:`10.1007/3-540-45402-0_38`.

[25] A. Weinzierl (2017): *Blending Lazy-Grounding and CDNL Search for Answer-Set Solving*. In M. Balduccini & T. Janhunen, editors: Logic Programming and Nonmonotonic Reasoning - 14th International Conference, Espoo, Finland, July 3-6, 2017, Proceedings, Lecture Notes in Computer Science 10377, Springer, pp. 191–204, doi:`10.1007/978-3-319-61660-5_17`.

Natlog: Embedding Logic Programming into the Python Deep-Learning Ecosystem

Paul Tarau
University of North Texas
paul.tarau@unt.edu

Driven by expressiveness commonalities of Python and our Python-based embedded logic-based language Natlog, we design high-level interaction patterns between equivalent language constructs and data types on the two sides.

By directly connecting generators and backtracking, nested tuples and terms, coroutines and first-class logic engines, reflection and meta-interpretation, we enable logic-based language constructs to access the full power of the Python ecosystem.

We show the effectiveness of our design via Natlog apps working as orchestrators for JAX and Pytorch pipelines and as DCG-driven GPT3 and DALL.E prompt generators.

Keyphrases: *embedding of logic programming in the Python ecosystem, high-level inter-paradigm data exchanges, coroutining with logic engines, logic-based neuro-symbolic computing, logic grammars as prompt-generators for Large Language Models, logic-based neural network configuration and training.*

1 Introduction

Turing-complete programming languages win users based on how easily one can make computers do things, for which we will use here *expressiveness* as an umbrella concept. Declarative programming has been seen over the years as an *expressiveness enhancer*, assuming that telling *what* rather than *how* to do things makes it easier to achieve a desired outcome. With this view of declarative programming in mind, besides competition from today's functional programming languages and proof assistants, all with strong declarative claims, logic-based languages face an even stiffer competition from the more radical approach coming from deep learning.

To state it simply, this manifests as replacement of rule-based, symbolic encoding of intelligent behaviors via machine learning, including unsupervised learning among which transformers [11] trained on large language models have achieved outstanding performance in fields ranging from natural language processing to computational chemistry and image processing. For instance, results in natural language processing with prompt-driven generative models like GPT3 [2] or prompt-to-image generators like DALL.E [5] or Stable Diffusion [12] have outclassed similarly directed symbolic efforts. In fact, it is quite hard to claim that a conventional programming language (including a logic-based one) is as declarative as entering a short prompt sentence describing a picture and getting it back in a few seconds.

Thus it is becoming clearer as time goes by, that ownership of the declarative umbrella is slowly transitioning to deep neural networks-based machine learning tools that replace human coding (including that done in declarative languages) with models directly extracted from labeled and more and more often from raw, unlabeled data.

This forces us to question, as lucidly as possible, what contribution logic-based reasoning and more concretely logic-based language design can bring to this fast evolving ecosystem, and what language features are needed to enable them.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 141–154, doi:10.4204/EPTCS.385.15

Language features facilitating interoperation of a logic-based language with the Python deep-learning ecosystem

Rather than fighting it, we will explore in this paper several ways to join and enhance this strongly disruptive ecosystem. Toward this end, we will design logic-based language features that will not only facilitate embedding of logic-based reasoning in the Python but also simplify, syntactically and semantically the interaction with the deep learning tools that make the ecosystem itself appealing.

While aware that more accurate (but also more intricate) solutions might come out in the near future as part of emerging neuro-symbolic computing research[1] we will apply our findings to two mechanisms of interoperation with deep learning tools, of practical interest today, where the embedding of a logic programming language in the ecosystem brings a fresh approach.

Logic languages as orchestrators for deep learning architectures

Let's start by observing that the actual code base enabling most of today's deep learning systems is a convoluted, ungeneralizable mix of unstructured scripting, low level GPU-acceleration and linear-algebra libraries.

This suggests that there's plenty of room to enhance and clarify the design and implementation of the networks themselves in the declarative frameworks logic-based programming languages provide. To facilitate interoperation with the ecosystem of logic-based orchestrators for deep-learning tools, we will design a component composition framework implemented in a lightweight logic language embedded in Python.

Logic languages as prompt-generators and Large Language Model interaction refiners

The emergence of Large Language Models like GPT3 and their refinements like ChatGPT and text-to-image variants like DALL.E exhibits a strong dependency on an essentially declarative component: *prompt engineering*. Prompt engineering can be seen as a goal-driven language generation, a logically specifiable endeavor for which we will show that logic grammars are a natural match.

To make our research goals and related language design proposals experimentally testable, we have embedded in Python a lightweight Prolog-like language, Natlog. We refer to [9] for a description, in an early, proof-of-concept version, of its concrete syntactic and semantic features, while we will focus here on advanced use cases based on a fresh "from-scratch" reimplementation.

The paper is organized as follows. Section 2 introduces and motivates the key design ideas behind the Natlog embedded logic-based language, in particular Natlog's coroutining mechanisms using First-Class Logic Engines. Section 3 shows uses of Natlog scripts as orchestrators for designing, training and testing deep-learning systems in the JAX and Pytorch ecosystems. Section 4 discusses the use Natlog's logic grammars as prompt generators for text-to-text and text-to-image large language model neural networks. Section 5 discusses related work and section 6 concludes the paper.

Caveat emptor: As the paper is about interoperation of language constructs between Python and a Prolog-like logic programming language implemented in it, we assume that the reader is fluent in Prolog and Python. We also assume exposure to essential language constructs of today's very high-level programming languages as well as familiarity with deep learning frameworks and some key design and implementation decisions behind them.

---

[1] a fast evolving research field, thoroughly overviewed in [6]

2 Key Design Ideas behind Natlog, a Lightweight Prolog-dialect Embedded in Python

Our Natlog system has been originally introduced in [9], to which we refer to for syntax, semantics and low level implementation details. It is currently evolving as a fresh implementation[2], and it will be used as a testbed for the key ideas of this paper.

2.1 Prolog's semantics, but with a lighter syntax

While keeping Natlog's semantics the same as Prolog's LD-resolution, we have brought its syntax a step closer to natural language. In particular, we are not requiring predicate symbols to wrap parenthesized arguments or predicate symbols to be constants. As a quick glimpse at its syntactic simplifications, here is a short extract from the usual "family" program in Natlog syntax.

```
sibling of X S: parent of X P, parent of S P, distinct S X.

grand parent of X GP: parent of X P, parent of P GP.
```

```
ancestor of X A : parent of X  P, parent or ancestor P A.

parent or ancestor P P.
parent or ancestor P A : ancestor of P A.
```

2.2 A Quick Tour of Natlog's "Expressiveness Lifters"

Expressiveness is the relevant distinguishing factor between Turing-complete languages. It can be seen as a pillar of code development automation as clear and compact notation entails that more is delegated to the machine. At the same time, expressiveness enhancers need to be kept as simple as possible, with their users experience in mind.

2.2.1 A finite function API

Finite functions (tuples, lists, dictionaries, sets) are instrumental in getting things done with focus on the problem to solve rather than its representation in the language.

In Natlog they are directly borrowed from Python. They can be easily emulated in any Prolog system, but often with a different complexity than if they were natively implemented.

In an immutable form, as well as enabled with backtrackable and non-backtrackable updates, finite functions implemented as dynamic arrays and hash-maps offer a more flexible and semantically simpler alternative to reliance on Prolog's `assert` and `retract` family of built-ins.

2.2.2 Built-ins as functions or generators

Reversible code like in Prolog's classic `append/3` examples or the use of DCGs in both parsing and generation are nice and unique language features derived from the underlying LD-resolution semantics, but trying to lend reversibility and multi-mode uses to built-ins obscures code and hinders debugging.

---

[2] at `https://github.com/ptarau/natlog`, ready to install with "`pip3 install natlog`"

Keeping Natlog's built-ins uniform and predictable, while not giving up on flexibility, can be achieved by restricting them to:

- functions with no meaningful return like `print`, denoted in Natlog by prefixing their Python calls with "#".
- functions of N inputs returning a single output as the last argument of the corresponding predicate with $N+1$ arguments, denoted in Natlog by prefixing their calls with a backquote symbol "`". Note that this syntax, more generally, also covers Python's *callables* and in particular class objects acting as instance constructors.
- generators with N inputs yielding a series of output values on backtracking by binding the $N+1$-th argument of the corresponding predicate, denoted in Natlog by prefixing their call with two backquotes "``".

2.2.3 Interoperation with Python, as seen from a few Natlog library predicates

Interaction with Python's finite functions happens via function and generator calls. For instance, the predicate `argx` implements backtracking over all elements of a tuple or list by relaying on the Python generator `range` that iterates over all index positions from where `arg` picks an item to be unified with variable `X`.

```
argx I T X: `len T L, ``range 0 L I, `arg T I X.
```

Given Natlog's expected practical uses as a Python package, even when inside Natlog's REPL, answers are shown as the corresponding Python objects, given the one-to-one correspondence between terms and nested tuples and between variable binding and dictionaries. Note also that constructors like `dict` (and any other Python callables) are usable directly with the same syntax as function calls.

```
?- `dict ((one 1) (two 2) (three 3)) D?
ANSWER: {'D': {'one': 1, 'two': 2, 'three': 3}}

?- eq (a b c) T, argx I T X?
ANSWER: {'T': ('a', 'b', 'c'), 'I': 0, 'X': 'a'}
ANSWER: {'T': ('a', 'b', 'c'), 'I': 1, 'X': 'b'}
ANSWER: {'T': ('a', 'b', 'c'), 'I': 2, 'X': 'c'}
```

Natlog uses cons-lists like `(1 (2 (3 ())))` for the usual, unification-based list operations. A few built-in predicates support their conversion to/from Python tuples or lists:

```
to_tuple Xs T : `from_cons_list_as_tuple Xs T.
to_cons_list T Xs : `to_cons_list T Xs.
```

2.2.4 Reflecting metaprogramming constructs

In function and generator calls, Python's `eval` is used to map the Natlog name of a function or generator to its Python definition. However, to conveniently access object and class attributes, Natlog implements `setprop` and `getprop` relying directly on corresponding Python built-ins.

```
setprop O K V : #setattr O K V.
getprop O K V : `getattr O K V.
```

Similary, method calls are supported via the Python function `meth_call` as in the following stack manipulation API:

```
stack S : `list S.  % note the use of the callable empty list constructor
push S X : #meth_call S append (X).
pop S X : `meth_call S pop () X.
```

In fact, a method call has a surprisingly succinct Python definition, a testimony that *elegant metaprogramming constructs on the two sides make language interoperation unusually easy*.

```
def meth_call(o, f, xs):
    m = getattr(o, f)
    return m(*xs)
```

As the reader familiar with Python will notice, a method "`m`" is simply an attribute of an object "`o`", directly callable once it has been retrieved from its name "`f`".

These predicates are part of the Natlog library code in file `lib.nat`[3] that can be included as part of a Natlog script with help of the Python function `lconsult`[4].

2.2.5 Reflecting the type system

As the following examples show, Python's "`type`" built-in can be used to reflect and inspect Natlog's data-types.

```
?- `type (a b) T?
ANSWER: {'T': <class 'tuple'>}
?- `type a T1, `type b T2, eq T1 T2.
ANSWER: {'T1': <class 'str'>, 'T2': <class 'str'>}
?- `type X V, eq X a, `type X C?
ANSWER: {'X': 'a', 'V': <class 'natlog.unify.Var'>, 'C': <class 'str'>}
```

Note in the last example, that after unification, the type of a variable is dereferenced to the type of its binding.

2.3 Natlog's First Class Logic Engines

Constraint solvers bring to logic-based languages an automated coroutining mechanism when they suspend computations until more data is available and propagate constraints to the inner loops of SLD-derivations with impressive performance gains including on NP-complete problems. However this implicit coroutining mechanism does not reflect control on backtracking and does not expose the interpreter itself as a first-class object.

One can think about First Class Logic Engines as a way to ensure the *full meta-level reflection* of the execution algorithm. As a result, they enable on-demand computations in an engine rather than the usual eager execution mechanism of Prolog.

We will spend more time on them as we see them as "the path not taken" that can bring significant expressiveness benefits to logic-based languages, similarly to the way Python's `yield` primitive supports creation of user-defined generators and other compositional asynchronous programming constructs.

---

[3] at `https://github.com/ptarau/natlog/blob/main/natlog/natprogs/lib.nat`

[4] in file `https://github.com/ptarau/natlog/blob/main/natlog/natlog.py`

2.3.1 A First-class Logic Engines API

To obtain the full reflection of Natlog's multiple-answer generation mechanism, we will make fresh instances of the interpreter first-class objects.

A *logic engine* is a Natlog language processor reflected through an API that allows its computations to be controlled interactively from another *logic engine*.

This is very much the same thing as a programmer controlling Prolog's interactive toplevel loop: launch a new goal, ask for a new answer, interpret it, react to it. The exception is that it is not the programmer, but it is the program that does it!

We will next summarize the execution mechanism of Natlog's interoperating logic engines.

The predicate `eng AnswerPattern Goal Engine` works as follows:

- creates a new instance of the Natlog interpreter, uniquely identified by `Engine`
- shares code with the currently running program
- it is initialized with `Goal` as a starting point, but not started
- `AnswerPattern` ensures that answers returned by the engine will be instances of the pattern.

The predicate `ask Engine AnswerInstance` works as follows:

- tries to harvest the answer computed from `Goal`, as an instance of `AnswerPattern`
- if an answer is found, it is returned as `(the AnswerInstance)`, otherwise the atom `no` is returned
- it is used to retrieve successive answers generated by an Engine, on demand
- it is responsible for actually triggering computations in the engine

One can see this as transforming Natlog's backtracking over all answers into a deterministic stream of lazily generated answers.

Finally, the predicate `stop Engine` works as follows:

- stops the Engine, reclaiming the resources it has used
- ensures that `no` is returned for all future queries

In Natlog these predicates are implemented as built-ins.

2.3.2 The coroutining Mechanism Implemented by the Engine API

Natlog's yield operation: a key co-routining primitive The annotation "`^Term`" extends our coroutining mechanism by allowing answers to be *yield from arbitrary places* in the computation. It works as follows:

- it saves the state of the engine and transfers *control* and a *result* `Term` to its client
- the client will receive a copy of `Term` simply by using its `ask/2` operation
- an engine returns control to its client when initiating a yield operation as when a computed answer becomes available.

As implemented in Python, engines can be seen simply as a special case of generators that yield one answer at a time, on demand.

We will outline next, with help from a few examples, a few expressiveness improvements First Class Logic Engines can bring to a logic-based programming language.

2.3.3 An infinite Fibonacci stream with yield

Like in a non-strict functional language, one can create an infinite recursive loop from which values are yielded as the computation advances:

```
fibo N Xs : eng X (slide_fibo 1 1) E,  take N E Xs.

slide_fibo X Y :  with X + Y as Z,  ^X, slide_fibo Y Z.
```

Note that the infinite loop's results, when seen from the outside, show up as a stream of answers as if produced on backtracking. With help of the library predicate `take`, we extract the first 5 (seen as a Python dictionary with name "`X`" of the variable as a key and the nested tuple representation of Natlog's list as a value), as follows:

```
?- fibo 5 Xs?
ANSWER: {'Xs': (1, (1, (2, (3, (5, ())))))}
```

2.3.4 The `trust` operation

When the special atom `trust` is yielded, the goal that follows it replaces the goal of the engine, with all backtracking below that point discarded and all memory consumed so far made recoverable. As a practical consequence, infinite loops can work in constant space, even in the absence of last call optimization.

Using it, the predicate `loop` shows how to generate an infinite sequence of natural numbers.

```
loop N N.
loop N X : with N + 1 as M, ^trust loop M X.
```

```
? - loop 0 X?
ANSWER: {'X': 0}
ANSWER: {'X': 1}
...
```

3 Natlog as an Orchestrator for Neuro-Symbolic Deep Learning Systems

We will show next how the use of Natlog as an embedded logic-based scripting language can simplify the design and the execution of neural networks as well as making their internal logic easily understandable.

3.1 Accessing the namespaces of the Python packages

To ensure unimpeded access to relevant Python objects that can be as many as a few hundred for packages like JAX or Pytorch, we have devised a namespace sharing mechanism with the Python side. For objects visible in the original namespace of the "`natlog`" package this can be achieved by calling `eval` on the name of the function or generator. However, when Natlog itself is imported as a package (as in the JAX and Pytorch apps relying on it) we collect to a dictionary the names visible in the Python client importing Natlog in the app and then pass it to Natlog.

3.2 Interoperation with JAX

A natural partner to logic-based languages when interacting with deep learning systems is a declaratively designed neural-network like Google and DeepMind's JAX [1].

JAX is referentially transparent (no destructive assignments) and fully compositional, via a lazy application of matrix/tensor operations, automatic gradient computations and a just-in-time compilation function also represented as a first-class language construct.

The interaction with a logic-based programming language is facilitated by the lazy functional syntax of JAX (seen as an embedded sublanguage in Python).

After defining a set of data loaders, initialization functions and basic neural network layers, both learning and inference can be expressed in JAX as a composition of functions (more exactly as a *future* consisting of such a composition to be eventually activated after a compilation step).

In a Python-based logic language like Natlog, this orchestration process can be expressed as a set of Horn Clauses with logic variables bound to immutable JAX objects transferring inputs and outputs between neural predicates representing the network layers. JAX's high-level, referentially transparent matrix operations can be reflected safely as Natlog predicates with the result of the underlying N-argument function unified with their N+1-th extra argument. As such, they can be passed as bindings of logic variables between clause heads and clause bodies in an easy to understand, goal-driven design and execution model.

JAX's equivalents of Natlog's compound terms, called *pytrees*, hold arbitrary aggregates of data and can be differentiated with JAX's `grad` operator as a single unit.

Hyperparameter optimization searches can be naturally expressed as constraint-driven optimization processes.

The synergy between the declarative neural framework and the declarative logic orchestrator can also help with identifying the complex causal chains needed for debugging, optimizing the network architecture as well as with the explainability of both the design and the execution of the resulting neuro-symbolic system.

The following Natlog code snippet[5] generates a "deep xor" dataset, known to be unusually challenging even for deep neural nets.

```
xor 0 0 0.
xor 0 1 1.
xor 1 0 1.
xor 1 1 0.
```

The predicate `iter` recurses N times over the truth table of `xor` to obtain the truth table of size 2^N of X_1 *xor* X_2 *xor* ...X_n that we will use as our synthetic dataset.

```
iter N Op X Y: iter_op N Op () E 0 Y, to_tuple E X.
```

```
iter_op 0 _Op E E R R.
iter_op I Op  E1 E2 R1 R3 :
   when I > 0, with I - 1 as J,
   Op X R1 R2,
   with X + X as XX,    % x->2x-1 maps {0,1} into {-1,1} to facilitate
   with XX - 1 as X1,   % the work of the network's Linear Layers
   iter_op J Op (X1 E1) E2 R2 R3.
```

---

[5] see `https://github.com/ptarau/natlog/blob/main/apps/deepnat/natjax.nat` for more details

The dataset will be passed to JAX after conversion to tuples of tuples, that will eventually become JAX tensors.

```
dataset N Op Xss (Ys):
  findall (X Y) ( iter N Op X Y) XssYsList,
  to_pairs XssYsList XssList YsList,
  to_tuple XssList Xss,
  to_tuple YsList Ys.

to_pairs ()()()  . % note Prolog's [X|Y] becoming (X Y) in Natlog
to_pairs ((Xs Y)  Zss) (Xs Xss) (Y Ys) :
   to_pairs Zss Xss Ys.
```

We also generate, depending on the number of variables `N`, an appropriate list of hidden-layer sizes (via the predicate `hidden_sizes`) for a Multi-Layer Perceptron that we design in JAX to be trained on our synthetic dataset. We use the “’” marker to indicate Python calls to functions like `train_model` and `test_model` implemented in a companion Python file[6].

```
run N Op Seed Epochs Loss Acc LossT AccT:
   dataset N Op Xss Ys,
   split_dataset Xss Ys X Xt Y Yt,
   hidden_sizes N Sizes,
   #print hidden sizes are Sizes,
   `train_model X Y Sizes Epochs (Model LossFun), % training set
   `test_model Model LossFun X Y (Loss Acc),      % validation set
   `test_model Model LossFun Xt Yt (LossT AccT).  % testing set
```

Similar calls to pass the dataset to Python and to split it into “train” and “test” subsets, as well as seamless interaction with objects like JAX-arrays (seen by Natlog as constant symbols) are shown in the Natlog definition of `split_dataset`.

```
split_dataset Xss Ys Xtr Xt Ytr Yt:
   `array Xss X,
   `array Ys Y0,
   `transpose Y0 Y,
   `split X Y 0 0.1 (Xtr Xt Ytr Yt),
   show_sizes (X Y Xtr Xt Ytr Yt).
```

For instance, “`‘array Xss X`” converts a Natlog tuple of tuples `Xss` to a two-dimensional JAX array `X`, relying on the fact that JAX’s underlying `numpy` matrix library does such operations automatically. Note that the embedding of Natlog in Python makes such data exchanges $O(1)$ in time and space as no data-conversions need to be performed. Note also that the same applies to compound terms that correspond one-to-one to immutable nested tuples in Python and in particular to `pytrees`, instrumental in creating more advanced neural nets in the JAX ecosystem.

3.3 Interoperation with Pytorch

In the `Pytorch` ecosystem a combination of object-orientation and callable models encapsulate the underlying complexities of dataset management, neural network architecture choices, automatic differentiation, backpropagation and optimization steps. However, we can follow a similar encapsulation of

---

[6] `https://github.com/ptarau/natlog/blob/main/apps/deepnat/natjax.py`

architectural components as we have shown for JAX, and delegate to Python the details of building and initializing the network layers[7] with Natlog used to glue together the training and inference steps[8].

4 Logic Grammars as Prompt Generators

We will next overview Natlog applications for text-to-text and text-to-image generation. We refer to the Natlog code[9] and its Python companion[10] for full implementation details.

4.1 Prompt engineering by extending GPT3's text completion

GPT3 is basically a text completion engine, which, when given an initial segment of a sentence or paragraph as a *prompt*, it will complete it, often with highly coherent and informative results.

Thus, to get from GPT3 the intended output (e.g., answer to a question, elations extracted from a sentence, building analogies, etc.) one needs to rewrite the original input into a prompt that fits GPT3's text completion model.

We will use here Natlog's syntactically lighter Definite Clause Grammars, with one or more terminal symbols prefixed by "`@`" and "`=>`" replacing Prolog's "`-->`". A prompt generator with ability to be specialized for several "kinds" of prompts is described by the DCG rule:

```
prompt Kind QuestText => prefix Kind, sent QuestText, suffix Kind.
```

The predicate `sent` takes a question sentence originating from a user's input and maps it into a DCG non-terminal transforming cons-list `Ws1` into cons-list `Ws2`:

```
sent QuestText Ws1 Ws2 :
   `split QuestText List, to_cons_list List Ws, append Ws Ws2 Ws1.
```

The predicate `query` takes the DCG-generated `prompt` derived from user question `Q` and converts it back to a string passed to GPT'3 completion API by a call to the function `complete`, implemented in Python.

```
query Kind Q A: prompt Kind Q Ps (), to_list Ps List, `join List P, `complete P A.
```

Next we will describe specializations to question/answering, relation extraction and analogy invention.

An easy way to transform a question answering task into a completion task is to emulate a hypothetical conversation:

```
prefix question =>   @ 'If' you would ask me.
suffix question =>  @ 'I' would say that.
```

Similarly, extraction of subject-verb-object phrases can be mapped to completion tasks as in:

```
prefix relation =>
   @ 'If' you would ask me what are the subject and the verb and the object in .
suffix relation =>
   @  'I' would say subject is.
```

For analogy invention we create a custom trigger as follows:

---

[7] see `https://github.com/ptarau/natlog/blob/main/apps/deepnat/nattorch.py`

[8] see `https://github.com/ptarau/natlog/blob/main/apps/deepnat/nattorch.nat`

[9] see `https://github.com/ptarau/natlog/blob/main/apps/natgpt/chat.nat`

[10] see `https://github.com/ptarau/natlog/blob/main/apps/natgpt/chat.py`

```
trigger X Y Z =>
   @ given that X relates to Y by analogy 'I' would briefly say that Z relates to.

analogy X Y Z A:
   trigger X Y Z Ps (), to_list Ps List, `join List P, `complete P A.
```

We will next show interaction examples for all these use cases. First, question answering:

```
?- query question 'how are transformers used in GPT' R.
ANSWER: {'R': 'transformers are used in GPT (Generative Pre-trained Transformer)
models  to generate text from a given prompt. The transformer architecture is
used to learn the context of the input text and generate a response based on the
context. GPT models are  used in many natural language processing tasks such as
question answering, machine translation, summarization, and text generation.'}
```

Next, relation extraction. Note that after some preprocessing, the extracted triplets can be used as building blocks for knowledge graphs.

```
?- query relation 'the quick brown fox jumps over the lazy dog' R.
ANSWER: {'R': '"quick brown fox", verb is "jumps" and object is "lazy dog".'}

?- query relation 'high interest rates try to desperately contain inflation' R.
ANSWER: {'R': '"high interest rates", verb is "try to desperately contain",
and object is "inflation".'}
```

Finally, some examples of analogical reasoning that show GPT3 finding the missing component and explaining its reasoning.

```
?- analogy car wheel bird A?
ANSWER: {'A': 'wing by analogy. This is because both car and wheel are used for
transportation, while bird and wing are used for flight.'}

?- analogy car driver airplane A?
ANSWER: {'A': 'pilot by analogy. The pilot is responsible for the safe operation
of the airplane, just as the driver is responsible for the safe operation
of the car.'}

?-  analogy cowboy horse advertiser A?
ANSWER: {'A': 'customer by analogy in that they both need each other to achieve
a goal. The advertiser needs the customer to purchase their product or service
and the customer needs the advertiser to provide them with the product or
service they are looking for.'}
```

4.2 Text-to-image with DALL.E

With magic wands on a lease from generators like DALL.E [5] or Stable Diffusion [12], Natlog's Definite Clause Grammars can work as easy to customize prompt generators for such systems.

As the same OpenAI API (with a slightly different Python call) can be used for text-to-image generation (followed by displaying the generate picture in the user's default browser), the interaction with Python is expressed succinctly by the predicate `paint` that receives as `Prompt` the description of the intended picture from the user.

```
paint Prompt: `paint Prompt URL, #print URL, #browse URL.
```

The query to visualize in the user's browser one of the DCG-generated prompts is:

```
?- paint '<text description of intended image>'.
```

with some detail delegated to Python and taking advantage of the fact that the same OpenAI API is used for both text-to-text and text-to-image generation.

The Natlog DCG, in generation mode, can iterate over possible styles and content elements of a desired painting as in the following example:

```
image => style, subject, verb, object.

style => @photorealistic rendering.
style => @a dreamy 'Marc' 'Chagall' style picture.
style => @an action video game graphics style image.

subject => @of, adjective, noun.
noun => @robot.
verb => @walking.
adjective => @shiny.

object => location, @with,  instrument.

location => @on planet 'Mars'.
instrument => @high hills and a blue purse.
instrument => @a sombrero hat.
```

This generates text ready to be passed via the OpenAI Python APIs to DALL.E:

```
?- image Words (), `to_tuple Words Ws, #writeln Ws, nl, fail.
photorealistic rendering of shiny robot walking on planet Mars with high hills
and a blue purse
photorealistic rendering of shiny robot walking on planet Mars with a sombrero hat
.....
```

Besides the expected dependence on the `style` component (photorealistic vs. Chagall-style), as an illustration of GPT3's stereotyping bias, female and respectively male features would be derived from the generated robot pictures depending on the `purse` vs. `sombrero hat` picked by the DCG, as can be seen in the generated images[11].

5 Related Work

An introduction to Natlog, its initial proof-of-concept implementation and its content-driven indexing mechanism are covered in [9], but the language constructs and application discussed in this paper are all part of a fresh, "from scratch" implementation. We have implemented similar First-Class Logic Engines in the BinProlog [8] and Jinni Prolog systems [7], but their addition to Natlog, is motivated by their strong similarity with Python's own coroutining mechanisms. The use of coroutining in languages like `C#`, JavaScript and Python has also been used in the Yield Prolog system [10] as a facilitator for

---

[11] at `https://github.com/ptarau/natlog/tree/main/apps/natgpt/pics`

implementing backtracking, similarly to our implementation. Contrary to Natlog, which adopts its own surface syntax and reflection-based interaction with the host language, Yield Prolog requires idiomatic use of the syntax of the target language, making it significantly more cumbersome to work with.

Interoperation with Python has been also used in Problog [3] and DeepProblog [**?**], in the latter as a facilitator for neuro-symbolic computations. A comprehensive overview of neuro-symbolic reasoning, including logic-based, term-rewriting and graph-based is given in [6]. While in [9] we describe, as an example of neuro-symbolic interaction the use of a neural network as an alternative multi-argument indexer for Natlog, in this paper our focus is on the use of Natlog as an orchestrator for putting together and training deep learning systems and as a prompt generator for Large Language Models.

OpenAI's own `GPT 3.5`-based `ChatGPT`[12] automates the mapping of more queries (e.g., questions, code generation, dialog sessions, etc.) using an extensive Reinforcement Learning With Human Advice process [4]. By contrast, our DCG-supported approach relies exclusively on the pure GPT3 text-completion API on top of which we engineer task-specific prompts.

6 Conclusion

Motivated by expressiveness challenges faced by logic-based programming languages in the context of today's competitive landscape of alternative paradigms as well as from neural net-based machine learning frameworks, we have sketched some implementationally speaking "low-hanging" enhancements to them, with emphasis on coroutining methods and neuro-symbolic interoperation mechanisms. The main contributions of this work, likely to be reusable when bridging other Prolog systems to deep-learning ecosystems, are techniques that facilitate interoperation in the presence of high-level language constructs like finite functions, generators, on-demand computations, backtracking, nested immutable data types and strong reflection and metaprogramming features. The use cases described in the paper show the practical expressiveness of the Natlog-Python symbiosis by enhancing interaction with today's latest generation deep-learning tools with the declarative convenience of a lightweight embedded logic programming language.

References

[1] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne & Qiao Zhang (2018-2022): *JAX: composable transformations of Python+NumPy programs*. Available at `http://github.com/google/jax`.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever & Dario Amodei (2020): *Language Models are Few-Shot Learners*. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan & H. Lin, editors: *Advances in Neural Information Processing Systems*, 33, Curran Associates, Inc., pp. 1877–1901, doi:10.48550/arXiv.2005.14165. Available at `https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf`.

[3] Luc De Raedt, Angelika Kimmig & Hannu Toivonen (2007): *ProbLog: A Probabilistic Prolog and Its Application in Link Discovery.* In: *IJCAI*, 7, pp. 2462–2467, doi:10.5555/1625275.1625673.

---

[12] https://chat.openai.com/chat

[4] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike & Ryan Lowe (2022): *Training language models to follow instructions with human feedback*, doi:10.48550/ARXIV.2203.02155. Available at `https://arxiv.org/abs/2203.02155`.

[5] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen & Ilya Sutskever (2021): *Zero-Shot Text-to-Image Generation*, doi:10.48550/ARXIV.2102.12092. Available at `https://arxiv.org/abs/2102.12092`.

[6] Md Kamruzzaman Sarker, Lu Zhou, Aaron Eberhart & Pascal Hitzler (2021): *Neuro-Symbolic Artificial Intelligence: Current Trends*, doi:10.48550/ARXIV.2105.05330. Available at `https://arxiv.org/abs/2105.05330`.

[7] Paul Tarau (1999): *Inference and Computation Mobility with Jinni*. In K.R. Apt, V.W. Marek & M. Truszczynski, editors: *The Logic Programming Paradigm: a 25 Year Perspective*, Springer, Berlin Heidelberg, pp. 33–48, doi:10.1007/978-3-642-60085-2˙2. ISBN 3-540-65463-1.

[8] Paul Tarau (2000): *Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects*. In John Lloyd, editor: *Computational Logic–CL 2000: First International Conference*, London, UK, doi:10.1007/3-540-44957-4˙82. LNCS 1861, Springer-Verlag.

[9] Paul Tarau (2021): *Natlog: a Lightweight Logic Programming Language with a Neuro-symbolic Touch*. In Andrea Formisano, Yanhong Annie Liu, Bart Bogaerts, Alex Brik, Veronica Dahl, Carmine Dodaro, Paul Fodor, Gian Luca Pozzato, Joost Vennekens & Neng-Fa Zhou, editors: Proceedings 37th International Conference on *Logic Programming (Technical Communications) , 20-27th September 2021*, doi:10.4204/EPTCS.345.27.

[10] Jeff Thmpson (2019): *Yield Prolog*. Available at `https://yieldprolog.sourceforge.net/`.

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser & Illia Polosukhin (2017): *Attention is All you Need*. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan & R. Garnett, editors: *Advances in Neural Information Processing Systems*, 30, Curran Associates, Inc., doi:10.5555/3295222.3295349. Available at `https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.

[12] CompVis Machine Vision & Learning Research Group at LMU Munich (2018-2022): *Stable Diffusion*. Available at `https://github.com/CompVis/stable-diffusion`.

Solving Recurrence Relations using Machine Learning, with Application to Cost Analysis

[1,2]Maximiliano Klemen, [3]Miguel Á. Carreira-Perpiñán, and [4,2]Pedro Lopez-Garcia

[1]Universidad Politécnica de Madrid (UPM) Madrid, Spain

[2]IMDEA Software Institute, Madrid, Spain

[3]University of California, Merced, USA

[4]Spanish Council for Scientific Research, Madrid, Spain

`{maximiliano.klemen,pedro.lopez}@imdea.org, mcarreira-perpinan@ucmerced.edu`

Automatic static cost analysis infers information about the resources used by programs without actually running them with concrete data, and presents such information as functions of input data sizes. Most of the analysis tools for logic programs (and other languages) are based on setting up recurrence relations representing (bounds on) the computational cost of predicates, and solving them to find closed-form functions that are equivalent to (or a bound on) them. Such recurrence solving is a bottleneck in current tools: many of the recurrences that arise during the analysis cannot be solved with current solvers, such as Computer Algebra Systems (CASs), so that specific methods for different classes of recurrences need to be developed. We address such a challenge by developing a novel, general approach for solving arbitrary, constrained recurrence relations, that uses machine-learning sparse regression techniques to *guess* a candidate closed-form function, and a combination of an SMT-solver and a CAS to *check* whether such function is actually a solution of the recurrence. We have implemented a prototype and evaluated it with recurrences generated by a cost analysis system (the one in CiaoPP). The experimental results are quite promising, showing that our approach can find closed-form solutions, in a reasonable time, for classes of recurrences that cannot be solved by such a system, nor by current CASs.

1 Introduction and Motivation

The motivation of the work presented in this paper stems from automatic static cost analysis and verification of logic programs [3, 2, 4, 13, 17, 9, 8]. The goal of such analysis is to infer information about the resources used by programs without actually running them with concrete data, and present such information as functions of input data sizes and possibly other (environmental) parameters. We assume a broad concept of resource as a numerical property of the execution of a program, such as number of *resolution steps*, *execution time*, *energy consumption*, *memory*, number of *calls* to a predicate, number of *transactions* in a database, etc. Estimating in advance the resource usage of computations is useful for a number of applications, such as automatic program optimization, verification of resource-related specifications, detection of performance bugs, helping developers make resource-related design decisions, security applications (e.g., detection of side channels attacks), or blockchain platforms (e.g., smart-contract gas analysis and verification).

The challenge we address originates from the established approach of setting up recurrence relations representing the cost of predicates, parameterized by input data sizes [18, 16, 3, 2, 4, 13, 1, 17, 9], which are then solved to obtain *closed forms* of such recurrences (i.e., functions that provide either exact, or upper/lower bounds on resource usage in general). Such approach can infer different classes of functions (e.g., polynomial, factorial, exponential, summation, or logarithmic).

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 155–168, doi:10.4204/EPTCS.385.16

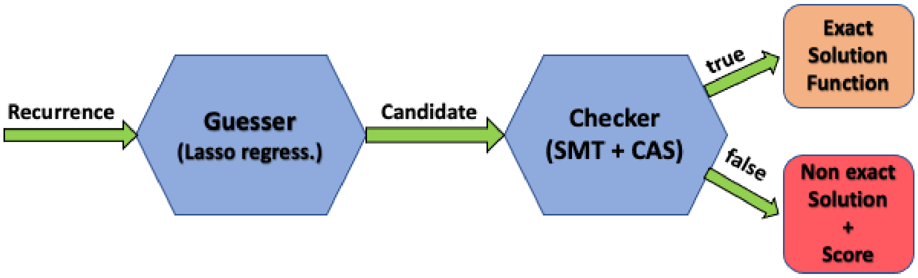


Figure 1: Control flow diagram of our novel solver based on machine learning.

The applicability of these resource analysis techniques strongly depends on the capabilities of the component in charge of solving (or safely approximating) the recurrence relations generated during the analysis, which has become a bottleneck in some systems.

A common approach to automatically solving such recurrence relations consists of using a Computer Algebra System (CAS) or a specialized solver to find a closed form. However, this approach poses several difficulties and limitations. For example, some recurrence relations contain complex expressions or recursive structures that most of the well-known CASs cannot solve, making it necessary to develop ad-hoc techniques to handle such cases. Moreover, some recurrences may not have the form required by such systems because an input data size variable does not decrease, but increases instead. Note that a decreasing-size variable could be implicit in the program, i.e., it could be a function of a subset input data sizes (a ranking function), which could be inferred by applying established techniques used in termination analysis [15]. However, such techniques are usually restricted to linear arithmetic.

In order to address this challenge we have developed a novel, general method for solving arbitrary, constrained recurrence relations. It is a *guess and check* approach that uses machine learning techniques for the *guess* stage, and a combination of an SMT-solver and a CAS for the *check* stage (see Figure 1). To the best of our knowledge, there is no other approach that does this. The resulting closed-form function solutions can be of different kinds, such as polynomial, factorial, exponential, summation, or logarithmic.

The rest of this paper is organized as follows. Section 2 gives and overview of our novel *guess and check* approach. Then Section 3 provides some background information and preliminary notation. Section 4 presents a more detailed, formal and algorithmic description of our approach. Section 5 describes the use of our approach in the context of static cost analysis. Section 6 comments on our prototype implementation and its experimental evaluation. Finally, Section 7 summarizes some conclusions and lines for future work.

2 Overview of our Approach

We now give an overview of the two stages of our approach already mentioned: *guess* a candidate closed-form function, and *check* whether such function is actually a solution of the recurrence relation.

Given a recurrence relation for a function $f(\vec{x})$, solving it means to find a closed-form function $\hat{f}(\vec{x})$ that has the same domain as $f(\vec{x})$, and for all $\vec{x}$ in such domain, $\hat{f}(\vec{x}) = f(\vec{x})$. By a closed-form function $\hat{f}$ we mean an expression that is built by using only elementary arithmetic functions, e.g., constants, addition, subtraction, multiplication, division, exponential, or even factorial functions. In particular, this means that $\hat{f}$ does not contain any subexpressions built by using the same function $\hat{f}$ (i.e., $\hat{f}$ is not recursively defined). We will use the following recurrence as an example to illustrate our approach:

$$\begin{array}{rll} f(x) = & 0 & \text{if } x = 0 \\ f(x) = & f(f(x-1)) + 1 & \text{if } x > 0 \end{array} \tag{1}$$

2.1 The "guess" stage (sparse linear regression via Lasso)

We use a sparse linear regression mechanism (see Section 3 for more details), so that any possible model we can obtain (which constitutes a candidate solution) must be a linear combination of a predefined set of terms, but using a usually small subset of terms. That is, a function $\hat{f}(\vec{x})$ of the form:

$$\hat{f}(\vec{x}) = \beta_0 + \beta_1\, t_1(\vec{x}) + \beta_2\, t_2(\vec{x}) + \cdots + \beta_n\, t_n(\vec{x})$$

where the t_i's are arbitrary functions on $\vec{x}$ from a set T of candidate terms that we call *base functions*, and the β_i's are the coefficients (real numbers) that are estimated by regression, but so that only a few coefficients are nonzero. Currently, the set T is fixed, and contains the base functions that are representative of the common complexity orders (in Section 7 we comment on future plans to obtain it). For illustration purposes, assume that we use the following set T of base functions:

$$T = \{\lambda x.x, \lambda x.x^2, \lambda x.x^3, \lambda x.\lceil \log_2(x) \rceil, \lambda x.2^x, \lambda x.x \cdot \lceil \log_2(x) \rceil\}$$

where each base function is represented as a lambda expression. Then, the sparse linear regression is performed as follows:

1. Generate a training set S. First, a set $X_{\text{train}} = \{\vec{x}_1, \ldots, \vec{x}_k\}$ of input values to the recurrence function is randomly generated. Then, starting with an initial $S = \emptyset$, for each input value $\vec{x}_i \in X_{\text{train}}$, a training case s_i is generated and added to S. For any input value $\vec{x} \in X_{\text{train}}$ the corresponding training case s is a tuple of the form:

   $$s = \langle b, c_1, \ldots, c_n \rangle$$

   where $c_i = [\![t_i]\!]_{\vec{x}}$ for $1 \leq i \leq n$, and $[\![t_i]\!]_{\vec{x}}$ represents the result (a scalar) of evaluating the base function $t_i \in T$ for input value $\vec{x}$, where T is a set of n base functions, as already explained. The (dependent) value b (also a constant number) is the result of evaluating the recurrence $f(\vec{x})$ that we want to solve or approximate, in our example, the one defined in Equation 1. Assuming that there is an $\vec{x} \in X_{\text{train}}$ such that $\vec{x} = \langle 5 \rangle$, its corresponding training case s in our example will be:

   $$\begin{array}{rcl} s & = & \langle \mathbf{f(5)}, [\![x]\!]_5, [\![x^2]\!]_5, [\![x^3]\!]_5, [\![\lceil \log_2(x) \rceil]\!]_5, \ldots \rangle \\ & = & \langle \mathbf{5}, 5, 25, 125, 3, \ldots \rangle \end{array}$$

2. Perform the sparse regression in two steps using the training set S created above. In the first step, we use linear regression with Lasso (ℓ_1) regularization [6] on the coefficients. This is a penalty term that encourages coefficients whose associated base functions have a small correlation with the dependent value to be exactly zero. This way, typically most of the base functions in T will be discarded, and only those that are really needed to approximate our target function will be kept. The level of penalization is controlled by a hyperparameter $\lambda \geq 0$. As commonly done in machine learning [6], the value of λ that generalizes optimally on unseen (test) inputs is found via cross-validation on a separate validation set (generated randomly in the same way as the training set). The result of this step is a (column) vector $\vec{\beta}$ of coefficients, and an independent coefficient β_0. Finally, we generate a test set X_{test} (again, randomly in the same way as the training set) of

input values to the recurrence function to obtain a measure R^2 of the accuracy of the estimation. Additionally, we discard those terms whose corresponding coefficient is less than a given threshold ε. The resulting closed-form expression that estimates the target function is

$$\hat{f}(\vec{x}) = \mathsf{rm}_{\varepsilon}(\vec{\beta}^T) \cdot E(T, \vec{x}) + \beta_0$$

where $E(T, \vec{x})$ is a vector of the terms in T with the arguments bound to $\vec{x}$, and $\mathsf{rm}_{\varepsilon}$ takes a vector of coefficients and returns another vector where the coefficients less than ε are rounded to zero. Both the Lasso regularization and the pruning function discard many terms from T in the final function.

3. Finally, our method performs again a standard linear regression (without Lasso regularization) on the training set S, but without using those base functions corresponding to the terms discarded previously by Lasso and the ε-pruning. In our example, with $\varepsilon = 0.05$, we obtain:

$$\hat{f}(x) = 1.0\,x$$

with a value $R^2 = 1$, which means that the estimation obtained predicts exactly the values for the test set, and thus, it is a candidate solution for the recurrence in Equation 1. If R^2 were less than 1, it would mean that the function obtained is not a candidate (exact) solution, but a (possibly unsafe) approximation, as there are values in the test set that cannot be exactly predicted.

2.2 The "check" stage

Once a function that is a candidate solution for the recurrence has been guessed, the second step of our method tries to verify whether such a candidate is actually a solution. To do so, the recurrence is encoded as a first order logic formula where the references to the target function are replaced by the candidate solution whenever possible. Afterwards, we use an SMT-solver to check whether the negation of such formula is satisfiable, in which case we can conclude that the candidate is not a solution for the recurrence. Otherwise, if such formula is unsatisfiable, then the candidate function is an exact solution. Sometimes, it is necessary to consider a precondition for the domain of the recurrence, which is also included in the encoding.

To illustrate this process, Expression (2) below shows the recurrence relation we target to solve, followed by the candidate solution obtained previously using linear regression:

$$\begin{array}{rll} f(x) = & 0 & \text{if } x = 0 \\ f(x) = & f(f(x-1)) + 1 & \text{if } x > 0 \\ \hat{f}(x) = & x & \text{if } x \geq 0 \end{array} \tag{2}$$

Now, Expression (3) below shows the encoding of the recurrence as a first order logic formula.

$$\forall x \left((x = 0 \implies \underline{f(x)} = 0) \wedge (x > 0 \implies \underline{f(x)} = \underline{f(\underline{f(x-1)})} + 1) \right) \tag{3}$$

Finally, Expression (4) below shows the negation of such formula, as well as the references to the function name substituted by the definition of the candidate solution. We underline both the subexpressions to be replaced, and the subexpressions resulting from the substitutions.

$$\exists x \neg(((x = 0 \implies \underline{x} = 0) \wedge (x > 0 \implies \underline{x} = \underline{x-1} + 1))) \tag{4}$$

It is easy to see that Formula (4) is unsatisfiable. Therefore, $\hat{f}(x) = x$ is an exact solution for $f(x)$ in the recurrence defined by Equation 1.

For some cases where the candidate solution contains transcendental functions, our implementation of the method uses a CAS to perform simplifications and transformations, in order to obtain a formula supported by the SMT-solver. We find this combination of CAS and SMT-solver particularly useful, since it allows solving more problems than only using one of these systems in isolation.

3 Preliminaries

Recurrence relations. A recurrence relation of order k, $k > 0$, for a function f, is a set of equations that give k initial values for f, and an equation that recursively defines any other value of f as a function g that takes k previous values of f as parameters. For example, the following recurrence relation of second order ($k = 2$), with g being the arithmetic addition $+$, defines the Fibonacci function:

$$f(n) = \begin{cases} 1 & \text{if } \mathrm{n} = 0 \text{ or } \mathrm{n} = 1 \\ f(n-1) + f(n-2) & \text{if } \mathrm{n} \geq 2 \end{cases} \tag{5}$$

A challenging class of recurrences that we can solve with our approach are "nested" recurrences, e.g., recurrences of the form $f(n) = g(f(f(n-1)))$.

We use the letters x, y, z to denote variables, and a, b, c, d to denote constants and coefficients. We use f, g to represent functions, and e, t to represent arbitrary expressions. We use φ to represent arbitrary boolean constraints over a set of variables. Sometimes, we also use β to represent coefficients obtained with linear regression. In all cases, the symbols can be subscribed. We use $\vec{x}$ to denote a finite sequence $\langle x_1, x_2, \ldots, x_n \rangle$, for some $n > 0$. Given a sequence S and an element x, $\langle x | S \rangle$ is a new sequence with first element x and tail S.

Given a piecewise function:

$$f(\vec{x}) = \begin{cases} e_1(\vec{x}) & \text{if } \varphi_1(\vec{x}) \\ e_2(\vec{x}) & \text{if } \varphi_2(\vec{x}) \\ \vdots & \vdots \\ e_k(\vec{x}) & \text{if } \varphi_k(\vec{x}) \end{cases} \tag{6}$$

where $f \in \mathscr{D} \to \mathbb{R}^+$, with $\mathscr{D} = \{\vec{x} | \vec{x} \in \mathbb{Z}^m \wedge \varphi_{\text{pre}}(\vec{x})\}$ for some boolean constraint φ_{pre}, and $e_i(\vec{x}), \varphi_i(\vec{x})$ are arbitrary expressions and constraints over $\vec{x}$ respectively. We say that φ_{pre} is the *precondition* of f, and that f is a *constrained recurrence relation* if and only if:

- $\exists i \in [1, k]$ such that e_i contains a call to f.
- $\exists i \in [1, k]$ such that e_i does not contain any call to f (i.e., it is in *closed form*).
- $\varphi_{\text{pre}} \models \bigvee_{1 \leq i \leq k} \varphi_i$.

Given a concrete input $\vec{d} \in \mathscr{D}$, we evaluate $f(\vec{d})$ deterministically, assuming the evaluation of f as a nested *if-then-else* control structure as follows:

```
if $\varphi_1(\vec{d})$ then
  return $e_1(\vec{d})$
else
  if $\varphi_2(\vec{d})$ then
    return $e_2(\vec{d})$
  else
    ...
  end if
end if
```

More formally, let $\mathsf{def}(f)$ denote the definition of a (piecewise) constrained recurrence relation f represented as the sequence $\langle(e_1(\vec{x}),\varphi_1(\vec{x})),\ldots,(e_k(\vec{x}),\varphi_k(\vec{x}))\rangle$, where each element of the sequence is a pair representing a case. The order of such sequence determines the evaluation strategy. Then, the evaluation of f for a concrete value $\vec{d}$, denoted $\mathsf{EvalFun}(f(\vec{d}))$, is defined as follows:

$$\mathsf{EvalFun}(f(\vec{d})) = \mathsf{EvalBody}(\mathsf{def}(f),\vec{d})$$

$$\mathsf{EvalBody}(\langle(e,\varphi)|\mathsf{Ps}\rangle,\vec{d}) = \begin{cases} [\![e]\!]_{\vec{d}} & \text{if } \varphi(\vec{d}) \\ \mathsf{EvalBody}(\mathsf{Ps},\vec{d}) & \text{if } \neg\varphi(\vec{d}) \end{cases}$$

Our goal is to find a function $\hat{f} \in \mathscr{D} \to \mathbb{R}^+$ such that for all $\vec{d} \in \mathscr{D}$:

- If $\mathsf{EvalFun}(f(\vec{d}))$ terminates, then $\mathsf{EvalFun}(f(\vec{d})) = [\![\hat{f}]\!]_{\vec{d}}$, and
- $\hat{f}$ does not contain any recursive call in its definition.

In particular, we look for a definition of the form:

$$\hat{f}(\vec{x}) = \beta_0 + \beta_1\, t_1(\vec{x}) + \beta_2\, t_2(\vec{x}) + \cdots + \beta_n\, t_n(\vec{x}) \tag{7}$$

where $\beta_i \in \mathbb{R}$, and t_i are expressions over $\vec{x}$, not including recursive references to $\hat{f}$. If the above conditions are met, we say that $\hat{f}$ is a *closed form* for f.

To illustrate the need of introducing an evaluation strategy for the recurrence that is consistent with the termination of the program, consider the following Prolog program which does not terminate for a call `p(X)` where `X` is bound to an integer:

```
p(X) :- X > 0, X1 is X + 1, p(X1).
p(X) :- X = 0.
```

The following recurrence relation for its cost (in resolution steps) can be set up:

$$\begin{array}{lll} \mathtt{C_p}(x) = & 1 & \text{if } x = 0 \\ \mathtt{C_p}(x) = & 1 + \mathtt{C_p}(x+1) & \text{if } x > 0 \end{array} \tag{8}$$

A CAS will give the closed form $\mathtt{C_p}(x) = 1 - x$ for such recurrence, however, the cost analysis should give $\mathtt{C_p}(x) = \infty$.

Linear Regression. Linear regression [5] is a statistical technique used to approximate the linear relationship between a number of independent variables and a dependent (output) variable. Given a vector of independent (input) variables $X = (X_1, \ldots, X_p)^T \in \mathbb{R}^p$, we predict the output variable Y using the formula

$$Y = \beta_0 + \sum_{i=1}^{p} \beta_i X_i \tag{9}$$

which is defined through the vector of coefficients $\beta = (\beta_0, \ldots, \beta_p)^T \in \mathbb{R}^p$. Such coefficients are estimated from a set of observations $\{y_i, x_{i1}, \ldots, x_{ip}\}_{i=1}^{n}$ so as to minimize a loss function, most commonly the sum of squares

$$\beta = \operatorname*{arg\,min}_{\beta \in \mathbb{R}^p} \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 \tag{10}$$

Sometimes (as is our case) some of the input variables are not relevant to explain the output, but the above least-squares estimate will almost always assign nonzero values to all the coefficients. In order to force the estimate to make exactly zero the coefficients of irrelevant variables (hence removing them and doing *feature selection*), various techniques have been proposed. The most widely used one is the Lasso [6], which adds an ℓ_1 penalty on β (i.e., the sum of absolute values of each coefficient) to Expression 10:

$$\beta = \operatorname*{arg\,min}_{\beta \in \mathbb{R}^p} \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \tag{11}$$

where $\lambda \geq 0$ is a hyperparameter that determines the level of penalization: the greater λ, the greater the number of coefficients that are exactly equal to 0. The Lasso has two advantages over other feature selection techniques for linear regression. First, it defines a convex problem whose unique solution can be efficiently computed even for datasets where either of n or p are large (almost as efficiently as a standard linear regression). Second, it has been shown in practice to be very good at estimating the relevant variables.

4 Algorithmic Description of the Approach

In this section we describe our approach for generating and checking candidate solutions for recurrences that arise in resource analysis. Algorithms 1 and 2 correspond to the *guesser* and *checker* components, respectively, which are shown in Figure 1.

Algorithm 1 receives a recurrence relation for a function F to solve, a set of base functions, and a threshold to decide when to discard irrelevant terms. The output is a closed-form expression $\hat{F}$ for F, and a *score* S that reflects the accuracy of the approximation, in the range $[0,1]$. If $S \sim 1$, the approximation can be considered a candidate solution. Otherwise, $\hat{F}$ is a (possibly unsafe) approximation. In line 1 we start by generating a set $\mathscr{I}$ of random inputs for F. Each input $\vec{x_i}$ is a m-tuple verifying precondition φ_{pre}, where m is the number of arguments of F. In line 2 we produce the training set $\mathscr{X}$. The independent inputs are generated by evaluating the base functions in $T = \langle t_1, t_2, \ldots, t_p \rangle$ with each tuple $\vec{x} \in \mathscr{I}$. This is done by using function E, defined as follows:

$$E(\langle t_1, t_2, \ldots, t_p \rangle, \vec{x}) = \langle t_1(\vec{x}), t_2(\vec{x}), \ldots, t_p(\vec{x}) \rangle$$

We also evaluate the recurrence equation for input $\vec{x}$, and add the observed output $F(\vec{x})$ as the first element in the vectors of the training set. In line 3 we generate a first linear model by applying function CVLassoRegression to the generated training set. CVLassoRegression performs a linear regression

Algorithm 1: Candidate Solution Generation (*Guesser*).

```
Input  : $F \in \mathscr{D} \to \mathbb{R}^+$: target recurrence relation.
         $\varphi_{\text{pre}}$: precondition defining $\mathscr{D}$.
         $T \subseteq \mathscr{D} \to \mathbb{R}^+$: set of base functions.
         $\Lambda$: range of values to automatically choose a Lasso hyperparameter $\lambda \in \mathbb{R}^+$ that
         maximizes the performance of the model via cross-validation.
         $k$: indicates performing $k-$fold cross-validation, $k \geq 2$.
         $\varepsilon \in \mathbb{R}^+$: threshold for term ($t_i \in T$) selection.
Output: $\hat{F} \in \mathsf{Exp}$: a candidate solution (or an approximation) for $F$.
         $S \in [0,1]$: score, indicates the accuracy of the estimation ($R^2$).
1 $\mathscr{I} \leftarrow \{\vec{x}_i | \vec{x}_i \in \mathbb{Z}^m \wedge \varphi_{\text{pre}}(\vec{x}_i)\}_{i=1}^{N}$ ;        // N Random inputs for F
2 $\mathscr{X} \leftarrow \{\langle F(\vec{x}) | E(T,\vec{x})\rangle | \vec{x} \in \mathscr{I}\}$ ;        // Training set
3 $(\vec{\beta}', \beta_0') \leftarrow \mathsf{CVLassoRegression}(\mathscr{X}, \Lambda, k)$;
4 $(T', \mathscr{X}') \leftarrow \mathsf{RemoveTerms}(T, \mathscr{X}, \vec{\beta}', \beta_0', \varepsilon)$;
5 $(\vec{\beta}, \beta_0, S) \leftarrow \mathsf{LinearRegression}(\mathscr{X}')$;
6 $\hat{F} \leftarrow \lambda\vec{x} \cdot \vec{\beta}^T \times E(T',\vec{x}) + \beta_0$;
7 return $(\hat{F}, S)$;
```

with Lasso regularization. As already mentioned, Lasso regularization requires a hyperparameter λ that determines the level of penalization for the coefficients. Instead of using a single value for λ, CVLassoRegression uses a range of possible values, applying cross-validation on top of the linear regression to automatically select the best value for that parameter, from the given range. The parameter k indicates performing $k-$fold cross-validation, which means that the training set is split into k parts or *folds*. Then, each fold is taken as the validation set, training the model with the remaining $k-1$ folds. Finally, the performance measure reported is the average of the values computed in the k iterations. The result of this function is the vector of coefficients $\vec{\beta}'$, together with the intercept β_0'. These coefficients are used in line 4 to decide which base functions are discarded before the last regression step. Note that RemoveTerms removes the base functions from T together with their corresponding input values from the training set $\mathscr{X}$, returning the new set of base functions T' and its corresponding training set $\mathscr{X}'$. In line 5, standard linear regression (without regularization nor cross-validation) is applied, obtaining the final coefficients $\vec{\beta}$ and β_0. Additionally, from this step we also obtain the score S of the resulting model. In line 6 we set up the resulting closed-form expression, given as a function on the variables in $\vec{x}$. Note that we use the function E to bind the variables in the base functions to the arguments of the closed-form expression. Finally, the closed-form expression and its corresponding score are returned as the result of the algorithm.

Algorithm 2 mainly relies on an SMT-solver and a CAS. Concretely, given the constrained recurrence relation $F \in \mathscr{D} \to \mathbb{R}^+$ defined as

$$F(\vec{x}) = \begin{cases} e_1(\vec{x}) & \text{if } \varphi_1(\vec{x}) \\ e_2(\vec{x}) & \text{if } \varphi_2(\vec{x}) \\ \vdots & \vdots \\ e_k(\vec{x}) & \text{if } \varphi_k(\vec{x}) \end{cases}$$

```
Algorithm 2: Solution Checking (Checker).
    Input  : $F \in \mathscr{D} \to \mathbb{R}^+$: target recurrence relation.
             $\varphi_{\text{pre}}$: precondition defining $\mathscr{D}$.
             $\hat{F} \in \mathsf{Exp}$: a candidate solution for $F$.
    Output: true if $\hat{F}$ is a solution for $F$, false otherwise.
 1  $\varphi_{\text{previous}} \leftarrow \mathsf{true}$ ;
 2  Formula $\leftarrow$ true ;
 3  foreach $(e, \varphi) \in \mathsf{def}(F)$ do
 4      Eq $\leftarrow$ replaceCalls("$F(\vec{x}) - e = 0$", $F(\vec{x}), \hat{F}, \varphi_{\text{pre}}, \varphi$);
 5      if $\neg$ containsCalls(Eq, $F$) then
 6          Eq $\leftarrow$ simplifyCAS(inlineCalls(Eq, $\hat{F}$, def($\hat{F}$)));
 7          if supportedSMT(Eq) then
 8              Formula $\leftarrow$ "Formula $\wedge$ ($\varphi_{\text{pre}} \wedge \varphi_{\text{previous}} \wedge \varphi \implies$ Eq)";
 9              $\varphi_{\text{previous}} \leftarrow$ "$\varphi_{\text{previous}} \wedge \neg\varphi$" ;
10          else
11              return false;
12          end
13      else
14          return false;
15      end
16  end
17  return ($\not\models_{SMT}$ $[\![\neg$Formula$]\!]_{SMT}$);
```

our algorithm constructs the logic formula:

$$\left[\!\left[\bigwedge_{i=1}^{k} \left(\left(\bigwedge_{j=1}^{i-1} \neg\varphi_j(\vec{x}) \right) \wedge \varphi_i(\vec{x}) \wedge \varphi_{\text{pre}}(\vec{x}) \implies \mathsf{Eq}_i \right) \right]\!\right]_{\text{SMT}} \tag{12}$$

where Eq_i is the result of replacing in $F(\vec{x}) = e_i(\vec{x})$ each occurrence of F, if possible, by the definition of the candidate solution $\hat{F}$ (by using replaceCalls in line 4), and performing a simplification by the CAS (by using simplifyCAS in line 6). A goal of such simplification is to obtain (sub)expressions supported by the SMT-solver. The function $\mathsf{replaceCalls}(\mathsf{expr}, F(\vec{x}'), \hat{F}, \varphi_{\text{pre}}, \varphi)$ replaces every subexpression in expr of the form $F(\vec{x}')$ by $\hat{F}(\vec{x}')$, if $\varphi_{\text{pre}}(\vec{x}') \wedge \varphi \implies \varphi_{\text{pre}}(\vec{x}')$. The operation $[\![e]\!]_{\text{SMT}}$ is the translation of any expression e to an SMT-LIB expression. Although all variables appearing in Formula 12 are declared as integers, we omit these details in Algorithm 2 and in Formula 12 for the sake of brevity. Note that this encoding is consistent with the evaluation (EvalFun) described in Section 3. Finally, the algorithm asks the SMT-solver for models of the negated formula (line 17). If no model exists, then it returns true, concluding that $\hat{F}$ is an exact solution to the recurrence, i.e., $\hat{F}(\vec{x}) = F(\vec{x})$ for any input $\vec{x} \in \mathscr{D}$ such that $\mathsf{EvalFun}(F(\vec{x}))$ terminates. Otherwise, it returns false. Note that, if it is not possible to replace all occurrences of F by $\hat{F}$, or if after performing the simplification by simplifyCAS there are subexpressions not supported by the SMT-solver, then the algorithm finishes returning false.

5 Our Approach in the Context of Static Cost Analysis

In this section, we describe how our approach could be used in the context of the motivating application, Static Cost Analysis. Although it is general, and could be integrated into any cost analysis system based on recurrence solving, we illustrate its use in the context of the CiaoPP system. Using a logic program, we first illustrate how CiaoPP sets up recurrence relations representing the sizes of output arguments of predicates and the cost of such predicates. Then, we show how our novel approach is used to solve a recurrence relation that cannot be solved by CiaoPP.

Example 1 *Consider predicate* `p/2` *in Figure 2, and calls to it where the first argument is bound to a non-negative integer and the second one is a free variable. Upon success of these calls, the second argument is bound to an non-negative integer too. Such calling mode, where the first argument is input and the second one is output, is automatically inferred by CiaoPP (see [7] and its references).*

```
:- entry p/2: nnegint*var.
p(X,0):-
   X=0.
p(X,Y):-
   X>0,
   X1 is X - 1,
   p(X1,Y1),
   p(Y1,Y2),
   Y is Y2 + 1.
```

Figure 2: A program with a nested recursion.

The CiaoPP system first infers size relations for the different arguments of predicates, using a rich set of size metrics (see [13, 17] for details). Assume that the size metric used in this example, for the numeric argument `X` is the *actual value* of it (denoted `int(X)`). The system will try to infer a function $\mathtt{S}_{\mathtt{p}}(x)$ that gives the size of the output argument of `p/2` (the second one), as a function of the size (x) of the input argument (the first one). For this purpose, the following size relations for $\mathtt{S}_{\mathtt{p}}(x)$ are automatically set up (the same as the recurrence in Equation 1 used in Section 2 as example):

$$\begin{array}{lll} \mathtt{S}_{\mathtt{p}}(x) = & 0 & \text{if } x = 0 \\ \mathtt{S}_{\mathtt{p}}(x) = & \mathtt{S}_{\mathtt{p}}(\mathtt{S}_{\mathtt{p}}(x-1)) + 1 & \text{if } x > 0 \end{array} \tag{13}$$

The first and second recurrence correspond to the first and second clauses respectively (i.e., base and recursive cases). Once recurrence relations (either representing the size of terms, as the ones above, or the computational cost of predicates, as the ones that we will see latter) have been set up, a solving process is started.

Nested recurrences, as the one that arise in this example, cannot be handled by most state-of-the-art recurrence solvers. In particular, the modular solver used by CiaoPP fails to find a closed-form function for the recurrence relation above. In contrast, the novel approach that we propose, sketched in next section, obtains the closed form $\hat{\mathtt{S}}_{\mathtt{p}}(x) = x$, which is an exact solution of such recurrence (as shown in Section 2).

Once the size relations have been inferred, CiaoPP uses them to infer the computational cost of a call to `p/2`. For simplicity, assume that in this example, such cost is given in terms of the number of *resolution steps*, as a function of the size of the input argument, but note that CiaoPP's cost analysis is parametric with respect to resources, which can be defined by the user by means of a rich assertion language, so that it can infer a wide range of resources, besides resolution steps. Also for simplicity, we assume that all builtin predicates, such as arithmetic/comparison operators have zero cost (in practice there is a "trust"assertion for each builtin that specifies its cost as if it had been inferred by the analysis).

In order to infer the cost of a call to `p/2`, represented as $\mathtt{C_p}(x)$, CiaoPP sets up the following cost relations, by using the size relations inferred previously:

$$\begin{array}{lll} \mathtt{C_p}(x) = & 1 & \text{if } x = 0 \\ \mathtt{C_p}(x) = & \mathtt{C_p}(x-1) + \mathtt{C_p}(\mathtt{S_p}(x-1)) + 1 & \text{if } x > 0 \end{array} \tag{14}$$

We can see that the cost of the second recursive call to predicate `p/2` depends on the size of the output argument of the first recursive call to such predicate, which is given by function $\mathtt{S_p}(x)$, whose closed form $\mathtt{S_p}(x) = x$ is computed by our approach, as already explained. Plugin such closed form into the recurrence relation above, it can be solved now by CiaoPP, obtaining $\mathtt{C_p}(x) = 2^{x+1} - 1$.

6 Implementation and Experimental Evaluation

We have implemented a prototype of our novel approach and performed an experimental evaluation in the context of the CiaoPP system, by solving recurrences generated during static cost analysis. Our prototype takes a recurrence and returns a closed form obtained together with two measures: 1) the accuracy of the estimation (*score*) of the candidate closed-form solution generated by the machine learning phase, and 2) an indication of whether such closed form is an exact solution of the recurrence (i.e., if it has been formally verified). It is implemented in *Python 3*, using *Sympy* [11] as CAS, and *Scikit-Learn* [14] for the regression with Lasso regularization. We use *Z3* [12] as SMT-solver, and *Z3Py* [19] as interface.

Our experimental results are shown in Table 1. Column **Bench** shows the name that we have assigned to each recurrence that we have chosen (which is inspired in the logic program such recurrence originated from during cost/size analysis), and Column **Recurrence** shows their definitions, where we use the same function symbol, f, for all of them. Such recurrences are challenging for CiaoPP, either because they cannot be solved by any of the back-end solvers, or because they are necessarily over-estimated in the solving process. Some recurrences, like **nested**, are problematic even for most of the current state-of-the-art solvers. For each recurrence and for each argument x of it, there is an implicit constraint that x is an integer, $x \geq 0$, which we do not include for brevity. Also, the disjunction of all the constraints defining the cases of the recurrence is a constraint φ_{pre} that defines the domain of the corresponding function. Column **CF** shows the closed forms obtained by our previous recurrence solver, and Column **CFNew** shows the closed forms obtained by our approach, applying Algorithms 1 and 2. All of them have been verified as exact solutions to the recurrences by Algorithm 2. As already said in Section 3, the closed form solution of any recurrence gives the same results as the recurrence for the inputs for which the evaluation of the recurrence terminates. The base cases of the recurrences have been considered apart from the others, and included in the final solution. Finally, Column **T(s)** shows the total time, in seconds (executing on a MacBook Pro machine, 2.4GHz Intel Core i7 CPU, 8 GB 1333 MHz DDR3 memory), needed to obtain the closed forms and verify them. For all the experiments, we have set $k = 2$, in order to perform $2-$fold cross-validation. We have also set the range for λ to 100 values taken from the interval $[0.001, 1]$, and $\varepsilon = 0.05$. Regarding the set T of base functions, for recurrences with one or two arguments, we provide a predefined set of representative functions of the most common complexity orders, as well as some compositions of them. For recurrences with three or more arguments, we provide an initial set of simple functions, that are combined automatically to generate the base functions t_i for the set T.

Table 1: Closed forms obtained with the previous (**CF**) and new solver (**CFNew**).

Bench	**Recurrence**	**CF**	**CFNew**	**T (s)**
merge-sz	$f(x,y)=\begin{cases} max(f(x-1,y), & \\ f(x,y-1))+1 & \text{if } x>0 \wedge y>0 \\ x & \text{if } x>0 \wedge y=0 \\ y & \text{if } x=0 \wedge y>0 \end{cases}$	$-$	$x+y$	0.92
merge	$f(x,y)=\begin{cases} max(f(x-1,y), & \\ f(x,y-1))+1 & \text{if } x>0 \wedge y>0 \\ 0 & \text{if } x=0 \vee y=0 \end{cases}$	$-$	$\begin{cases} x+y-1 & \text{if } x>0 \wedge y>0 \\ 0 & \text{if } x=0 \vee y=0 \end{cases}$	0.71
nested	$f(x)=\begin{cases} f(f(x-1))+1 & \text{if } x>0 \\ 0 & \text{if } x=0 \end{cases}$	$-$	x	0.13
open-zip	$f(x,y)=\begin{cases} f(x-1,y-1)+1 & \text{if } x>0 \wedge y>0 \\ f(x,y-1)+1 & \text{if } x=0 \wedge y>0 \\ f(x-1,y)+1 & \text{if } x>0 \wedge y=0 \\ 0 & \text{if } x=0 \wedge y=0 \end{cases}$	$-$	$\max(x,y)$	0.12
div	$f(x,y)=\begin{cases} f(x-y,y)+1 & \text{if } x>=y \wedge y>0 \\ 0 & \text{if } x<y \wedge y>0 \end{cases}$	$-$	$\left\lfloor \frac{x}{y} \right\rfloor$	0.13
div-ceil	$f(x,y)=\begin{cases} f(x-y,y)+1 & \text{if } x>=y \wedge y>0 \\ 1 & \text{if } x<y \wedge x>0 \\ 0 & \text{if } x=0 \wedge y>0 \end{cases}$	$-$	$\left\lceil \frac{x}{y} \right\rceil$	0.12
s-max	$f(x,y)=\begin{cases} max(y,f(x-1,y))+1 & \text{if } x>0 \\ y & \text{if } x=0 \end{cases}$	$x+y$	$x+y$	0.12
s-max-1	$f(x,y)=\begin{cases} max(y,f(x-1,y+1))+1 & \text{if } x>0 \\ y & \text{if } x=0 \end{cases}$	$-$	$2x+y$	0.14
sum-osc	$f(x,y)=\begin{cases} f(x-1,y)+1 & \text{if } x>0 \wedge y>0 \\ f(x+1,y-1)+y & \text{if } x=0 \wedge y>0 \\ 1 & \text{if } y=0 \end{cases}$	$-$	$\begin{cases} x+\frac{y^2}{2}+\frac{3y}{2} & \text{if } y>0 \\ 1 & \text{if } y=0 \end{cases}$	0.13

As we can see, none of the recurrences are solvable by the current CiaoPP solver, except `s-max`. The specialized solver for such recurrence has been developed relatively recently. Also, none of the recurrences are solvable by the CASs *Mathematica* [10] and *Sympy* [11], which we can arguably consider state-of-the-art CASs. In contrast, our new solver is able to infer exact closed-forms functions for all the recurrences in a reasonable time.

7 Conclusions and Future Work

We have developed a novel approach for solving or approximating arbitrary, constrained recurrence relations. It consists of a *guess* stage that uses a sparse linear regression via Lasso regularization and cross-validation to infer a candidate closed-form solution, and a *check* stage that combines an SMT-solver and a CAS to verify that such candidate is actually a solution. We have implemented a prototype and evaluated it with recurrences generated by the cost analysis module of CiaoPP, and are not solvable by it nor by the (arguably state-of-the-art) CASs *Mathematica* and *Sympy*. The experimental results are quite promising, showing that our approach can find exact, verified, closed-form solutions, in a reasonable time, for such recurrences, which imply potentially, arbitrarily large accuracy gains in cost analysis of (logic) programs. Not being able to solve a recurrence can cause huge accuracy losses, for instance, if such a recurrence corresponds to a predicate that is deep in the control flow graph of the program, and such accuracy loss is propagated to the main predicate, inferring not useful information at all.

Since our technique uses linear regression with a randomly generated training set (by evaluating the recurrence to obtain the dependent value), it is not guaranteed that a solution can be found. Even if an exact solution is found in the first stage, it is not always possible to prove its correctness in the second stage. Therefore, in this sense, this approach is *not complete*. However, it is able to find some solutions that current state-of-the-art solvers are unable to find. As a proof of concept, we have considered a particular deterministic evaluation for constrained recurrence relations, and the verification of the candidate solution is consistent with this evaluation. However, it is possible to implement different evaluation semantics for the recurrences, adapting the verification stage accordingly. Note that we need to require the termination of the recurrence evaluation as a precondition for the conclusions obtained. This is also due to the particular evaluation strategy of recurrences that we are considering. In practice, non-terminating recurrences can be discarded in the first stage, by setting a timeout. Our approach can also be combined with a termination prover in order to guarantee such a precondition. Finally, note that an alternative use of our tool is to omit the verification stage, using only the closed-form function inferred by the first stage, together with an error measure. This can be useful in some applications (e.g., granularity control in parallel/distributed computing) where it is enough to have good although unsafe approximations.

As a future work, we plan to fully integrate our novel solver into the CiaoPP system, combining it with its current set of back-end solvers in order to improve the static cost analysis. We also plan to further refine and improve our algorithms in several directions. As already explained, currently the set T of base functions is fixed, user-provided. We plan to automatically infer it by using different heuristics. We can perform an automatic analysis of the recurrence we are solving, to extract some features that allow selection of the terms that most likely are part of the solution. For example, if the recurrence has a nested, double recursion, then we can select a quadratic term, etc. Also, machine learning techniques may be applied to learn a good set of base functions from some features of the programs.

Acknowledgments This work has been partially supported by MICINN projects PID2019-108528RB-C21 *ProCode*, TED2021-132464B-I00 *PRODIGY*, and FJC2021-047102-I, and the Tezos foundation. The authors would also like to thank Louis Rustenholz, John Gallagher, Manuel Hermenegildo, José F. Morales and the anonymous reviewers for very useful feedback. Louis Rustenholz also recreated the experimental results and double-checked them.

References

[1] E. Albert, P. Arenas, S. Genaim & G. Puebla (2011): *Closed-Form Upper Bounds in Static Cost Analysis*. *Journal of Automated Reasoning* 46(2), pp. 161–203, doi:10.1007/s10817-010-9174-1.

[2] S. K. Debray & N. W. Lin (1993): *Cost Analysis of Logic Programs*. *ACM TOPLAS* 15(5), pp. 826–875, doi:10.1145/161468.161472.

[3] S. K. Debray, N.-W. Lin & M. V. Hermenegildo (1990): *Task Granularity Analysis in Logic Programs*. In: *Proc. PLDI'90*, ACM, pp. 174–188, doi:10.1145/93542.93564.

[4] S. K. Debray, P. Lopez-Garcia, M. V. Hermenegildo & N.-W. Lin (1997): *Lower Bound Cost Estimation for Logic Programs*. In: *ILPS'97*, MIT Press, pp. 291–305, doi:10.7551/mitpress/4283.001.0001.

[5] Trevor Hastie, Robert Tibshirani & Jerome Friedman (2009): *The Elements of Statistical Learning: Data Mining, Inference and Prediction*, second edition. Springer New York, NY, doi:10.1007/978-0-387-84858-7.

[6] Trevor Hastie, Robert Tibshirani & Martin Wainwright (2015): *Statistical Learning with Sparsity: The Lasso and Generalizations*. Chapman & Hall/CRC, doi:10.1201/b18401.

[7] M. Hermenegildo, G. Puebla, F. Bueno & P. Lopez Garcia (2005): *Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)*. *Science of Computer Programming* 58(1–2), pp. 115–140, doi:10.1016/j.scico.2005.02.006.

[8] P. Lopez-Garcia, L. Darmawan, M. Klemen, U. Liqat, F. Bueno & M. V. Hermenegildo (2018): *Interval-based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption*. *TPLP* 18(2), pp. 167–223, doi:10.1017/S1471068418000042.

[9] P. Lopez-Garcia, M. Klemen, U. Liqat & M. V. Hermenegildo (2016): *A General Framework for Static Profiling of Parametric Resource Usage*. *TPLP (ICLP'16 Special Issue)* 16(5-6), pp. 849–865, doi:10.1017/S1471068416000442.

[10] (2023): *Wolfram Mathematica (v13.2): the World's Definitive System for Modern Technical Computing*. `https://www.wolfram.com/mathematica`. Accessed: May 25, 2023.

[11] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason Keith Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman & Anthony Scopatz (2017): *SymPy: symbolic computing in Python*. *PeerJ Computer Science* 3, p. e103, doi:10.7717/peerj-cs.103.

[12] Leonardo Mendonça de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, *Lecture Notes in Computer Science* 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[13] J. Navas, E. Mera, P. Lopez-Garcia & M. Hermenegildo (2007): *User-Definable Resource Bounds Analysis for Logic Programs*. In: *Proc. of ICLP'07*, *LNCS* 4670, Springer, pp. 348–363, doi:10.1007/978-3-540-74610-2_24.

[14] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot & Edouard Duchesnay (2011): *Scikit-learn: Machine Learning in Python*. *Journal of Machine Learning Research* 12, pp. 2825–2830, doi:10.5555/1953048.2078195. Available at `https://dl.acm.org/doi/10.5555/1953048.2078195`.

[15] A. Podelski & A. Rybalchenko (2004): *A Complete Method for the Synthesis of Linear Ranking Functions*. In: *VMCAI'04*, *LNCS* 2937, Springer, pp. 239–251, doi:10.1007/978-3-540-24622-0_20.

[16] M. Rosendahl (1989): *Automatic Complexity Analysis*. In: *4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, ACM Press, pp. 144–156, doi:10.1145/99370.99381.

[17] A. Serrano, P. Lopez-Garcia & M. V. Hermenegildo (2014): *Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types*. *TPLP, ICLP'14 Special Issue* 14(4-5), pp. 739–754, doi:10.1017/S147106841400057X.

[18] B. Wegbreit (1975): *Mechanical Program Analysis*. *Communications of the ACM* 18(9), pp. 528–539, doi:10.1145/361002.361016.

[19] (2023): *Z3 API in Python*. `https://ericpony.github.io/z3py-tutorial`. Accessed: May 25, 2023.

On the Independencies Hidden in the Structure of a Probabilistic Logic Program

Kilian Rückschloß
Ludwig-Maximilians-Universität München
Oettingenstraße 67, 80538 München, Germany

`kilian.rueckschloss@lmu.de`

Felix Weitkämper
Ludwig-Maximilians-Universität München
Oettingenstraße 67, 80538 München, Germany

`felix.weitkaemper@lmu.de`

Pearl and Verma developed d-separation as a widely used graphical criterion to reason about the conditional independencies that are implied by the causal structure of a Bayesian network. As acyclic ground probabilistic logic programs correspond to Bayesian networks on their dependency graph, we can compute conditional independencies from d-separation in the latter.

In the present paper, we generalize the reasoning above to the non-ground case. First, we abstract the notion of a probabilistic logic program away from external databases and probabilities to obtain so-called program structures. We then present a correct meta-interpreter that decides whether a certain conditional independence statement is implied by a program structure on a given external database. Finally, we give a fragment of program structures for which we obtain a completeness statement of our conditional independence oracle. We close with an experimental evaluation of our approach revealing that our meta-interpreter performs significantly faster than checking the definition of independence using exact inference in ProbLog 2.

1 Introduction

A probabilistic logic program is a logic program, in which each clause holds with a specified probability. The most common semantics for these programs is the distribution semantics [13], which assigns to each ground program a joint probability distribution over the atoms occurring in it. It is the basis for many programming languages such as the Independent Choice Logic [9], PRISM [14], Logic Programs with Annotated Disjunctions [15] and ProbLog [1].

Since conditional independence is a rather fundamental notion in probability theory, it is natural to ask for its counterpart in probabilistic logic programming. Moreover, considering the work of Holtzen et al. [4], such an analysis may contribute to speed up lifted inference. In this paper, we extend the effort of Rückschloß and Weitkämper [11] to establish a calculus deriving the conditional independencies that are determined by the clause structure of a ProbLog program. Let us illustrate this problem in the following example:

Example 1. *We consider storages, which consist of rooms, tanks, employees and liquids that are given by the unary predicates* $room/1$*,* $tank/1$*,* $employee/1$*,* $liquid/1$*, respectively. For each storage a database table* $passage(R,R')$ *tells us, between which rooms* R *and* R' *we find a passage. Moreover, in each room* R *we find tanks* T*, denoted by* $in(T,R)$ *and a tank* T *may store a liquid* L*, denoted by* $stores(T,L)$*. Finally, we assume a database table* $flammable(L)$*, indicating the flammable liquids. In this context, it makes sense to assume that each tank stores at most one liquid which results in the following integrity constraint:*

$$\bot \leftarrow tank(T), liquid(L_1), liquid(L_2), L_1 \neq L_2, stores(T,L_1), stores(T,L_2). \tag{Const}$$

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 169–182, doi:10.4204/EPTCS.385.17

With a certain probability _ we find an employee E opening a tank T, denoted by $opens(E,T)$. *This leads to the following ProbLog clause:*

$$\_ :: opens(E,T) \leftarrow employee(E), tank(T). \tag{RC1}$$

Further, if employee E opens tank T, it may be that E does not close the tank T properly, which then causes the tank T to leak, denoted by $leaks(T)$. *Again we assume that we don't know the exact probabilities and capture the mechanism in the following clause:*

$$\_ :: leaks(T) \leftarrow employee(E), tank(T), opens(E,T). \tag{RC2}$$

Moreover, an employee E may smoke in a room R, denoted by $smokes(E,R)$:

$$\_ :: smokes(E,R) \leftarrow employee(E), room(R). \tag{RC3}$$

If employee E smokes in a room R, which contains a leaking tank storing an flammable liquid, this may cause a fire in room R. Further, the smoke immediately spreads to the rooms R_1, *which are connected to the room R by passages, where it may trigger a sensor. In this case, we observe the event* $fire(R_1)$. *This mechanism is captured in the clauses below:*

$$\begin{aligned}
&connected(R,R) \leftarrow room(R).\\
&connected(R,R_1) \leftarrow room(R), room(R_1), room(R_2), passage(R_2,R_1),\\
&\quad connected(R,R_2). && \text{(Int)}\\
&\_ :: fire(R_1) \leftarrow room(R_1), room(R), employee(E), tank(T), liquid(L), in(T,R),\\
&\quad (connected(R,R_1); connected(R_1,R)), stores(T,L), flammable(L),\\
&\quad smokes(E,R), leaks(T). && \text{(RC4)}
\end{aligned}$$

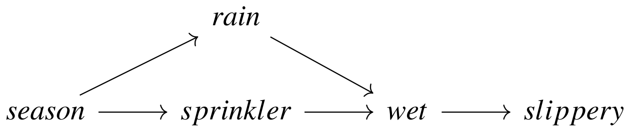
Given (RC1), (RC2), (RC3), (RC4), (Const) and (Int) together with a concrete storage, we aim to answer queries about conditional independencies. For instance, we want to determine whether the event $smokes(e1,r)$ *of employee* $e1$ *smoking in room* r *is independent of the event* $opens(e2,t)$ *of employee* $e2$ *opening the tank* t *without knowing the probabilities of the clauses (RC1), (RC2), (RC3) and (RC4). Further, we would also like to predict how things change if we observe* $fire(r_1)$ *a fire in room* r_1.

We accomplish our goal by grounding ProbLog programs to Bayesian networks, where we can apply the theory of d-separation [16] to derive the desired independence statements. In this way we can efficiently reason about conditional independencies. Further, we highlight that the theory of d-separation lies at the basis of constraint based causal structure discovery, i.e. it enables us to reason about causal relationships on the basis of observational data. Hence, we suppose that the present work also serves as a starting point in the development of causal structure discovery techniques for probabilistic logic programming.

2 On the Independencies Hidden in a Propositional Causal Structure

At the beginning, we discuss how conditional independencies can be inferred from a causal structure in the propositional case. Here, we identify a **causal structure** on a set of random variables $\mathbf{V}$ with a directed acyclic graph G, i.e. a partial order, on $\mathbf{V}$. The intuition is that X is a **cause** of Y if there is a directed path from X to Y in G. In this case, we also say that Y is an **effect** of X. Further, we say that X is a **direct cause** of Y if the edge $X \to Y$ exists in G, i.e. if and only if the node $X \in \mathrm{Pa}(Y)$ lies in the set $\mathrm{Pa}(Y)$ of **parents** of Y.

Example 2. *Consider a road that passes by a field with a sprinkler in it. The sprinkler is switched on by a weather sensor and the pavement of the road may be wet, denoted by wet, because the sprinkler is on, denoted $sprinkler$ or because it rains, denoted by $rain$. Further, we know that the events rain and $sprinkler$ are caused by the season, denoted by $season$, as they both are triggered by the weather. Finally, we note that a wet road is more likely to be slippery, denoted by $slippery$. The situation above gives rise to the following causal structure on the random variables* $\mathbf{V} := \{season, rain, sprinkler, wet, slippery\}$*:*



(1)

In particular, we find that $season$ is a cause of $slippery$ but not a direct cause, whereas wet is a direct cause of $slippery$. Further, there is no causal relationship between $sprinkler$ and $rain$.

Next, a given probability distribution π on the random variables $\mathbf{V}$ is consistent with a causal structure G if the influence of any cause X on an effect Y is moderated by the direct causes of Y. This intuition is formally captured in the following definition:

Definition 1 (Markov Condition)**.** *We say that the distribution π on the set of random variables $\mathbf{V}$ satisfies the* ***Markov condition*** *with respect to a causal structure G on $\mathbf{V}$, if every random variable $X \in \mathbf{V}$ is independent of its causes in G, once we observe its direct causes* $\mathrm{Pa}(X)$*. In this case, we write $\pi \models G$.*

Example 3. *In Example 2 the Markov condition states for instance that the influence of season on the event $slippery$ is completely moderated by the event wet. Once we know that the pavement of the road is wet, we expect it to be slippery regardless of the event that caused the road to be wet.*

If a distribution $\pi \models G$ satisfies the Markov condition with respect to a given causal structure G, it is represented by a Bayesian network on G and vice versa [8, §1.2.3]:

Definition 2 (Bayesian Network)**.** *A* ***Bayesian network*** *on a set of random variables $\mathbf{V}$ consists of a causal structure G on $\mathbf{V}$ and the probability distributions $\pi(X|\mathrm{Pa}(X))$ of the random variables $X \in \mathbf{V}$ conditioned on their direct causes in G. A Bayesian network gives rise to a joint probability distribution on $\mathbf{V} = \{X_1, ..., X_k\}$ by setting*

$$\pi(X_1 = x_1, ..., X_k = x_k) := \prod_{i=1}^{k} \pi(X_i = x_i|\,\mathrm{pa}(X_i)),$$

where $\mathrm{pa}(X_i) := \{X_j = x_j | X_j \in \mathrm{Pa}(X_i)\}$.

The Markov condition equips a causal structure with a semantics that is given by conditional independence statements. Further, Verma and Pearl [16] derive d-separation as a criterion to compute all conditional independencies that follow if we apply the Markov condition to a given causal structure.

Definition 3 (d-Separation)**.** *Let G be a directed acyclic graph, i.e. it is a causal structure. An* ***undirected path*** *P between two nodes A and B is an alternating sequence of nodes and edges*

$$P = R_0 \overset{E_1}{-} R_1 \overset{E_2}{-} R_2 \overset{E_3}{-} \dots \overset{E_{n-1}}{-} R_{n-1} \overset{E_n}{-} R_n,$$

where $E_i \in \{R_{i-1} \to R_i,\ R_{i-1} \leftarrow R_i\}$ for all $1 \leq i \leq n$. We call a node R_i of P a ***collider*** *if P is of the form $\dots \to R_i \leftarrow \dots$, otherwise R_i is said to be a* ***non-collider*** *of P.*

Further, let $\mathbf{Z}$ *be a set of nodes. We say that a node* N *is* ***blocked*** *by* $\mathbf{Z}$ *if it lies in* $\mathbf{Z}$*, i.e. if we have that* $N \in \mathbf{Z}$*. Moreover,* N *is said to be* ***activated*** *by* $\mathbf{Z}$ *if there exists a directed path from* N *to a node in* $\mathbf{Z}$*. The undirected path* P *is a* ***d-connecting path*** *with respect to the* ***observations*** $\mathbf{Z}$ *if every non-collider* N *of* P *is not blocked and if every collider* C *of* P *is activated.*

We say that $\mathbf{Z}$ ***d-connects*** *two nodes* A *and* B *if there exists a d-connecting path between* A *and* B *with respect to* $\mathbf{Z}$*. Otherwise, we say that* $\mathbf{Z}$ ***d-separates*** A *and* B*. Finally, two sets of nodes* $\mathbf{A}$ *and* $\mathbf{B}$ *are said to be* ***d-separated*** *by* $\mathbf{Z}$ *if* $\mathbf{Z}$ *d-separates* A *and* B *for every* $A \in \mathbf{A}$ *and every* $B \in \mathbf{B}$*. Otherwise, the sets* $\mathbf{A}$ *and* $\mathbf{B}$ *are* ***d-connected*** *by* $\mathbf{Z}$*.*

Note that the term “d-connected” is a shorthand for “directionally connected” [3].

Example 4. *Let us consider the causal structure (1) again and take* $\mathbf{Z} := \{slippery\}$ *for the observations. We find the following d-connecting path* $P := season \to rain \to wet \leftarrow sprinkler$*. The intuition behind this d-connecting path is as follows:*

Assume we observe the event $\mathbf{Z}$*. We know that this increases the probability for wet, which itself is triggered by rain or sprinkler. If we additionally suppose that it is summer, this decreases the probability for rain, which increases the probability of sprinkler as we have an increased probability for wet. To summarize we expect season and sprinkler to be dependent once we observed slippery.*

Note that the argument above does not go through anymore, if we observe additionally that it rains or if we observe nothing.

The reasoning of Example 4 is now formalized in the following theorem.

Theorem 1 (Verma and Pearl [16])**.** *Let* G *be a causal structure on the set* $\mathbf{V}$*, let* $\mathbf{Z} := \{Z_1, ..., Z_n\} \subseteq \mathbf{V}$ *be a subset of nodes and let* $A, B \in \mathbf{V}$ *be nodes of* G*. If* $\mathbf{Z}$ *d-separates* A *and* B*, we obtain that* A *and* B *are independent conditioned on* $\mathbf{Z}$ *in every distribution* $\pi \models G$*, which is Markov to* G*. Here, being conditionally independent means that*

$$\begin{aligned}&\forall_{a \text{ value of } A}\ \forall_{b \text{ value of } B}\ \forall_{z_1,...,z_n \text{ values of } Z_1,...,Z_n}:\\ &\pi(A=a, B=b \mid \{Z_i = z_i\}_{i=1}^{n}) = \pi(A=a \mid \{Z_i = z_i\}_{i=1}^{n}) \cdot \pi(B=b \mid \{Z_i = z_i\}_{i=1}^{n}). \ \square\end{aligned} \tag{2}$$

However, McDermott [5] demonstrates that in general d-separation does not yield a complete independence oracle. This observation motivates the following definition.

Definition 4 (Faithfulness)**.** *A distribution* π *is* ***(causally) faithful*** *to a causal structure* G *on* $\mathbf{V}$ *if it is Markov to* G *and if every conditional independence of two random variables* $A, B \in \mathbf{V}$ *with respect to a set of observations* $\mathbf{Z} \subseteq \mathbf{V}$ *can be derived from d-separation by Theorem 1.*

Fortunately, Meek [6] shows that faithfulness holds for almost all Boolean Bayesian networks in the following sense:

Theorem 2. *Let* G *be a causal structure and let* $\theta \in [0,1]^n$ *be the vector, which determines the conditional distributions that turn* G *into a Boolean Bayesian network representing the distribution* π*. In this case we obtain finitely many non-trivial polynomial equations such that* π *is faithful to* G *unless* θ *solves one of these equations.* $\square$

Note that Theorem 2 states that d-separation enables us to derive all conditional independence statements that are implied by a causal structure under the Markov condition.

Finally, we identify a causal structure G with the database that contains a fact `X ---> Y` for every edge $X \to Y$ in G. In this case the predicate `dseparates/3` in following meta-interpreter decides whether two nodes X and Y are d-separated by a list of observations $\mathbf{Z}$.

Program 1 (Deciding d-Separation)**.**

```
% Implement hactivates/2 as the transitive closure of (--->)/2 and
% calculate activated nodes
hactivates(X,X).  hactivates(Z,X):-(Y ---> Z), hactivates(Y,X).
activates([Z|_],X):-hactivates(Z,X).  activates([_|TZ],X):-activates(TZ,X).
% Implement dconnects/3 by case distinction over the last orientation
dconnects(X,Y,Z):-\+member(X,Z),  \+member(Y,Z), dconnects(X,Y,Z,_).
dconnects(X,Y,_,right):-(X--->Y).  dconnects(X,Y,_,left):-(Y--->X).
dconnects(X,Y,Z,right):-(Y1--->Y),\+member(Y1,Z),dconnects(X,Y1,Z,right).
dconnects(X,Y,Z,right):-(Y1--->Y),\+member(Y1,Z),dconnects(X,Y1,Z,left).
dconnects(X,Y,Z,left):-(Y--->Y1),\+member(Y1,Z),dconnects(X,Y1,Z,left).
dconnects(X,Y,Z,left):-(Y--->Y1),activates(Z,Y1),dconnects(X,Y1,Z,right).
% dseparates/3 is the complement of dconnects/3
dseparates(X,Y,Z):-\+dconnects(X,Y,Z).
```

3 A Formalism for Lifted Probabilistic Logic Programming

Recall that events of the trivial probabilities zero and one are independent of every other event. To overcome this obstruction, we introduce a language which separates logical predicates, denoting logical statements with probabilities zero and one, from random predicates, denoting events with a probability lying between zero and one.

Let us fix a **query language** i.e. a language $\mathfrak{Q} \supseteq \mathfrak{L} \supseteq \mathfrak{E}$ in three parts with an **external vocabulary** $\mathfrak{E}$ and a **logical vocabulary** $\mathfrak{L}$. Here, $\mathfrak{Q}$ is a finite relational vocabulary with equality $\doteq$, i.e. it consists of a finite set of relation symbols, a finite set of constants as well as a countably infinite set of variables. Further, $\mathfrak{L}$ is a subvocabulary of $\mathfrak{Q}$ containing all of the variables and constants of $\mathfrak{Q}$ as well as a (possibly empty) subset of the relation symbols of $\mathfrak{Q}$. Moreover, $\mathfrak{E}$ is a subvocabulary of $\mathfrak{L}$, which satisfies the same properties in $\mathfrak{L}$ as $\mathfrak{L}$ does regarding $\mathfrak{Q}$.

Example 5. *In Example 1 the vocabulary* $\mathfrak{E}$ *consists of the predicates* $room/1$, $employee/1$, $tank/1$, $liquid/1$, $passage/2$, $in/2$, $stores/2$ *and* $flammable/1$, *which we assume to be given by a database. Further,* $\mathfrak{L}$ *extends* $\mathfrak{E}$ *by the predicate* $connected/2$, *which is deterministically defined in terms of the predicates of* $\mathfrak{E}$. *Finally,* $\mathfrak{Q}$ *extends* $\mathfrak{L}$ *by the predicates* $opens/2$, $leaks/1$, $smokes/2$ *and* $fire/1$, *which we expect to denote non-deterministic random variables.*

As usual, an **atom** is an expression of the form $r(t_1,\ldots,t_n)$ or $t_1 \doteq t_2$, where r is a relation symbol and t_1 to t_n are constants or variables, and a **literal** is an expression of the form A or $\neg A$ for an atom A. It is called an **external atom** or **literal** if r is in $\mathfrak{E}$, a **logical atom** or **literal** if r is in $\mathfrak{L}$, an **internal atom** or **literal** if r is in $\mathfrak{L} \setminus \mathfrak{E}$ and a **random atom** or **literal** if r is in $\mathfrak{Q} \setminus \mathfrak{L}$. Here, we regard equality $\doteq$ as a relation in $\mathfrak{E}$. A literal of the form A is called **positive** and a literal of the form $\neg A$ is called **negative**. A literal L is said to be **ground** if no variable occurs in it. Finally, we use $\mathrm{var}(E)$ to refer to the variables occurring in a given expression E.

Example 6. *In Example 5* $passage(R,R')$ *is an external atom, whereas* $connected(R,R')$ *is an internal atom and* $fire(R)$ *is a random atom.*

Formulas, as well as **existential** and **universal formulas** are defined as usual in first-order logic. The logical vocabulary will be used to formulate constraints and conditions for our probabilistic logic program. The purpose of a probabilistic logic program, however, is to define distributions for the random variables determined by the language $\mathfrak{Q}$. This is done by so-called ProbLog clauses.

Definition 5 (ProbLog Clause)**.** *A **generalized ProbLog clause** RC is an expression of the form*

$$(\_ :: R \leftarrow R_1,...,R_m,L_1,...,L_n.) = (\_ :: \mathrm{effect}(RC) \leftarrow \mathrm{causes}(RC) \cup \mathrm{cond}(RC)),$$

which is given by the following data:

i) a random atom $R := \mathrm{effect}(RC)$*, called the **effect** of RC*

ii) a finite and possibly empty set of random literals $\mathrm{causes}(RC) := \{R_1,...,R_m\}$*, called the **causes** of RC*

iii) a finite and possibly empty set of logical literals $\mathrm{cond}(RC) := \{L_1,...,L_n\}$*, called the **condition** of RC*

*We call RC **positive** if the set of causes* $\mathrm{causes}(RC)$ *contains only positive literals. Further, we obtain a **ProbLog clause*** $RC^{\pi(RC)} := (\pi(RC) :: R \leftarrow R_1,...,R_m,L_1,...,L_n.)$ *from the generalized ProbLog clause RC by choosing a **probability*** $\pi(RC) \in [0,1]$.

In i) and ii) of Definition 5 we use the terminology of cause and effect to reflect that under our semantics, a ground program represents a functional causal model [8, §1.4].

Example 7. *Note that (RC4) of Example 1 yields a generalized ProbLog clause. Further, if we choose the probability* $\pi(RC4) := 0.6$ *we obtain the ProbLog clause* $RC4^{0,6}$ *below.*

$$\begin{aligned}0.6 :: &fire(R1) \leftarrow room(R1), room(R), employee(E), tank(T), liquid(L), in(T,R),\\ &(connected(R,R1); connected(R1,R)), stores(T,L), flammable(L),\\ &smokes(E,R), leaks(T).\end{aligned} \tag{3}$$

After having established the necessary syntax we proceed to the semantics. Let us begin with the logical expressions. The semantics of logical expressions is given in a straightforward way.

We highlight the unique names assumption in our definition of a structure:

An $\mathfrak{L}$**-structure** Λ consists of a domain Δ, an element of Δ for every constant in $\mathfrak{L}$, *such that two different constants are mapped to different elements*, and an n-ary relation on Δ for every relation symbol of arity n in $\mathfrak{L}$.

Whether a logical formula is **satisfied** by a given $\mathfrak{L}$-structure (under a given **interpretation** of its free variables) is determined by the usual rules of first-order logic. Finally, note that the semantics of external expressions is defined analogously.

For the semantics of clauses and programs we choose the FCM-semantics [12] since it directly relates an acyclic ground program to a Bayesian network. We start with the definition of a lifted program:

Definition 6 (Lifted Program and Program Structure)**.** *A **program structure*** $\boldsymbol{P} := (\boldsymbol{R},\boldsymbol{I},\boldsymbol{C})$ *is a triple, which consists of the following data*

i) a finite set of integrity constraints $\boldsymbol{C}(\boldsymbol{P}) := \boldsymbol{C}$ *of the form* $(\bot \leftarrow L_1,...,L_k.)$ *for logical literals* L_i *with* $1 \leq i \leq k$*, which we call the **constraints** of* $\boldsymbol{P}$*.*

ii) a finite set of normal clauses $\boldsymbol{I}(\boldsymbol{P}) := \boldsymbol{I}$ *of the form* $(H \leftarrow B_1,...,B_m.)$ *with logical literals* $B_1,...,B_m$ *and an internal atom H, which we call the **internal part** of* $\boldsymbol{P}$*.*

iii) a finite set of generalized ProbLog clauses $\boldsymbol{R}(\boldsymbol{P}) := \boldsymbol{R}$*, which we call the **random part** of* $\boldsymbol{P}$*.*

We say that $\boldsymbol{P}$ *is* ***stratified*** *if its internal part* $\boldsymbol{I}$ *is a stratified set of normal clauses and we say that* $\boldsymbol{P}$ *is* ***positive*** *if every generalized ProbLog clause of its random part* $\boldsymbol{R}(\boldsymbol{P})$ *is positive.*

A ***choice of parameters*** *for the program structure* $\boldsymbol{P}$ *is a function* $\pi : \boldsymbol{R}(\boldsymbol{P}) \rightarrow [0,1]$. *A program structure* $\boldsymbol{P}$ *and a choice of parameters* π *yield a* ***(lifted) program*** $\boldsymbol{P}^\pi := (\boldsymbol{R}(\boldsymbol{P})^\pi, \boldsymbol{I}(\boldsymbol{P}), \boldsymbol{C}(\boldsymbol{P}))$, *where* $\boldsymbol{R}(\boldsymbol{P})^\pi$ *is the set of ProbLog clauses* $\pi(RC) :: \mathrm{effect}(RC) \leftarrow \mathrm{causes}(RC) \cup \mathrm{cond}(RC)$ *for* $RC \in \boldsymbol{R}(\boldsymbol{P})$. *In this case,* $\boldsymbol{C}(\boldsymbol{P})$ *are the* ***constraints****,* $\boldsymbol{I}(\boldsymbol{P})$ *is the* ***internal part*** *and* $\boldsymbol{R}(\boldsymbol{P})^\pi$ *is the* ***random part*** *of the program* $\boldsymbol{P}^\pi$. *Further,* $\boldsymbol{P}$ *is called the* ***structure*** *of the program* $\boldsymbol{P}^\pi$. *Finally, the program* $\boldsymbol{P}^\pi$ *is* ***stratified*** *or* ***positive*** *if* $\boldsymbol{P}$ *is.*

Example 8. *In Example 1 we obtain a stratified program structure* $\boldsymbol{P}$ *with random part (RC1), (RC2), (RC3), (RC4), internal part (Int) and constraints (Const). Further, we obtain a choice of parameters* π *by assigning* $\pi(RC1) := 0.8$, $\pi(RC2) := 0.1$, $\pi(RC3) := 0.5$ *and* $\pi(RC4) := 0.05$. *Finally, the choice of parameters* π *gives rise to the following (lifted) program* $\boldsymbol{P}^\pi$*:*

```
%Random part
0.8 :: opens(E,T) :- employee (E), tank(T).
0.1 :: leaks(T) :- employee(E), tank(T), opens(E,T).
0.5 :: smokes(E,R) :- employee(E), room(R).
0.05 :: fire(R1) :- employee(E), room(R), room(R1), tank(T), liquid(L),
        flammable(L), in(T,R), stores(T,L), (connected(R,R1); connected(R1,R)),
        smokes(E,R),leaks(T).
%Internal part
connected(R,R):-room(R).
connected(R,R1) :- room(R), room(R1), room(R2), passage(R2,R1),
        connected(R, R2).
%Constraints
:- tank(T), liquid(L1), liquid(L2), L1 \= L2, stores(T,L1), stores(T,L2).
```

Definition 7 (Ground Variable and External Database)**.** *Let* $\boldsymbol{P}$ *be a stratified program structure and let* $\mathscr{E}$ *be an* $\mathfrak{E}$*-structure. In our setting, we may assume without loss of generality that* $\mathscr{E}$ *is a Herbrand model of a language* $\mathfrak{E}^*$*, which extends the external language* $\mathfrak{E}$ *by constants. Further, denote by* $\mathfrak{L}^*$ *and* $\mathfrak{Q}^*$ *respectively the extension of the languages* $\mathfrak{L}$ *and* $\mathfrak{Q}$ *by the new constants in* $\mathfrak{E}^*$*. We write* $\mathscr{E}^{\boldsymbol{I}} := \mathscr{E}^{\boldsymbol{I}(\boldsymbol{P})} := \{L \text{ ground atom of } \mathfrak{L}^* : \boldsymbol{I} \cup \mathscr{E} \models L\}$ *for the minimal Herbrand model of* $\boldsymbol{I} \cup \mathscr{E}$*, which is the result of applying the stratified Datalog program* $\boldsymbol{I}$ *to* $\mathscr{E}$*.*

Further, we call $\mathscr{E}$ *an* ***external database*** *of the program structure* $\boldsymbol{P}$ *if it satisfies the constrains of* $\boldsymbol{P}$ *after applying the Datalog program* $\boldsymbol{I}$ *to* $\mathscr{E}$*, i.e. if*

$$\mathscr{E}^{\boldsymbol{I}} \models \bigwedge_{\substack{\bot \leftarrow L_1,\dots,L_n \in \boldsymbol{C}(\boldsymbol{P}) \\ \kappa \text{ interpretation on } \operatorname{var}(L_1,\dots,L_n)}} \neg\left(\bigwedge_{i=1}^{n} L_i^\kappa\right).$$

A ***ground variable*** *is a ground atom* $G := r(x_1,\dots,x_n)$ *of* $\mathfrak{Q}^*$ *with a random predicate* $r \in \mathfrak{Q}$*. Finally, we write* $\mathscr{G}(\mathscr{E})$ *for the set of all ground variables.*

The term ground variable indicates that we expect G to denote a proper random variable under our semantics. From now on we restrict ourselves to the study of stratified program structures. Hence, let us fix a stratified program structure **P** for the rest of this work.

Definition 8 (FCM-Semantics of Lifted Programs)**.** *Let π be a choice of parameters and let $\mathscr{E}$ be an external database for $\boldsymbol{P}$. The* ***grounding*** $\boldsymbol{P}^{(\pi,\mathscr{E})}$ *of the program structure* $\boldsymbol{P}$ *with respect to π and $\mathscr{E}$ is the system of Boolean equations given by*

$$G := \bigvee_{\substack{RC\in \boldsymbol{R}(\boldsymbol{P}) \\ \kappa \text{ interpretation on } \mathrm{var}(RC) \\ \mathrm{effect}(RC)^{\kappa}=G \\ \left(\mathscr{E}^{\boldsymbol{I}},\kappa\right)\models \mathrm{cond}(RC)}} \left(\bigwedge_{C\in \mathrm{causes}(RC)} C^{\kappa} \wedge u(RC,\kappa) \right) \tag{4}$$

for every ground variable $G \in \mathscr{G}(\mathscr{E})$. Here, the ***error term*** $u(RC,\kappa)$ *is a distinct Boolean random variable with the distribution $\pi(u(RC,\kappa)) = \pi(RC)$ for every generalized ProbLog clause $RC \in \boldsymbol{R}(\boldsymbol{P})$ and every variable interpretation κ on $\mathrm{var}(RC)$. Besides, the error terms $u(RC,\kappa)$ are assumed to be mutually independent. Finally, the* ***grounding*** *of the program $\boldsymbol{Q} := \boldsymbol{P}^{\pi}$ is given by $\boldsymbol{Q}^{\mathscr{E}} := \boldsymbol{P}^{(\pi,\mathscr{E})}$.*

Example 9. *It is easy to observe that the program structure* $\boldsymbol{P}$ *of Example 8 is indeed a stratified program structure in our sense. Now assume we are given a specific storage, which consists of four rooms* `r1`, `r2`, `r3` *and* `r4`*. These rooms are connected by passages as follows:*

`passage(r1,r2), passage(r2,r3)`

Moreover, we have five tanks `t1`, `t2`, `t3`, `t4` *and* `t5` *with*

`in(t1,r1), in(t2,r2), in(t3,r3), in(t4,r4), in(t5,r4).`

The tanks contain two types of liquids `gasoline` *and* `water`*, which we describe in the following way:*

`stores(gasoline,t1), stores(gasoline, t2), stores(water, t3),`
`stores(water, t4), stores(gasoline,t5), flammable(gasoline)`

Finally, assume there are two employees Mary and John, which we express simply as `mary` *and* `john`*. In this case, one checks that the storage above satisfies the integrity constraint (Const), i.e. we are given an external database $\mathscr{E}$ for the program structure* $\boldsymbol{P}$*.*

Further, let π be the choice of parameters in Example 8. For $e \in \{john, mary\}$, $t \in \{t_1,t_2,t_3,t_4,t_5\}$ and $r \in \{r_1,r_2,r_3,r_4\}$ the grounding $\boldsymbol{P}^{(\pi,\mathscr{E})}$ is given by the equations of the form:

- $opens(e,t) := u((\_ :: opens(E,T) \leftarrow employee(E), tank(T).), \{E \mapsto e, T \mapsto t\})$ *such that* $u((\_ :: opens(E,T) \leftarrow employee(E), tank(T).), \{E \mapsto e, T \mapsto t\})$ *is true with probability* 0.8
- $leaks(t) := opens(e,t) \wedge u(RC2, \{E \mapsto e, T \mapsto t\})$ *such that* $u(RC2, \{E \mapsto e, T \mapsto t\})$ *is true with probability* 0.1
- $smokes(e,r) := u(RC3, \{E \mapsto e, R \mapsto r\})$ *such that* $u(RC3, \{E \mapsto e, R \mapsto r\})$ *is true with probability* 0.5
- $fire(r_i) := \bigvee_{j=1}^{3} smokes(e,r_j) \wedge leaks(t_j) \wedge u(RC4, \{R_1 \mapsto r_i, E \mapsto e, R \mapsto r_j, T \mapsto t_j\})$ *such that* $u(RC4, \{R_1 \mapsto r_i, E \mapsto e, R \mapsto r_j, T \mapsto t_j\})$ *is true with probability* 0.05 *for* $1 \leq i,j \leq 3$
- $fire(r_4) := smokes(e,r_4) \wedge leaks(t_5) \wedge u(RC4, \{R_1 \mapsto r_4, E \mapsto e, R \mapsto r_4, T \mapsto t_5\})$ *such that* $u(RC4, \{R_1 \mapsto r_4, E \mapsto e, R \mapsto r_4, T \mapsto t_5\})$ *is true with probability* 0.05

In the present paper, we reason on a syntactic level about the conditional independencies implied by the program structure **P** and an external database $\mathscr{E}$. We proceed as Geiger and Pearl [2] and restrict ourselves to those independence statements following from d-separation in the corresponding propositional causal structures, which we call ground graphs.

Definition 9 (Ground Graph and Acyclicity)**.** *Let $\mathscr{E}$ be an external database for $\boldsymbol{P}$. We define the **ground graph** $\mathrm{Graph}_{\mathscr{E}}(\boldsymbol{P})$ to be the directed graph on the set of ground variables $\mathscr{G}(\mathscr{E})$, which is obtained by drawing an edge $G_1 \to G_2$ if and only if there exists a generalized ProbLog clause $RC \in \boldsymbol{R}(\boldsymbol{P})$, a cause $C \in \mathrm{causes}(RC)$ and a variable interpretation ι on $\mathrm{var}(C) \cup \mathrm{var}(\mathrm{effect}(RC))$ such that the following assertions are satisfied:*

i) $\left(\mathscr{E}^{\boldsymbol{I}}, \iota\right) \models \exists_{\mathrm{var}(\mathrm{cond}(RC))\setminus(\mathrm{var}(C)\cup\mathrm{var}(\mathrm{effect}(RC)))}\, \mathrm{cond}(RC)$

ii) $C^{\iota} \in \{G_1, \neg G_1\}$

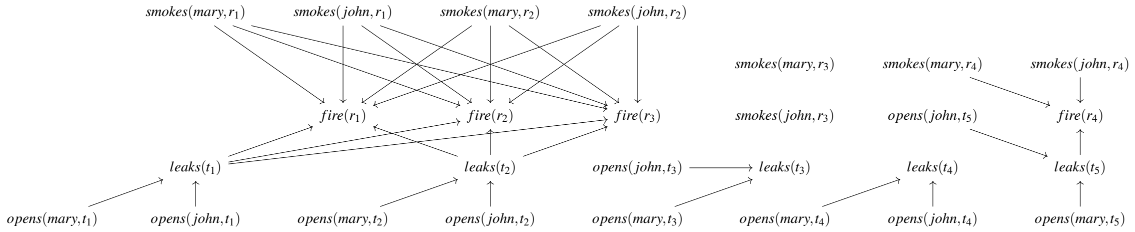
iii) $\mathrm{effect}(RC)^{\iota} = G_2$.

*In this case we say that the edge $G_1 \to G_2$ is **induced** by RC. Moreover, we call $\boldsymbol{P}$ an **acyclic** program structure if $\mathrm{Graph}_{\mathscr{E}}(\boldsymbol{P})$ is a directed acyclic graph i.e. a causal structure for every external database $\mathscr{E}$.*

From now on we assume the program structure **P** to be acyclic. In this case it is easy to see that the grounding $\mathrm{P}^{(\pi,\mathscr{E})}$ yields a unique expression for every ground variable $G \in \mathscr{G}(\mathscr{E})$ in terms of the error terms $u(RC,\kappa)$ for every choice of parameters π and for every external database $\mathscr{E}$. Hence, it induces a unique probability distribution on $\mathscr{G}(\mathscr{E})$. Rückschloß and Weitkämper [12] further show that this distribution coincides with the distribution semantics of the ProbLog program $\mathbf{P}^{\pi} \cup \mathscr{E}$. Overall Definition 6 yields a causal generalization of the standard semantics for acyclic ProbLog programs. Further, we obtain the following result:

Proposition 1 (Rückschloß and Weitkämper [12])**.** *For every external database $\mathscr{E}$ and for every choice of parameters π the grounding $\boldsymbol{P}^{(\pi,\mathscr{E})}$ yields a distribution, which is Markov to the ground graph $\mathrm{Graph}_{\mathscr{E}}(\boldsymbol{P})$.* □

Example 10. *The program structure $\boldsymbol{P}$ of Example 8 is indeed acyclic in our sense. Further, in the situation of Example 9 we obtain the following ground graph.*



As we now defined the necessary refinement of the ProbLog language [1], we can return to reasoning about conditional independence.

4 A Symbolic Calculus for Conditional Independencies

By Proposition 1, for every external database $\mathscr{E}$ the causal information about the conditional independencies implied by our program structure **P** lies in the ground graph $\mathrm{Graph}_{\mathscr{E}}(\mathbf{P})$. Hence, applying Theorem 1 we obtain a correct conditional independence oracle if we combine a Prolog representation of the ground graph $\mathrm{Graph}_{\mathscr{E}}(\mathbf{P})$ with the d-separation oracle in Program 1.

Assume we are given a meta-predicate `random/2` that indicates all random predicates together with their arities.

Example 11. *In Example 8, this means we add the facts*

```
random(opens,2). random(leaks,1). random(smokes,2). random(fire,1).
```

to the program structure $\boldsymbol{P}$.

In this case, Program 2 computes the ground graph $\mathrm{Graph}_{\mathscr{E}}(\mathbf{P})$ when applied to the program structure $\mathbf{P}$ and an external database $\mathscr{E}$.

Program 2 (Representation of the Ground Graph)**.**

```
underlyingAtom(Atom,Atom) :- Atom \= (\+_).
underlyingAtom(Literal,Atom) :- Literal = (\+Literal1),
        underlyingAtom(Literal1,Atom).
% Determine the conditions in a body of a random clause
conditions(Body,true) :- Body \= (_,_), underlyingAtom(Body,Atom),
    functor(Atom,R,Arity), random(R,Arity).
conditions(Body,Body) :- Body \= (_,_), underlyingAtom(Body,Atom),
    functor(Atom,R,Arity), \+random(R,Arity).
conditions((C1,Body), Cond) :- underlyingAtom(C1,Atom),
    functor(Atom,R,Arity), random(R,Arity), conditions(Body,Cond).
conditions((C1,Body),(C1,Cond)) :- underlyingAtom(C1,Atom),
    functor(Atom,R,Arity), \+random(R,Arity), conditions(Body,Cond).
% Calculate the potential parents
potentialParents(Body,[Atom]) :- Body \= (_,_), underlyingAtom(Body,Atom),
    functor(Atom,R,Arity), random(R,Arity).
potentialParents(Body,[]) :- Body \= (_,_), underlyingAtom(Body,Atom),
    functor(Atom,R,Arity), \+random(R,Arity).
potentialParents((C1,Body), [Atom|Parents]) :- underlyingAtom(C1,Atom),
    functor(Atom,R,Arity), random(R,Arity), potentialParents(Body,Parents).
potentialParents((C1,Body), Parents) :- underlyingAtom(C1,Atom),
    functor(Atom,R,Arity), \+random(R,Arity),
    potentialParents(Body,Parents).
%Check whether edge exists
(X ---> Y) :- random(R,Arity), functor(Y,R,Arity), clause((_::Y),Body),
        potentialParents(Body,Parents), member(X,Parents),
        conditions(Body,Conds), Conds.
```

Together, Program 1 and 2 yield a meta-interpreter that computes valid conditional independence statements implied by an acyclic, stratified program structure and an external database. In the appendix we prove the following result that establishes the completeness of this conditional independence oracle for a fragment of program structures.

Theorem 3 (Completeness)**.** *Let* $\boldsymbol{P}$ *be a positive program structure and let* $\mathscr{E}$ *be an external database such that the ground graph* $\mathrm{Graph}_{\mathscr{E}}(\boldsymbol{P})$ *is singly connected. Further, let* π *be a choice of parameters for* $\boldsymbol{P}$ *with values in* $(0,1)$ *that yields proper unconditional probabilities for all ground variables. In this case, the grounding* $\boldsymbol{P}^{(\pi,\mathscr{E})}$ *yields a distribution that is faithful to the ground graph* $\mathrm{Graph}_{\mathscr{E}}(\boldsymbol{P})$.

In particular, if the ground graph $\mathrm{Graph}_{\mathscr{E}}(\boldsymbol{P})$ *is singly connected for every external database* $\mathscr{E}$ *and has a generalized ProbLog clause grounding to a probabilistic fact for every source in* $\mathrm{Graph}_{\mathscr{E}}(\boldsymbol{P})$*, our meta-interpreter is correct and complete for every choice of parameters* π *of* $\boldsymbol{P}$ *with values in* $(0,1)$.

As singly connected ground graphs imply mutual independence of body atoms, the fragment of Theorem 3 reminds us of the (Ind,Ind) assumption [10], under which marginal probabilities can be computed in a much simpler way.

Example 12. *As the program structure* $\boldsymbol{P}$ *of the introduction lies in the fragment of Theorem 3, Program 1 and 2 yield a correct and complete conditional independence oracle whenever we choose probabilities in* $(0,1)$ *for every generalized ProbLog clause in* $\boldsymbol{P}$.

5 Experimental Evaluation

To evaluate our conditional independence oracle in Program 1 and 2, we investigate the lifted program **P** that defines the random predicate $random(p,1)$ with the random part below.

$$random(p,1). \qquad 0.5 :: p(X) \leftarrow n(X). \qquad 0.5 :: p(Y) \leftarrow p(X), n(X), n(Y), e(X,Y).$$

Further, the program **P** expects a directed acyclic graph G as external database, represented by storing its nodes $n/1$ and its edges $e/2$. Here, $p/1$ is a random property that holds with a base probability of 0.5 for every node of G. If we observe $p(n)$ for a node n of G, this enhances the probability of $p(c)$ for every child c of n by a factor of 0.5.

Further, we generate directed acyclic graphs G with $S := 5i$ nodes for $1 \leq i \leq 20$ by sampling five times from the following ProbLog program.

$$n(1). \qquad n(Y) \leftarrow n(X),\ Y = X+1,\ X < S+1. \qquad \frac{1}{\sqrt{S}} :: e(X,Y) \leftarrow n(X),\ n(Y),\ X < Y.$$

In this way we obtain five directed acyclic graphs on the nodes $\{1,\ldots,S\}$ that have an edge $i \rightarrow j$ with probability $\sqrt{S}^{-1}$ for every $1 \leq i < j \leq S$. Next, we choose for every graph size S ten tuples (a,b) of even numbers between one and S. We then process the queries

$$dseparates(p(a), p(b), []) \text{ and } dseparates(p(a), p(b), [p(i): \ i \text{ odd number between 1 and S}])$$

on **P** for every graph of size S using the meta-interpreter of Program 1 and 2 with a timeout of 10 seconds.

Finally, we aim to compare our approach with checking the definition of independence, i.e. with checking whether $\pi(p(a) \wedge p(b)) = \pi(p(a)) \cdot \pi(p(b))$. To this aim we additionally calculate the probabilities of $p(a)$, $p(b)$ and $p(a) \wedge p(b)$ with the evidence $\{p(i): \ i \text{ odd number between 1 and S}\}$ and without evidence using ProbLog 2 with a timeout of 10 seconds. The median and maximal run times of the queries described above are visualized in Figure 1, clearly demonstrating that our approach performs significantly faster than checking (conditional) independencies with exact inference in ProbLog 2.

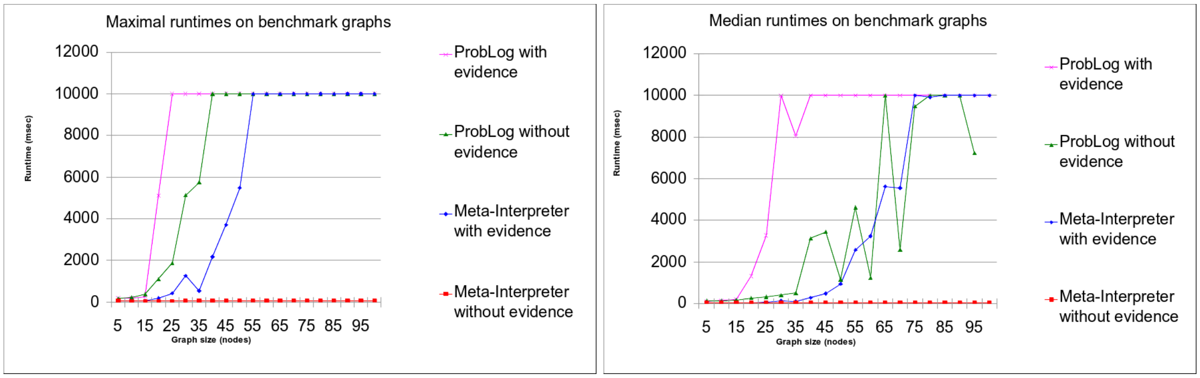


Figure 1:

6 Conclusion

First, we establish a framework for lifted probabilistic logic programming, i.e. we abstract the notion of a ProbLog program away from a concrete database and away from the concrete probabilities mentioned in the clauses. In this way we obtain so-called program structures. As our main result we then use the theory of d-separation from Bayesian networks to reason about conditional independence on the basis of these program structures. We also implement the corresponding independence oracle as a meta-interpreter in Prolog. Finally, we prove the completeness of our conditional independence oracle for a fragment of programs structures in Theorem 3. The paper then closes with an experimental evaluation of our results revealing that our approach processes significantly faster than checking the definition of independence with exact inference in ProbLog 2.

As the theory of d-separation is the basis of causal structure discovery in Bayesian networks, one direction for future work is to develop the analogue of this theory in probabilistic logic programming. In this context, we note that causal faithfulness is needed to extract possible causal structures from an observed distribution. Furthermore, Theorem 2 suggests that our independence oracle is complete for most program structures and choices of parameters. Hence, in our opinion determining completeness results for more general fragments of program structures is a promising direction for future work.

References

[1] Luc De Raedt, Angelika Kimmig & Hannu Toivonen (2007): *ProbLog: A Probabilistic Prolog and Its Application in Link Discovery*. In: *20th International Joint Conference on Artificial Intelligence*, 7, AAAI Press, Hyderabad, India, pp. 2462–2467. Available at `https://www.ijcai.org/Proceedings/07/Papers/396.pdf`.

[2] Dan Geiger & Judea Pearl (1990): *On the Logic of Causal Models*. In: *Uncertainty in Artificial Intelligence*, *Machine Intelligence and Pattern Recognition* 9, North-Holland, pp. 3–14, doi:10.1016/B978-0-444-88650-7.50006-8.

[3] Dan Geiger, Thomas Verma & Judea Pearl (1990): *Identifying Independence in Bayesian networks*. *Networks* 20(5), pp. 507–534, doi:10.1002/net.3230200504.

[4] Steven Holtzen, Guy Van den Broeck & Todd Millstein (2020): *Scaling Exact Inference for Discrete Probabilistic Programs*. *Proc. ACM Program. Lang.*, doi:10.1145/3428208.

[5] Michael McDermott (1995): *Redundant Causation*. *The British Journal for the Philosophy of Science* 46(4), pp. 523–544, doi:10.1093/bjps/46.4.523.

[6] Christopher Meek (1995): *Strong Completeness and Faithfulness in Bayesian networks*. In: *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, UAI'95, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 411–418, doi:10.5555/2074158.2074205.

[7] Judea Pearl (1988): *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, doi:10.1016/C2009-0-27609-4.

[8] Judea Pearl (2000): *Causality*, 2 edition. Cambridge University Press, Cambridge, UK, doi:10.1017/CBO9780511803161.

[9] David Poole (1997): *The independent choice logic for modelling multiple agents under uncertainty*. *Artificial Intelligence* 94(1), pp. 7–56, doi:10.1016/S0004-3702(97)00027-1.

[10] Fabrizio Riguzzi (2014): *Speeding Up Inference for Probabilistic Logic Programs*. *The Computer Journal* 57(3), pp. 347–363, doi:10.1093/comjnl/bxt096.

[11] Kilian Rückschloß & Felix Weitkämper (2021): *Reasoning about Independence in Open Universe Probabilistic Logic Programs*. In: *Proceedings of the International Conference on Logic Programming 2021 Workshops*, *CEUR Workshop Proceedings* 2970, CEUR-WS.org. Available at `https://ceur-ws.org/Vol-2970/plppaper3.pdf`.

[12] Kilian Rückschloß & Felix Weitkämper (2022): *Exploiting the Full Power of Pearl's Causality in Probabilistic Logic Programming*. In: *Proceedings of the International Conference on Logic Programming 2022*

Workshops, *CEUR Workshop Proceedings* 3193, CEUR-WS.org. Available at `http://ceur-ws.org/Vol-3193/paper1PLP.pdf`.

[13] Taisuke Sato (1995): *A Statistical Learning Method for Logic Programs with Distribution Semantics*. In: *Logic Programming: The 12th International Conference*, The MIT Press, pp. 715–729, doi:10.7551/mitpress/4298.003.0069.

[14] Taisuke Sato & Yoshitaka Kameya (1997): *PRISM: A Language for Symbolic-Statistical Modeling*. In: *Proceedings of the Fifteenth International Joint Conference on Artifical Intelligence - Volume 2*, IJCAI'97, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 1330–1335. Available at `https://dl.acm.org/doi/10.5555/1622270.1622348`.

[15] Joost Vennekens, Sofie Verbaeten & Maurice Bruynooghe (2004): *Logic Programs with Annotated Disjunction*. In: *Logic Programming*, Springer Berlin Heidelberg, pp. 431–445, doi:10.1007/978-3-540-27775-0_30.

[16] Thomas Verma & Judea Pearl (1990): *Causal Networks: Semantics and Expressiveness*. In: *Uncertainty in Artificial Intelligence*, *Machine Intelligence and Pattern Recognition* 9, North-Holland, pp. 69–76, doi:10.1016/B978-0-444-88650-7.50011-1.

Appendix

Lemma 1 (Weak Transitivity, [7, p.137, Exercise 3.10])**.** *Let $\mathbf{A}$ and $\mathbf{C}$ be sets of random variables. Further, let B be a Boolean random variable and let π be a distribution on $\mathbf{A} \cup \mathbf{C} \cup \{B\}$. Choose possible values $\mathbf{a}$ and $\mathbf{c}$ for the random variables in $\mathbf{A}$ and $\mathbf{C}$ respectively. Further, choose a possible value b for B. If b is both dependent on $\mathbf{a}$ and on $\mathbf{c}$, we find that $\mathbf{a}$ and $\mathbf{c}$ are marginally dependent or they become dependent once we condition on b.* □

Lemma 2. *Let π be a distribution on the set Ω of all valuations on the variables $X, X_1, \ldots, X_n$ such that $\{X\} \cup \{X_i\}_{i=1,\ldots,n}$ encodes a mutually independent set of events and let φ be a propositional formula (in minimal DNF) in which X occurs, and only occurs positively.*

Then φ and X are positively correlated as events on Ω.

Proof. We must show that $\pi(\varphi \mid X) > \pi(\varphi)$. Let ψ be the formula obtained by substituting $\top$ for X in φ. Then we have the following:

$$\pi(\varphi \mid X) = \pi(\psi \mid X) = \pi(\psi).$$

The first equality holds because φ and ψ are equivalent on valuations satisfying X, and the second equality holds because π is an independent distribution. As ψ is staisfied by all valuations satisfying φ, but also on at least one valuation not satisfying φ (since X occurs in the minimal DNF formula φ), this implies the statement. □

Proof of Theorem 3. Let $G := \mathrm{Graph}_{\mathscr{E}}(\mathbf{P})$ be the ground graph of the program structure $\mathbf{P}$ with respect to the external database $\mathscr{E}$. Let P be a d-connecting path from A to B over a subset of nodes $\mathbf{Z}$ in G. We show that for all truly probabilistic choices of parameters, A and B are dependent over $\mathbf{Z}$. Let π be the probability distribution on the random variables corresponding to such a choice of parameters. We proceed by induction on the length of P.

As we use this in the argument for colliders below, we first show that for directed paths, one obtains the stronger statement that A and B are positively correlated over $\mathbf{Z}$.

We begin with paths of length two. If the length of P is two, there is an edge between A and B. Assume without loss of generality that the edge goes from A to B. Consider the definition φ of B in the Clark completion of the logic program associated with $\mathbf{P}$. This is a propositional formula in the parents of B and the error terms of ground clauses with head B; Since A is a parent of B, there is a clause $\mathscr{C}$

whose body includes A, and thus since B is true if and only if A is true when the error terms associated with all other clauses are set to false and all other facts in the body of $\mathscr{C}$ are true, A occurs in a minimal disjunctive normal form representation of φ. As G is singly connected, the only path between any two parents of B is via $B \notin \mathbf{Z}$, and thus the distribution induced by $\pi \mid Z$ on the parents of B is independent by the correctness of d-separation. Therefore A and B are positively correlated by Lemma 2.

So assume it to be true for directed paths of length n and let $A_1 \dots A_n A_{n+1}$ be a directed path of length $n+1$. Then $\pi(A_{n+1} \mid A_1, Z)$ is equal to

$$\pi(A_{n+1} \mid A_n, A_1, Z)\pi(A_n \mid A_1, Z) + \pi(A_{n+1} \mid \neg A_n, A_1, Z)\pi(\neg A_n \mid A_1, Z)$$

and $\pi(A_{n+1} \mid Z)$ is equal to

$$\pi(A_{n+1} \mid A_n, Z)\pi(A_n \mid Z) + \pi(A_{n+1} \mid \neg A_n, Z)\pi(\neg A_n \mid Z).$$

Since P is the only path between A_1 and A_{n+1}, A_1 and A_{n+1} are d-separated by A_n. Thus, we find $\pi(A_{n+1} \mid A_n, A_1, Z) = \pi(A_{n+1} \mid A_n, Z)$ and $\pi(A_{n+1} \mid \neg A_n, A_1, Z) = \pi(A_{n+1} \mid \neg A_n, Z)$. Furthermore, by the induction hypothesis, $\pi(A_{n+1} \mid A_n, Z) > \pi(A_{n+1} \mid \neg A_n, Z)$ and $\pi(A_n \mid A_1, Z) > \pi(A_n \mid Z)$. Overall, this implies that $\pi(A_{n+1} \mid A_1, Z) > \pi(A_{n+1} \mid Z)$ as required. (□)

We now return to the case of a general d-connecting path. As paths of length 2 are necessarily directed, the base step of the induction follows from the directed case above. So assume that dependence holds for all paths of length n and let P be a path of length $n+1 > 2$.

We proceed by appeal to weak transitivity.

Let A', C and B be the final three nodes in P. We first cover the case where C is not a collider.

Then by the induction hypothesis, A and C are dependent over $\mathbf{Z}$, and by the length two case above, C and B are dependent over $\mathbf{Z}$. Thus, either A and B are dependent over $\mathbf{Z}$, or they are dependent over $\{C\} \cup \mathbf{Z}$. However, since G is singly connected, P is the only path from A to B and thus A and B are d-separated by $\{C\} \cup \mathbf{Z}$. Together with the correctness of d-separation we can conclude that A and B are dependent over $\mathbf{Z}$.

Now consider the case of a collider where C has a descendant in $\mathbf{Z}$. Note that since conditioning on a descendant C_i of C blocks the (only) path between A and B and any descendant C' of C_i, we can assume by correctness of d-separation without loss of generality that all descendants C_i of C in $\mathbf{Z}$ are non-descendants of each other. Let $\{C_i\}_{i=1,\dots,n}$ be the descendants of C in $\mathbf{Z}$, and let $\mathbf{Z}' := \mathbf{Z} \setminus \{C_i\}_{i=1,\dots,n}$.

We show by induction on n that $\pi(X \mid C_1, \dots, C_n, \mathbf{Z}') > \pi(X, \mathbf{Z}')$ for $X \in \{A', B\}$. The base case of the induction is given by the special case of a directed path above. So assume $\pi(X \mid C_1, \dots, C_n, \mathbf{Z}') > \pi(X \mid \mathbf{Z}')$. We want to show that $\pi(X \mid C_1, \dots, C_n, C_{n+1}, \mathbf{Z}') > \pi(X \mid \mathbf{Z}')$. However, we know that

$$\pi(X \mid C_1, \dots, C_n, C_{n+1}, \mathbf{Z}') > \pi(X \mid C_1, \dots, C_n, \mathbf{Z}')$$

by the special case of directed paths above, from which the claim follows. (□)

To apply weak transitivity, we alter the graph by introducing a new node $\bigwedge C$ with arrows from every C_i into $\bigwedge C$. We extend the probability distribution to $\bigwedge C$ by setting $\bigwedge C$ to be true of and only if all C_i are true. By the argument above, A' and B are both positively correlated with $\bigwedge C$ over $\mathbf{Z}'$. Thus, by weak transitivity, either A' and B are dependent over $\mathbf{Z}'$ or they are dependent over $\{\bigwedge C\} \cup \mathbf{Z}'$. The former is excluded by the correctness of d-separation and the fact that the collider C blocks the only path from A' to B (since the original graph was singly connected). Therefore A' and B are dependent over $\{\bigwedge C\} \cup \mathbf{Z}'$, which implies A' and B being dependent over $\mathbf{Z} = C_1, \dots, C_n, \mathbf{Z}'$ in the original graph.

We can now conclude the proof by using weak transitivity and singly-connectedness a final time to deduce that A and B are dependent over Z from the fact that A and A' are dependent over Z (the induction hypothesis) and the fact that A' and B are dependent over Z. □

Understanding ProbLog as Probabilistic Argumentation

Francesca Toni Nico Potyka
Department of Computing
Imperial College London, UK
{ft, n.potyka}@ic.ac.uk

Markus Ulbricht
Department of Computer Science
Leipzig University, Germany
mulbricht@informatik.uni-leipzig.de

Pietro Totis
Department of Computer Science
KU Leuven, Belgium
pietro.totis@kuleuven.be

ProbLog is a popular probabilistic logic programming language/tool, widely used for applications requiring to deal with inherent uncertainties in structured domains. In this paper we study connections between ProbLog and a variant of another well-known formalism combining symbolic reasoning and reasoning under uncertainty, i.e. probabilistic argumentation. Specifically, we show that ProbLog is an instance of a form of Probabilistic Abstract Argumentation (PAA) that builds upon Assumption-Based Argumentation (ABA). The connections pave the way towards equipping ProbLog with alternative semantics, inherited from PAA/PABA, as well as obtaining novel argumentation semantics for PAA/PABA, leveraging on prior connections between ProbLog and argumentation. Further, the connections pave the way towards novel forms of argumentative explanations for ProbLog's outputs.

1 Introduction

ProbLog [10, 20] is a popular probabilistic logic programming formalism, equipped with efficient tools in support of applications[1] In a nutshell, ProbLog programs amount to logic programs where facts may be equipped with probabilities. Thus, ProbLog can be naturally used for applications requiring to deal with inherent uncertainties, e.g. in neuro-symbolic settings to learn how to operate on images [18], object tracking [1], modelling metabolic networks[16], or synthesising inductive data models [7].

As a knowledge representation and reasoning (KRR) formalism, ProbLog can be seen as bringing together symbolic reasoning and reasoning under uncertainty. Thus, it makes sense to wonder how it relates to the family of KRR formalisms broadly referred to as *Probabilistic Argumentation* (PA) [14], which also bring together these two types of reasoning. Indeed, some existing works already study the connections between the two, in particular in the presence of inherent uncertainties, e.g. in neuro-symbolic settings with images [18]. Also, SMProbLog [23] extends ProbLog beyond the standard well-founded model semantics [11] towards the stable model semantics [12] so as to capture a form of PA. Further, [19] shows how the form of PA in [17] can be implemented in ProbLog. These existing works thus focus on showing how (variants of) Problog can obtain (forms of) PA. In this paper, instead, we focus on the opposite direction, specifically on whether (some forms of) PA can be used to obtain (variants of) ProbLog. In summary, our contribution is two-fold:

- We define a new instance of the well-known Assumption-based Argumentation (ABA) [3, 4] and use it to instantiate Probabilistic Abstract Argumentation (PAA) [9] (Section 3);
- We formally relate ProbLog and the proposed instance of PAA (Section 4).

This reinterpretation of ProbLog in argumentation terms opens new avenues (as discussed in Section 5).

---

[1] See `https://dtai.cs.kuleuven.be/problog/`.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 183–189, doi:10.4204/EPTCS.385.18

2 Background

Logic Programs. A *logic program* (LP) is a set of (implicitly universally quantified) rules of the form $l_0 \leftarrow l_1, \ldots, l_m$, with $m \geq 0$, l_0 an atom (called the *head* of the rule) and each l_i (for $1 \leq i \leq m$) an atom or the *negation-as-failure* $not\, a_i$ of an atom a_i. If $m = 0$ then the rule is called a *fact*. The *Herbrand base* of an LP is the set of all ground atoms obtained from the predicate, function/constant symbols in the LP.

ProbLog. The material in this section is adapted from [10, 20]. A *ProbLog program* is a set R of rules as in an LP together with a set F of *probabilistic facts* of the form $p :: l_0 \leftarrow$, where $p \in [0,1]$ and l_0 is an atom that is not the head of any rule in R. The semantics of a ProbLog program is based on distribution semantics [21]: a ProbLog program $T = F \cup R$ defines a probability distribution over LPs $L = F' \cup R, F' \subseteq \{l \leftarrow | p :: l \leftarrow \in F\}$:

$$P(L|T) = \prod_{l \leftarrow \in F', p::l \leftarrow \in F} p \cdot \prod_{l \leftarrow \notin F', p::l \leftarrow \in F} (1-p)$$

Then, for a *query* q, the *success probability* for q, given T, is^2

$$P_s(q|T) = \sum_{L = F' \cup R, F' \subseteq \{l \leftarrow | p::l \leftarrow \in F\}, \exists \theta : L \models q\theta} P(L|T)$$

for $\models$ the chosen logic programming semantics: if L is a *positive LP* (without negation as failure), as in [21], then this is the least Herbrand model; otherwise, let it be the well-founded model [11].

Example 1. *Consider the (propositional) LP amounting to the following rules:*

$$a \leftarrow b, not\, c; \qquad b \leftarrow; \qquad d \leftarrow not\, d$$

(referred to in short as ρ_1, ρ_2, ρ_3, respectively). Then, for $R = \{\rho_1, \rho_3\}$ and $F = \{0.3 :: b \leftarrow\}$, $T = F \cup R$ is a ProbLog program. Given that $R \not\models a$ (as the well-founded model of R deems a, b, c false and d undecided) and $R \cup \{\rho_2\} \models a$ (as the well-founded model of $R \cup \{\rho_2\}$ deems a, b true, c false and d undecided), $P_s(a|T) = P(R \cup \{\rho_2\}|T) = 0.3$.

ProbLog typically assumes that the well-founded model of every LP $L \subseteq F' \cup R, F' \subseteq \{l \leftarrow | p :: l \leftarrow \in F\}$ is two-valued [10, 23], but the well-founded model may be three-valued [13], as in the example .

Abstract Argumentation. An Abstract Argumentation (AA) framework [8] is a pair $(Args, Att)$ with $Args$ the *arguments* and $Att \subseteq Args \times Args$ the *attack* relation. AA frameworks can be equipped with various argumentative semantics [8], e.g. the *grounded* or *stable extensions* semantics [8], identifying sets of "acceptable arguments". For illustration, given $AAF = (Args, Att)$ with $Args = \{\alpha, \beta\}$ and $Att = \{(\beta, \beta)\}$, the grounded extension of AAF is $\{\alpha\}$ and there is no stable extension of AAF.

Assumption-based argumentation (ABA). An *ABA framework* (as originally proposed in [3], but presented here following more recent accounts by [22, 4]) is a tuple $\langle \mathcal{L}, \mathcal{R}, \mathcal{A}, \overline{} \rangle$ where

- $\langle \mathcal{L}, \mathcal{R} \rangle$ is a deductive system, where $\mathcal{L}$ is a *language* and $\mathcal{R}$ is a set of *(inference) rules* of the form $s_0 \leftarrow s_1, \ldots, s_m$ ($m \geq 0, s_i \in \mathcal{L}$, for $1 \leq i \leq m$);
- $\mathcal{A} \subseteq \mathcal{L}$ is a (non-empty) set of *assumptions*;
- $\overline{}$ is a total mapping from $\mathcal{A}$ into $\mathcal{L}$, where $\overline{a}$ is the *contrary* of a, for $a \in \mathcal{A}$.

---

[2] A query q is a possibly non-ground atom from the Herbrand base of T and θ denotes a possibly empty variable substitution.

Given a rule $s_0 \leftarrow s_1, \ldots, s_m$, s_0 is referred to as the *head* and $s_1, \ldots, s_m$ as the *body*; if $m = 0$ then the body is said to be *empty*. *Flat ABA frameworks* are restricted so that assumptions are not heads of rules.

Differently from AA, where arguments and attacks are given, in flat ABA they are derived from the building blocks of ABA frameworks. Specifically, *arguments* are deductions of claims using rules and supported by assumptions, and *attacks* are directed at the assumptions in the support of arguments:

- *an argument for (the claim) $s \in \mathscr{L}$ supported by $A \subseteq \mathscr{A}$ and $\mathscr{S} \subseteq \mathscr{R}$* (denoted $A \vdash_{\mathscr{S}} s$) is a finite tree with nodes labelled by sentences in $\mathscr{L}$ or by $true$[3], the root labelled by s, leaves either $true$ or assumptions in A, and non-leaves s' with, as children, the elements of the body of one rule in $\mathscr{S}$ with head s' ($true$ in the case of a rule with an empty body), and $\mathscr{S}$ the set of all these rules;
- an argument $A_1 \vdash_{\mathscr{S}_1} s_1$ *attacks* an argument $A_2 \vdash_{\mathscr{S}_2} s_2$ iff $s_1 = \overline{a}$ for some $a \in A_2$.

Given an ABA framework $\langle \mathscr{L}, \mathscr{R}, \mathscr{A}, \overline{} \rangle$, let $Args$ be the set of all arguments and Att be defined as above. Then $(Args, Att)$ is an AA framework (that we refer to as *the AA framework for* $\langle \mathscr{L}, \mathscr{R}, \mathscr{A}, \overline{} \rangle$) and standard argumentative semantics for the latter can be used to determine, e.g., grounded extensions.[4]

LPs as Flat ABA Frameworks. Flat ABA can be instantiated to capture logic programming under several semantics [3], including the well-founded model [11] and the stable model semantics [12].

Example 2. *The semantics of the LP* $\{\rho_1, \rho_2, \rho_3\}$ *from Example 1 can be captured by understanding it as a flat ABA framework* $\langle \mathscr{L}, \mathscr{R}, \mathscr{A}, \overline{} \rangle$ *with*

- $\mathscr{R} = \{\rho_1, \rho_2, \rho_3\}$*;*
- $\mathscr{L} = \{a, b, c, d, not\, a, not\, b, not\, c, not\, d\}$ *(this is the Herbrand base of the LP together with all the negation as failure literals that can be constructed from its Herbrand base);*
- $\mathscr{A} = \{not\, a, not\, b, not\, c, not\, d\}$ *(this is the set of all negation as failure literals in* $\mathscr{L}$*);*
- $\overline{not\, X} = X$ *for all* $X \in \{a, b, c, d\}$.

$(Args, Att)$ *can then be obtained from this ABA framework, with:*

- $Args = \{\{not\, c\} \vdash_{\{\rho_1, \rho_2\}} a,\ \{\} \vdash_{\{\rho_2\}} b,\ \{not\, d\} \vdash_{\{\rho_3\}} d\} \cup \{\{not\, X\} \vdash_{\{\}} not\, X | X \in \{a, b, c, d\}\}$
- $Att = \{(\{not\, c\} \vdash_{\{\rho_1, \rho_2\}} a, \{not\, a\} \vdash_{\{\}} not\, a),\quad (\{not\, d\} \vdash_{\{\rho_3\}} d, \{not\, d\} \vdash_{\{\}} not\, d),$
  $(\{not\, d\} \vdash_{\{\rho_3\}} d, \{not\, d\} \vdash_{\{\rho_3\}} d),\quad (\{\} \vdash_{\{\rho_2\}} b, \{not\, b\} \vdash_{\{\}} not\, b)\}.$

$\mathscr{R}$*, as a LP, admits no stable model [12], and there is no stable extension [8] of this ABA framework. The well-founded model [11] of* $\mathscr{R}$ *deems* a, b *true,* c *false and* d *undecided, and the grounded extension [8] of* $\langle \mathscr{L}, \mathscr{R}, \mathscr{A}, \overline{} \rangle$ *accepts*

$$\{\{not\, c\} \vdash_{\{\rho_1, \rho_2\}} a,\ \{\} \vdash_{\{\rho_2\}} b,\ \{not\, c\} \vdash_{\{\}} not\, c\}.$$

The two correspond in the sense that the atomic claims of accepted arguments in the grounded extension are true in the well-founded model, the negation as failure claims of accepted arguments are the complement of false atoms in the well-founded model, and no other atoms are true or false therein (and vice versa, i.e. every true atom and the negation as failure of every false atom in the well-founded model are claims of accepted arguments in the grounded extension, and no other arguments exist therein). Our results will leverage on the correspondence illustrated here (see [3] for formal results, including correspondences between other semantics of LPs and notions of extensions in flat ABA [3]).

---

[3] Here $true$ stands for the empty body of rules.

[4] ABA semantics were originally defined in terms of sets of assumptions and attacks between them [3], but can be reformulated, for flat ABA frameworks, in terms of sets of arguments and attacks between them (see [22]), as given here.

Probabilistic Abstract Argumentation (PAA). The material in this section is adapted from [9] (focusing on a single juror). A *PAA framework* basically consists of an AA framework and a probability distribution assigning probabilities to subframeworks. Formally, it is a triple $(AAF, PS, \Vdash)$ where:

- $AAF = (Args, Att)$ is an AA framework;
- $PS = (\mathscr{W}, P)$ is a probability space, with $\mathscr{W}$ the *world set* and P a *probability distribution* over $\mathscr{W}$;
- $\Vdash \subseteq \mathscr{W} \times Args$ specifies which arguments are *applicable* in worlds in PS, where $\alpha \in Args$ is applicable in $w \in \mathscr{W}$ iff $(w, \alpha) \in \Vdash$ (we also write $w \Vdash \alpha$ for $(w, \alpha) \in \Vdash$).

Given a possible world $w \in \mathscr{W}$, the *AAF wrt w*, denoted by $AAF_w = (Args_w, Att_w)$, is defined by restricting AAF to the arguments applicable in w. More formally, $Args_w = \{\alpha \in Args \mid w \Vdash \alpha\}$ and $Att_w = Att \cap (Args_w \times Args_w)$. We denote the grounded extension of AAF_w by $G(AAF_w)$. Then, the *grounded probability of argument* $\alpha \in Args$ is defined as follows:

$$Prob_G(\alpha) = \sum_{w \in \mathscr{W} : \alpha \in G(AAF_w)} P(w).$$

Intuitively, $Prob_G(\alpha)$ is the probability that α is accepted in a possible world (under the grounded extension semantics). PAA is a form of probabilistic argumentation under the constellation approach [14].

3 ProbLog-ABA and ProbLog-PAA

Several instances of (flat and non-flat) ABA have been studied [3, 5] (including the flat logic programming instance illustrated in Example 2 [3]). Here, we introduce the following (flat) instance, referred to as *ProbLog-ABA* since, as we will show, it is a first step towards capturing ProbLog programs as PAA.

Definition 1. *A* ProbLog-ABA framework *is a flat ABA framework* $\langle \mathscr{L}, \mathscr{R}, \mathscr{A}, \overline{} \rangle$ *where*

- $\mathscr{R}$ *is a (ground)*[5] *LP;*
- $\mathscr{L}$ *is a set of atoms and negation as failure literals; formally, for HB the Herbrand Base of* $\mathscr{R}$ *and* $HB^{not} = \{not\ p | p \in HB\}$, $\mathscr{L} = HB \cup HB^{not} \cup \{\chi\}$, *for a special symbol* $\chi \notin HB \cup HB^{not}$*;*
- $\mathscr{A} = HB^{not} \cup \mathscr{F}$, *for some* $\mathscr{F} \subseteq HB$*;*
- *for any* $not\ p \in \mathscr{A}$, $\overline{not\ p} = p$*; for any* $\phi \in \mathscr{F}$, $\overline{\phi} = \chi$.

Basically, ProbLog-ABA frameworks generalise the ABA frameworks capturing LPs by including additional assumptions ($\mathscr{F}$), all having the same contrary (χ) not occurring in the LP ($\mathscr{R}$). These additional assumptions can be seen as playing the role of abducibles in abductive logic programming [15]. They amount to the atoms in probabilistic facts in ProbLog programs $T = R \cup F$, as follows.[6]

Definition 2. *The* ProbLog-ABA framework corresponding to T *is* $\langle \mathscr{L}, \mathscr{R}, \mathscr{A}, \overline{} \rangle$ *as in Definition 1 where* $\mathscr{R} = R$ *and* $\mathscr{F} = \{l | (p :: l \leftarrow) \in F\}$.

We can then instantiate PAA with ProbLog programs, so that the AA framework component is drawn from the ProbLog-ABA framework corresponding to a given ProbLog program and the probability space is defined to mirror the probability distribution over LPs captured by the ProbLog program, as follows.

---

[5] This is in line with standard practices when studying the semantics of LPs: they are assumed to stand for the set of all their ground instances over their Herbrand universe.

[6] Given that we are focusing on semantics, in the remainder we assume that the ProbLog program is ground.

Definition 3. *Let* $\langle \mathscr{L}^T, \mathscr{R}^T, \mathscr{A}^T, \overline{}^T \rangle$ *be the ProbLog-ABA framework corresponding to* T*, for* $\mathscr{A}^T = HB^{not} \cup \mathscr{F}$*, and let* $AAF^T = (Args^T, Att^T)$ *be the AA framework for* $\langle \mathscr{L}^T, \mathscr{R}^T, \mathscr{A}^T, \overline{}^T \rangle$*. Then, the* PAA framework corresponding to T *is* $(AAF^T, PS^T, \Vdash^T)$ *where* $PS^T = (\mathscr{W}^T, P)$ *is the probability space with*

- $\mathscr{W}^T = 2^{\mathscr{F}}$,
- *for* $w \in \mathscr{W}^T$*:* $P(w) = \prod_{l \in w, p::l \leftarrow \in F} p \cdot \prod_{l \in \mathscr{F} \setminus w, p::l \leftarrow \in F} (1-p)$,

and $\Vdash^T$ *is such that, for* $w \in \mathscr{W}^T$*,* $\alpha \in Args^T$*,* $\alpha = A \vdash_{\mathscr{S}} s$*:* $\quad w \Vdash^T \alpha$ *iff* $A \cap \mathscr{F} \subseteq w$*.*

In line with [9], we assume that the AA frameworks in PAA frameworks corresponding to ProbLog programs are finite. Given such a PAA framework, we can measure the grounded probability of arguments (as in standard PAA), as well as the *grounded probability of queries*, as follows.

Definition 4. *Let* T *be a (ground) ProbLog program and* $(AAF^T, PS^T, \Vdash^T)$ *be the PAA framework corresponding to* T*, for* $AAF^T = (Args^T, Att^T)$ *and* $PS^T = (\mathscr{W}^T, P)$*. Then, the* grounded probability of (ground) query q *is*

$$Prob_G(q) = \sum_{w \in \mathscr{W}^T : \alpha \in G(AAF_w^T), \alpha = A \vdash_{\mathscr{S}} q} P(w).$$

This notion is more fine-grained than that of grounded probability of arguments, reflecting the structured nature of our PAA framework. We conjecture (and leave to future work) that this notion is an instance of that of grounded probability of sentences in the Probabilistic ABA of [9].

4 Results

We show that the semantics of ProbLog can be captured in probabilistic argumentation, as follows.

Proposition 1. *Let* T *be a (ground) ProbLog program and* $(AAF^T, PS^T, \Vdash^T)$ *be the PAA framework corresponding to* T*, for* $AAF^T = (Args^T, Att^T)$ *and* $PS^T = (\mathscr{W}^T, P)$*. Then, for any (ground) query* q*:*

$$P_s(q|T) = Prob_G(q); \tag{1}$$

$$P_s(q|T) \leq \sum_{\alpha \in Args^T : \alpha = A \vdash_{\mathscr{S}} q} Prob_G(\alpha). \tag{2}$$

Proof. (Sketch) $P_s(q|T) = \sum_{L = F' \cup R, F' \subseteq \{l \leftarrow | p::l \leftarrow \in F\}, L \models q} P(L|T)$ by definition. Given the correspondence between well-founded model of a LP and grounded extension of the AA framework for the ABA framework corresponding to the LP [3], and by definition of the probability space in the PAA framework corresponding to T, we obtain $P_s(q|T) = \sum_{w \in \mathscr{W}^T : \alpha \in G(AAF_w^T), \alpha = A \vdash_{\mathscr{S}} q} P(w)$, proving (1). (1) implies (2) because $\sum_{\alpha \in Args^T : \alpha = A \vdash_{\mathscr{S}} q} Prob_G(\alpha) = \sum_{\alpha \in Args^T : \alpha = A \vdash_{\mathscr{S}} q} \sum_{w \in \mathscr{W} : \alpha \in G(AAF_w)} P(w) = \sum_{w \in \mathscr{W}} P(w) \cdot |\{\alpha \in Args^T \mid \alpha = A \vdash_{\mathscr{S}} q\}| \geq \sum_{w \in \mathscr{W}^T : \alpha \in G(AAF_w^T), \alpha = A \vdash_{\mathscr{S}} q} P(w)$ (there may be multiple arguments for the same claim in the grounded extensions obtained by different choices of worlds). □

Thus, the notion of grounded probability of queries in the ProbLog-ABA instance of PAA corresponds exactly to the notion of success probability in ProbLog, whereas grounded probability of arguments in PAA approximates it.

5 Conclusion and Discussion

We have defined a form of PAA, instantiated with a novel form of ABA and probability spaces mirroring ProbLog's probability distributions over logic programs, to re-interpret ProbLog in argumentative terms.

This allows us to broaden the semantics of ProbLog beyond (two-valued) well-founded models, leveraging on other semantics for ABA, notably semantics of *sceptically preferred* and *ideal extensions* (see [4]). Moreover, it opens the way to different forms of explainability for the outputs of ProbLog (under the standard or new semantics). Indeed, argumentative abstractions of various reasoning problems have been shown to lend themselves to diverse explanatory formats, including interactive ones [6], and it is well known that different forms of explanations are needed to deal with different cognitive needs [2].

Future work includes exploring whether reasoning under the new semantics can be efficiently implemented in practice, and whether the new forms of explanations can be beneficially deployed in applications to increase user trust. Also, we have focused on success probability of queries: it would also be interesting to study whether and how explanation probability of queries can be captured argumentatively. It would also be interesting to explore whether ProbLog can be captured directly in Probabilistic ABA of the more general form proposed in [9] (again focusing on a single juror). It would also be interesting to explore whether credulous versions of ProbLog, notably SMProbLog [23], could inform the definition of novel credulous semantics in probabilistic argumentation. Finally, it would be interesting to study whether other languages/tools based on the distribution semantics [21] could also be captured in PAA.

Acknowledgements

Toni and Potyka were supported by the ERC under the EU's Horizon 2020 research and innovation programme (grant No. 101020934). Toni was also supported by J.P. Morgan and the UK RAEng under the Research Chairs and Senior Research Fellowships scheme. Ulbricht was supported by the German Federal Ministry of Education and Research (01/S18026A-F) by funding Big Data and AI "ScaDS.AI" Dresden/Leipzig. Totis was supported by the FWO project N. G066818N and the Flanders AI program.

References

[1] Laura Antanas, Plinio Moreno, Marion Neumann, Rui Pimentel de Figueiredo, Kristian Kersting, José Santos-Victor & Luc De Raedt (2019): *Semantic and geometric reasoning for robotic grasping: a probabilistic logic approach*. *Auton. Robots* 43(6), pp. 1393–1418, doi:`10.1007/s10514-018-9784-8`.

[2] Vijay Arya, Rachel K. E. Bellamy, Pin-Yu Chen, Amit Dhurandhar, Michael Hind, Samuel C. Hoffman, Stephanie Houde, Q. Vera Liao, Ronny Luss, Aleksandra Mojsilovic, Sami Mourad, Pablo Pedemonte, Ramya Raghavendra, John T. Richards, Prasanna Sattigeri, Karthikeyan Shanmugam, Moninder Singh, Kush R. Varshney, Dennis Wei & Yunfeng Zhang (2020): *AI Explainability 360: An Extensible Toolkit for Understanding Data and Machine Learning Models*. *J. Mach. Learn. Res.* 21, pp. 130:1–130:6. Available at `http://jmlr.org/papers/v21/19-1035.html`.

[3] Andrei Bondarenko, Phan Minh Dung, Robert A. Kowalski & Francesca Toni (1997): *An Abstract, Argumentation-Theoretic Approach to Default Reasoning*. *Artif. Intell.* 93, pp. 63–101, doi:`10.1016/S0004-3702(97)00015-5`.

[4] Kristijonas Cyras, Xiuyi Fan, Claudia Schulz & Francesca Toni (2017): *Assumption-based Argumentation: Disputes, Explanations, Preferences*. *FLAP* 4(8). Available at `http://www.collegepublications.co.uk/downloads/ifcolog00017.pdf`.

[5] Kristijonas Cyras, Quentin Heinrich & Francesca Toni (2021): *Computational complexity of flat and generic Assumption-Based Argumentation, with and without probabilities*. *Artif. Intell.* 293, doi:`10.1016/j.artint.2020.103449`.

[6] Kristijonas Cyras, Antonio Rago, Emanuele Albini, Pietro Baroni & Francesca Toni (2021): *Argumentative XAI: A Survey*. In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021*, pp. 4392–4399, doi:`10.24963/ijcai.2021/600`.

[7] Yann Dauxais, Clément Gautrais, Anton Dries, Arcchit Jain, Samuel Kolb, Mohit Kumar, Stefano Teso, Elia Van Wolputte, Gust Verbruggen & Luc De Raedt (2019): *SynthLog: A Language for Synthesising Inductive Data Models (Extended Abstract).* In: *Machine Learning and Knowledge Discovery in Databases - International Workshops of ECML PKDD 2019, Part I*, doi:`10.1007/978-3-030-43823-4_9`.

[8] Phan Minh Dung (1995): *On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and n-Person Games.* *Artif. Intell.* 77(2), pp. 321–358, doi:`10.1016/0004-3702(94)00041-X`.

[9] Phan Minh Dung & Phan Minh Thang (2010): *Towards (Probabilistic) Argumentation for Jury-based Dispute Resolution.* In: *Computational Models of Argument: Proceedings of COMMA 2010*, *Frontiers in Artificial Intelligence and Applications* 216, IOS Press, pp. 171–182, doi:`10.3233/978-1-60750-619-5-171`.

[10] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Sht. Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens & Luc De Raedt (2015): *Inference and learning in probabilistic logic programs using weighted Boolean formulas.* *Theory Pract. Log. Program.* 15(3), pp. 358–401, doi:`10.1017/S1471068414000076`.

[11] Allen Van Gelder, Kenneth A. Ross & John S. Schlipf (1991): *The Well-Founded Semantics for General Logic Programs.* *J. ACM* 38(3), pp. 620–650, doi:`10.1145/116825.116838`.

[12] Michael Gelfond & Vladimir Lifschitz (1988): *The Stable Model Semantics for Logic Programming.* In: *Logic Programming, Proceedings of the Fifth International Conference and Symposium*,, pp. 1070–1080.

[13] Spyros Hadjichristodoulou & David Scott Warren (2012): *Probabilistic Logic Programming with Well-Founded Negation.* In D. Michael Miller & Vincent C. Gaudet, editors: *42nd IEEE International Symposium on Multiple-Valued Logic, ISMVL 2012, Victoria, BC, Canada, May 14-16, 2012*, IEEE Computer Society, pp. 232–237, doi:`10.1109/ISMVL.2012.26`.

[14] Anthony Hunter, Sylwia Polberg, Nico Potyka, Tjitze Rienstra & Matthias Thimm (2021): *Probabilistic argumentation: A survey.* *Handbook of Formal Argumentation* 2, pp. 397–441.

[15] Antonis C. Kakas, Robert A. Kowalski & Francesca Toni (1998): *The role of abduction in logic programming.* In: *Handbook of logic in artificial intelligence and logic programming*, pp. 35–324.

[16] Angelika Kimmig & Fabrizio Costa (2012): *Link and Node Prediction in Metabolic Networks with Probabilistic Logic.* In: *Bisociative Knowledge Discovery - An Introduction to Concept, Algorithms, Tools, and Applications*, pp. 407–426, doi:`10.1007/978-3-642-31830-6_29`.

[17] Hengfei Li, Nir Oren & Timothy J. Norman (2011): *Probabilistic Argumentation Frameworks.* In: *Theories and Applications of Formal Argumentation - First International Workshop, TAFA 2011. Revised Selected Papers*, *Lecture Notes in Computer Science* 7132, Springer, pp. 1–16, doi:`10.1007/978-3-642-29184-5_1`.

[18] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester & Luc De Raedt (2018): *DeepProbLog: Neural Probabilistic Logic Programming.* In: *Advances in Neural Information Processing Systems 31: NeurIPS 2018*,, pp. 3753–3763. Available at `https://proceedings.neurips.cc/paper/2018/hash/dc5d637ed5e62c36ecb73b654b05ba2a-Abstract.html`.

[19] Theofrastos Mantadelis & Stefano Bistarelli (2020): *Probabilistic abstract argumentation frameworks, a possible world view.* *Int. J. Approx. Reason.* 119, pp. 204–219, doi:`10.1016/j.ijar.2019.12.006`.

[20] Luc De Raedt & Angelika Kimmig (2015): *Probabilistic (logic) programming concepts.* *Mach. Learn.* 100(1), pp. 5–47, doi:`10.1007/s10994-015-5494-z`.

[21] Taisuke Sato (1995): *A Statistical Learning Method for Logic Programs with Distribution Semantics.* In: *Proceedings of the Twelfth International Conference on Logic Programming*, pp. 715–729.

[22] Francesca Toni (2014): *A tutorial on Assumption-based Argumentation.* *Arg. & Comp* 5(1), pp. 89–117.

[23] Pietro Totis, Angelika Kimmig & Luc De Raedt (2023): *smProbLog: Stable Model Semantics in ProbLog for Probabilistic Argumentation.* *Theory and Practice of Logic Programming*, p. 1–50, doi:`10.1017/S147106842300008X`.

Towards a Representation of Decision Theory Problems with Probabilistic Answer Set Programs

Damiano Azzolini Elena Bellodi Fabrizio Riguzzi
University of Ferrara
Ferrara, Italy
{damiano.azzolini,elena.bellodi.fabrizio.riguzzi}@unife.it

Probabilistic Answer Set Programming under the Credal Semantics is a recently introduced formalism to express uncertainty with answer set programming where the probability of a query is described by a range. We propose to extend it to encode decision theory problems.

1 Encoding

Probabilistic Answer Set Programming under the Credal Semantics [5] extends answer set programming [3] with ProbLog [6] probabilistic facts of the form $\Pi :: f$, where Π is a floating point value (between 0 and 1) representing the probability of f of being true. A selection of a truth value for every probabilistic fact identifies a *world* whose probability is the product of the probabilities of the probabilistic facts true times the product of one minus the probability of every probability fact false. In formula:

$$P(w) = \prod_{f_i \text{ true} \in w} \Pi_i \cdot \prod_{f_i \text{ false} \in w} (1 - \Pi_i).$$

In Probabilistic Logic Programming [6] (PLP), a world is a Prolog program, so it has exactly one model. In Probabilistic Answer Set Programming under the credal semantics (PASP), every world is an answer set program [8], so it has zero or more (stable) models. Every world is required to have at least one stable model. In PASP, the probability of a query (i.e., a conjunction of ground atoms) is represented with a probability range, composed of a lower and upper bound. A world contributes to both the lower and upper bounds if the query is true in every answer set while it contributes only to the upper bound if it is true in some of the answer sets. That is, $P(q) = [\underline{P}(q), \overline{P}(q)]$ where

$$\underline{P}(q) = \sum_{w_i | \forall m \in AS(w_i),\ m \models q} P(w_i),\ \overline{P}(q) = \sum_{w_i | \exists m \in AS(w_i),\ m \models q} P(w_i).$$

The probability bounds can be computed with, for example, the tool described in [1]. The DTProbLog framework [4] allows to express decision theory problems with a ProbLog [6] program. It introduces a set U of utility atoms of the form $utility(a,r)$ where r is the reward obtained to satisfy a (i.e., when a is true) and a set of *decision atoms* D. Every subset of decision atoms defines a *strategy* σ (a ProbLog program) and its utility is computed as

$$Util(\sigma) = \sum_{(a,r) \in U} r \cdot P_\sigma(a)$$

where $P_\sigma(a)$ is the probability of a computed in the ProbLog program identified by fixing the decision atoms in σ to true. The goal is to find the strategy σ^* that maximizes the utility, i.e., $\sigma^* = argmax_\sigma Util(\sigma)$.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 190–192, doi:10.4204/EPTCS.385.19

We apply this framework to PASP, where every world has one or more stable models. Thus, the utility is described by a lower and an upper bound:

$$Util(\sigma) = \left[\underline{Util}(\sigma), \overline{Util}(\sigma)\right] = \left[\sum_{(a,r)\in U} r \cdot \underline{P}_{\sigma}(a), \sum_{(a,r)\in U} r \cdot \overline{P}_{\sigma}(a)\right]$$

We have now to select whether to optimize the lower or upper utility, and so there are two possible goals:

$$\sigma^* = argmax_{\sigma}(\underline{Util}(\sigma)).$$

or

$$\sigma^* = argmax_{\sigma}(\overline{Util}(\sigma)).$$

depending on the considered target probability. Note that with this representation the lower utility can be greater than the upper utility, when, for example, there are both positive and negative rewards. We consider here only the strategies where the lower utility is less or equal than the upper utility. To clarify, consider this simple program:

```
0.7::pa. 0.2::pb.
decision da. decision db.
utility(r0,3). utility(r1,-11).

r0:- pa, da.
r0;r1:- pb, db.
```

pa and *pb* are two probabilistic facts and *da* and *db* two decision atoms. The rewards we get to satisfy $r0$ and $r1$ are respectively 3 and -11. There are 4 possible strategies (no decision atoms selected, only one decision atom selected, and both decision atoms selected). Each of these have 4 worlds (no probabilistic facts true, only one probabilistic fact true, and both probabilistic facts true). If we consider the strategy $\sigma_{da} = \{da\}$, we have $P_{\sigma_{da}}(r0) = [0.7, 0.7]$ and $P_{\sigma_{da}}(r1) = [0,0]$ (only the first rule is considered), and $Util(\sigma_{da}) = [3 \cdot 0.7 + (-11) \cdot 0, 3 \cdot 0.7 + (-11) \cdot 0] = [2.1, 2.1]$. If we repeat this process for all the strategies and we consider the upper utility as target, we get that $\sigma_{da} = \{da\}$ is the one that maximizes the upper utility.

A naive algorithm consists in enumerating all the possible strategies and then computing the lower and upper probability for every decision atom [1]. However, this clearly cannot scale, since the number of possible strategies is $2^{|D|}$. Moreover, the algorithm of [1] adopts enumeration of stable models and projection on probabilistic facts [7], that also requires the generation of an exponential number of answer sets. Thus, a practical solution should adopt both approximate methods for the generation of the possible strategies and for inference. For the former, greedy algorithms such as genetic algorithms [9] can be a possible solution. For inference, approximate methods based on sampling, such as the one proposed in [2] can be of interest.

Acknowledgements

This research was partly supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No. 952215.

References

[1] Damiano Azzolini, Elena Bellodi & Fabrizio Riguzzi (2022): *Statistical Statements in Probabilistic Logic Programming*. In Georg Gottlob, Daniela Inclezan & Marco Maratea, editors: *Logic Programming and Nonmonotonic Reasoning*, Springer International Publishing, Cham, pp. 43–55, doi:10.1007/978-3-031-15707-3_4.

[2] Damiano Azzolini, Elena Bellodi & Fabrizio Riguzzi (2023): *Approximate Inference in Probabilistic Answer Set Programming for Statistical Probabilities*. In Agostino Dovier, Angelo Montanari & Andrea Orlandini, editors: *AIxIA 2022 – Advances in Artificial Intelligence*, Springer International Publishing, Cham, pp. 33–46, doi:10.1007/978-3-031-27181-6_3.

[3] Gerhard Brewka, Thomas Eiter & Mirosław Truszczyński (2011): *Answer Set Programming at a Glance*. *Communications of the ACM* 54(12), pp. 92–103, doi:10.1145/2043174.2043195.

[4] Guy Van den Broeck, Ingo Thon, Martijn van Otterlo & Luc De Raedt (2010): *DTProbLog: A Decision-Theoretic Probabilistic Prolog*. In Maria Fox & David Poole, editors: *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI Press, pp. 1217–1222.

[5] Fábio Gagliardi Cozman & Denis Deratani Mauá (2020): *The joy of Probabilistic Answer Set Programming: Semantics, complexity, expressivity, inference*. *International Journal of Approximate Reasoning* 125, pp. 218–239, doi:10.1016/j.ijar.2020.07.004.

[6] Luc De Raedt, Angelika Kimmig & Hannu Toivonen (2007): *ProbLog: A Probabilistic Prolog and Its Application in Link Discovery*. In Manuela M. Veloso, editor: *20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, 7, AAAI Press, pp. 2462–2467.

[7] Martin Gebser, Benjamin Kaufmann & Torsten Schaub (2009): *Solution Enumeration for Projected Boolean Search Problems*. In Willem-Jan van Hoeve & John Hooker, editors: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Springer Berlin Heidelberg, pp. 71–86, doi:10.1007/978-3-642-01929-6_7.

[8] Michael Gelfond & Vladimir Lifschitz (1988): *The stable model semantics for logic programming*. In: *5th International Conference and Symposium on Logic Programming (ICLP/SLP 1988)*, 88, MIT Press, pp. 1070–1080.

[9] Xin-She Yang (2021): *Chapter 6 - Genetic Algorithms*. In Xin-She Yang, editor: *Nature-Inspired Optimization Algorithms (Second Edition)*, second edition edition, Academic Press, pp. 91–100, doi:10.1016/B978-0-12-821986-7.00013-5.

On Feasibility of Declarative Diagnosis

Włodzimierz Drabent
Institute of Computer Science, Polish Academy of Sciences
`drabent` *at* `ipipan` *dot* `waw` *dot* `pl`

The programming language Prolog makes declarative programming possible, at least to a substantial extent. Programs may be written and reasoned about in terms of their declarative semantics. All the advantages of declarative programming are however lost when it comes to program debugging. This is because the Prolog debugger is based solely on the operational semantics. Declarative methods of diagnosis (i.e. locating errors in programs) exist, but are neglected. This paper discusses their possibly main weaknesses and shows how to overcome them. We argue that useful ways of declarative diagnosis of logic programs exist, and should be usable in actual programming.

Keywords: declarative diagnosis / algorithmic debugging, Prolog, coroutining, program correctness, program completeness

1 Introduction

The main concept of logic programming is that a program is a set of formulae, and computation produces its logical consequences. Such program can be understood declaratively – not as a description of any computation, but rather as a description of a problem to solve. Its answers depend solely on its "logic", i.e. on the formulae of which it consists. So logic programming is a declarative programming paradigm. The programmer can construct programs and reason about them at a higher level of logic, abstracting from their computations. Whole reasoning about program correctness (more precisely, correctness and completeness) can be done at this level. This is an important advantage of the paradigm. One needs to resort to the operational semantics (the "control") only to deal with termination and efficiency. Modifying the control does not change (the set of) the answers of the program. What is changed is the way they are computed; the logic is separated from the control [Kow79]. The programming language Prolog was introduced as an implementation of logic programming. It is still the major logic programming language. It makes declarative programming possible, at least to a substantial extent.

On the other hand, a Prolog program can be looked at from a point of view of its operational semantics. A program is a precise description of computations; one can program in Prolog imperatively. This is often necessary, for instance when a program has to interact with its environment. Dealing with the operational semantics is often difficult, due to among others the non-straightforward nature of backtracking, coroutining and tabulation.

An important activity in programming is program debugging. And when it comes to debugging of Prolog programs, all the declarativeness is lost. Declarative methods, known as declarative diagnosis (DD) or algorithmic debugging, exist but are neglected. Basically, the only tool available for a programmer is the Prolog debugger, which is based solely on the operational semantics. So the tool is incompatible with declarative programming (cf. e.g. [Llo87, DN94, Dra19]). To use the debugger the programmer has to go into details of the operational semantics, abandoning the declarative thinking. This makes the debugging difficult, and encourages programmers to resign from the declarative view of programs. The difficulties are even more serious when more sophisticated control is involved, like coroutining or tabulation.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 193–200, doi:10.4204/EPTCS.385.20

As the Prolog debugger is a rather powerful tool, one may expect that a "declarative programmer" can obtain from it the information she needs. An example of such information is which premises have been used to derive a given answer, displayed at an `exit` port. Obtaining such information is possible, but surprisingly tedious [Dra19].

In the next section we briefly introduce declarative diagnosis (DD), present a unifying view of incorrectness and incompleteness diagnosis, and suggest a kind of suitable DD tools. Then (Section 3) we identify the intended model problem, which has possibly been the main obstacle for acceptance of DD, and show how to overcome it. In Section 4 we discuss applying DD to Prolog with tabulation and delays.

Preliminaries. The presentation is rather informal, in most cases references are given to a more precise treatment. In some places the terminology of logic is used [Apt97], instead of that of Prolog manuals. So by an *atom* we mean an atomic formula, not a Prolog constant. We consider SLD-resolution dealing with *queries* (conjunctions of atoms), instead of goals (negated queries). From a programmer's point of view, the selected atom in a query in an SLD-derivation is a *procedure call*. By a (*computed*) *answer* of a program we mean the result of applying a (computed) answer substitution to the initial query. The set of clauses (in a program) beginning with a predicate symbol p will often be called *procedure* p. The Herbrand base will be denoted by $\mathcal{HB}$, and the least Herbrand model of a program P by $\mathcal{M}_P$.

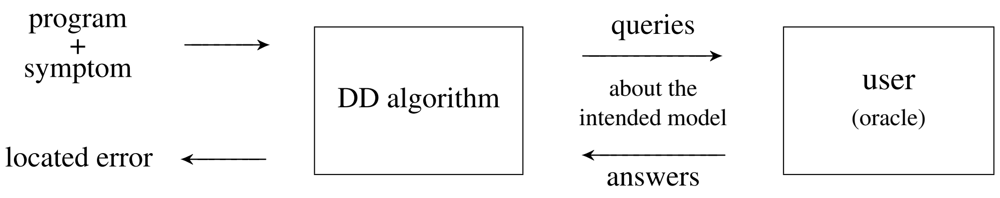
In imperative and functional programming, program correctness consists of program termination and partial correctness. The latter means that the program produces required results provided it terminates. In logic programming, due to its nondeterministic nature, partial correctness splits into *correctness* and *completeness* [Hog84, DM05, Dra16]. Correctness means that each answer of the program is compatible with the specification. Completeness means that all the answers required by the specification are produced by the program. Obviously, termination is a property of the operational semantics, in contrast to correctness and completeness. Debugging consists of program *diagnosis*, i.e. locating errors in programs, and program correction. In this paper we are interested in diagnosing incorrectness and incompleteness for logic / Prolog programs.

2 Declarative diagnosis

The idea of declarative diagnosis was introduced by Shapiro in his seminal thesis [Sha83], under the name algorithmic debugging. Shapiro gave algorithms for diagnosing incorrectness and incompleteness (and also non-termination). His ideas were followed and developed by further work, dealing mainly with logic programming, but also with other paradigms (see e.g. [Per86, DNTM89, Nai92, CRS17]).

Specifications, symptoms, errors. Diagnosis starts with a *symptom* of program incorrectness or incompleteness. The purpose is to locate in a program an *error*, i.e. a fragment responsible for the symptom. In incorrectness diagnosis a symptom is an incorrect answer of the program. In diagnosing incompleteness it is, generally speaking, a missing answer. Here by an *incompleteness symptom* we mean an atomic query for which the program terminates, but it does not produce some required answers.

Diagnosis must be based on a specification. In DD the specification is the *intended model* of the program. This is the least Herbrand model of the target program. A DD algorithm is informed about the intended model by an *oracle*, which answers queries about the model. Usually the oracle is the user.



In incorrectness diagnosis, an *incorrectness error* is an incorrect clause. In such clause the body atoms are true, and the head is not true in the intended specification. In logical terms $M \not\models C$, where M is the intended model and C the clause.

By an *incompleteness error* we will mean an atomic query A whose answer (required by the specification) cannot be produced by any clause of the program (out of atomic queries required by the specification). Logically, we have an instance $A\theta \in M$ such that the program does not contain a clause with an instance $A\theta \leftarrow \vec{B}$ such that $M \models \vec{B}$, where M is the intended model. So an incompleteness error points at a procedure responsible for the error (and presents an atom the procedure should additionally produce).

It can be shown that a more precise notion of an error is impossible. For instance, an incorrect clause can be corrected in various ways, leaving various its fragments unchanged. So we cannot consistently name such a fragments as correct, or incorrect.

DD algorithms, a unifying view. For incompleteness, we prefer Pereira style diagnosing [Nai92], as it is more convenient than that of [Sha83]. In such context, both incorrectness and incompleteness diagnosis can be treated as instances of a single algorithm. (See [Nai97, CRS17] for a similar observation.)

Let us first describe the oracle queries. A query in incorrectness diagnosis is an atom A (which was obtained as an answer in the computation). In incompleteness diagnosis it is an atom A (a procedure call from the computation) together with the computed answers $A\theta_1, \ldots, A\theta_n$ $(n \geq 0)$ obtained for A. The oracle has to answer whether the query is a symptom (of correctness or completeness, respectively).

The algorithm inspects an erroneous computation of the program, and builds builds a *DD search tree* of oracle queries. The root is the symptom we begin with. The tree contains a *target*, i.e. a node which is a symptom and its children are not. The algorithm employs the oracle to search the tree for a target. Now we give further details of the tree and explain how a target determines an error.

In incorrectness diagnosis the tree is a proof tree (the root is a wrong answer, each node together with its children is an instance of a program clause). Thus a target with its children is an incorrectness error.

For incompleteness diagnosis, each node $\mathscr{A}$ (which is an atom A with its computed answers) has children that are the top-level queries (together with the answers) used in the computation for A. (By top-level we mean one that is an instance of a body atom of a clause used to resolve A.) If $\mathscr{A}$ is a target and its children are not, then $\mathscr{A}$ contains an incompleteness error (which is the procedure call from $\mathscr{A}$). Informally, as we found no incompleteness related to the body atoms of the applicable clauses, the clauses are responsible (for the initial symptom).

It is important that with a DD algorithm the user can locate the error without looking at the program. Also, all the diagnosis is performed solely in terms of the declarative semantics. So we do not need to understand the operational semantics.

Non-algorithmic declarative diagnosis. Practice shows some inconveniences of using DD algorithms, at least in their basic form (cf. e.g. [DNTM89]). An inconvenient and counterproductive feature is that the order of queries is determined by the algorithm. The effort (and time) to answer various queries may differ substantially,[1] so it is desirable to first deal with queries considered by the user to be easier. As a result, some difficult queries many eventually not be asked. Also, it should be possible to modify one's previous answer, as errors in answering are possible. Such modifications are also useful in making and withdrawing temporary assumptions ("what if this were correct").

Such inconveniences disappear when the search is performed by the user. It is sufficient that a DD tool extracts from the computation the DD search tree, and gives the user convenient means to explore the tree. The author's experience with such prototype tools shows that they are more convenient to use than

---

[1] So it is not realistic to judge the efficiency of DD algorithms by means of query complexity [Sha83, CRS17], which deals solely with the number of queries.

(prototype implementations of) DD algorithms. The main burden – with understanding and answering queries – is the same (only the answers are not typed in). Additional human effort, due to performing the search, is minor. Being relieved of the restraints of DD algorithms is a substantial convenience. The user may look at various queries and choose which to answer first. Some answers may be postponed, the user may inspect the tree freely. An already visited part of the tree may be re-inspected, to correct former decisions (if erroneous, or assumed temporarily).

What we propose is, in a sense, a declarative counterpart of the Prolog debugger. The debugger gives the user access to the operational semantics of a program; we show how to give access to the declarative semantics.

3 The intended model problem

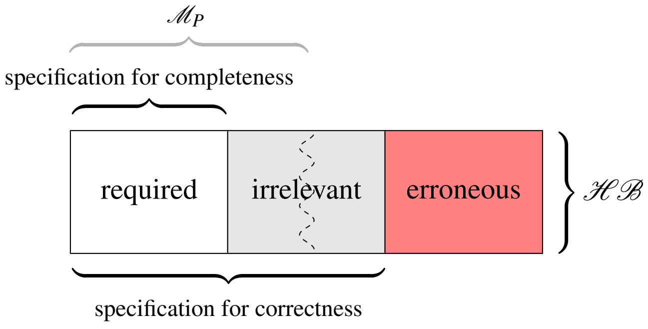
In this section, based on [Dra16, Section 7], we discuss what possibly is the main reason for lack of acceptance of DD in practice. Namely a fundamental notion of DD is that of the intended model. This means the the programmer has to know exactly the intended least Herbrand (2-valued) model of her program; in other words, to know exactly the relation to be defined by each predicate. This is usually not realistic.

As an example [Dra16] consider insertion sort and predicate insert/3, dealing with inserting an element into a sorted list (to obtain a sorted list as a result). The user does not (and should not) know how the predicate should behave on unsorted lists. Hence she is unable to answer an oracle query like "Is atom $B = insert(2,[3,1],[2,3,1])$ correct"? Let us call this phenomenon the *intended model problem*.

Both treating B as correct, or incorrect may be reasonable, and result in different debugged programs. For instance treating all such atoms as incorrect leads to a program in which insert/3 checks that its second argument is a list. Such program is inefficient. So choosing the intended model may then force certain design decisions, possibly undesirable ones. Often while developing a program it is impossible to know in advance its exact semantics (i.e. its intended model). Apparently the semantics of insert/3 (in a standard insertion sort program [SS94, Program 3.21]) is a result of making the program simple and efficient, and has not been decided in advance. See [Dra18] for a larger example.

In the author's opinion the intended model problem makes DD inapplicable in many, if not in most of, practical cases.

Approximate specifications. Such examples show a common feature of specifications in logic programming (both formal, and informal ones which programmers have in their minds): often such specifications have a 3-valued flavour. Some atoms are clearly true, some false, and the remaining ones are irrelevant. They may be accepted as answers, but should not be required to be answers of the program.



So instead of specifying a single intended model, we should provide two specifications, S^0 for completeness and S for correctness (where $S^0 \subseteq S$). The irrelevant atoms, like B in our example are in $S \setminus S^0$. Let us call such pair S^0, S of specifications an *approximate specification*.

A program P is correct w.r.t. an approximate specification S^0, S if $S \models A$ for each answer A of P (i.e. $P \models A$ implies $S \models A$); P is complete w.r.t. it if $S^0 \models A$ implies that A is an answer of P ($S^0 \models A$ implies $P \models A$).

A reasonable approximate specification for insert/3 of insertion sort is:

$$\text{for completeness:}\quad S^0_{insert} = \left\{ insert(n,l_1,l_2) \in \mathcal{HB} \;\middle|\; \begin{array}{l} l_1, l_2 \text{ are sorted lists of integers,} \\ elms(l_2) = \{n\} \cup elms(l_1) \end{array} \right\},$$

where $elms(l)$ – the multiset of elements of l,

$$\text{for correctness:}\quad S_{insert} = \left\{ insert(n,l_1,l_2) \in \mathcal{HB} \;\middle|\; \begin{array}{l} \text{if } n \text{ is an integer and} \\ l_1 \text{ is a sorted list of integers,} \\ \text{then } insert(n,l_1,l_2) \in S^0_{insert} \end{array} \right\}.$$

Note that neither S^0_{insert}, nor S_{insert} corresponds to the semantics of the actual corrected insertion sort program (i.e. neither of them is the set of atoms of the form $insert(\vec{t})$ from its least Herbrand model).

Obviously, diagnosis of incorrectness and that of incompleteness should be done employing the respective specification. With such approach to specifications, the standard DD methods (based on intended, 2-valued interpretations) are sufficient.

The intended model problem has been noticed, among others, by [Per86, DNTM89, Nai00, Fer93]. Pereira [Per86] introduces inadmissible atoms, which should not appear in the computation. In this way the debugging is no more declarative as it refers also to the intended operational semantics of the program.

Naish [Nai00] introduces a rather sophisticated 3-valued approach to DD, and two kinds of errors (e-bug and i-bug). The atoms from $S_{insert} \setminus S^0_{insert}$ above have in the 3-valued intended model the third logical value **i**. (Those from S^0_{insert} are **t**, and those from $\mathcal{HB} \setminus S_{insert}$ are **f**.) It seems strange, that both incorrectness and incompleteness diagnosis treat in the same way the **i** atoms; the discussion above shows that this should not be the case.

Any such complications are not necessary when we have separate specifications for correctness and completeness. Introducing such specifications solves the intended model problem. In the author's opinion this is a main step to make DD practical.

Negation and partial specifications. When we deal with logic programs with negation, a specification should also describe the negative results of programs; roughly, which ground atoms fail. In other words, which negated literals are program consequences under the chosen semantics.

As in the discussion above, a specification often has to distinguish between those atoms that have to fail (specifying completeness) from those that may fail (specifying correctness). To specify both positive and negative program answers, it is natural to use an approximate specification S^0, S as introduced above, and interpret it as follows [DM05]. The atoms that are not allowed to succeed have to fail. The atoms that are required to succeed cannot fail. Thus the set of atoms that have to fail is $\mathcal{HB} \setminus S$ (the "erroneous" in the diagram). Similarly, the set of atoms that may fail is $\mathcal{HB} \setminus S^0$ (the "irrelevant" and "erroneous" in the diagram). In a correct program, if $\neg A$ is its answer then $S^0 \models \neg A$. In a complete program, if $S \models \neg A$. then $\neg A$ is an answer. So the description of correctness for positive answers describes completeness of negative ones, and vice versa.

We do not discuss here DD of programs with negation [Llo87, DNTM89, NT90]. We only mention that finding $\neg A$ as a target in incorrectness diagnosis leads to incompleteness diagnosis for A (and vice versa).

4 Extracting the DD search tree.

This section considers implementing declarative diagnosis for Prolog with tabulation and coroutining. It turns out that only incompleteness diagnosis for coroutining poses problems. We discus two ways of dealing with them.

DD requires extracting the necessary information (i.e. the DD search tree) from a computation of the program. For basic Prolog (without tabulation and coroutining) we know how to do this, see e.g. [Sha83, DNTM89, Nai92]; we skip further details. For incorrectness diagnosis this means extracting the proof tree. (Most of) the methods to do this for basic Prolog seem easy to generalize for coroutining and tabulation. Such generalization is far from obvious for incompleteness diagnosis.

For incompleteness diagnosis (in Pereira-style) one needs to collect from the computation the procedure calls, and for each of them the obtained computed answers. For basic Prolog, a simplest way to do this seems top-level meta-interpretation for the considered procedure call (cf. [DNTM89], where however the DD search tree is not made explicit). So for a call $p(\vec{t})$, all the clauses $p(\vec{s}) \leftarrow B_1, \ldots, B_n$ are meta-interpreted, however each encountered procedure call $B_i\theta_j$ is executed by the Prolog system. All such calls (together with their answers) are children in the DD search tree of the node $p(\vec{t})$ (with its answers).

Other ways of obtaining the search tree for incompleteness DD may be suitable. In particular, obtaining the whole tree from a single computation may be preferable (as this mimics the actual side effects of a not purely declarative program).

Transferring such ideas to Prolog with tabulation should not create problems. The task is however difficult for programs with coroutining. This is because the answers for a given procedure call may actually not be computed (during the computation for the considered initial query). The query instance returned by Prolog as an answer (e.g. that appearing at the Exit port of Prolog debugger) will be called here a *pseudo-answer*. See [Dra23] for a formal description of coroutining.

Let us explain. During a computation for a sub-query A it may happen that (i) unblocking of a formerly delayed sub-query results in a pseudo-answer being an instance of an actual computed answer, or (ii) delaying of a sub-query results in a pseudo-answer being more general than an actual one (or an actual answer may not exist). When (i) and (ii) coincide, nothing can be concluded.

For sound incompleteness diagnosis, a DD query should contain actual computed answers for a given sub-query A, but what we can obtain from the computation are pseudo-answers. This difficulty was discussed in [Nai92], without a general solution. Executing A without coroutining is of no help in general, as it may result in non-termination. It is useful to do this under a suitable time limit; if A terminates, the problem is solved. For a general case consider the following. From a computation trace one can find if cases (i) or (ii) have been involved in producing a given pseudo answer for A. If not, the pseudo-answer is an answer. If (i) has not occurred then the pseudo-answer is more general than the actual answer (or the latter does not exist). If (ii) has not occurred, the pseudo-answer is an instance of an actual one. A pseudo-answer together with such information can be included in a DD query, instead of the unknown actual answer; the information makes it possible in some cases to answer the query (formally, to state whether A with the actual answers is an incompleteness symptom).[2]

This approach may fail to find that, despite of the involved coroutining, a pseudo-answer is an answer. We propose another method of checking this, and to find the actual answer in some cases.

From the computation producing a pseudo-answer $A\theta$ for A (w.r.t. a program P) a proof tree may be collected as in incorrectness diagnosis. Due to delays, some subtrees of an actual proof tree may be missing, so the obtained tree will be called a *pseudo-proof tree*. Such tree T, with root $A\theta$, is inspected to find if it is (a) a proof tree or (b) not. For (a), $A\theta$ is an answer of P, but it may not be a computed

---

[2] If case (i) has not occurred for any pseudo-answer for A, and some required answer is not an instance of any displayed (pseudo-)answer, then A (with its possibly unknown answers) is surely a symptom. If case (ii) has not occurred for any pseudo-answer for A, and each answer required by the specification is an instance of some displayed one, then A (with answers) is surely not a symptom.

answer for A (but its instance, case (i)). Out of the tree we can obtain the clauses of P used to compute $A\theta$. Applying the clauses, we can obtain a computed answer $A\sigma$ for A, so that T is an instance of the proof tree T' for $A\sigma$ (and $A\sigma$ is more general than $A\theta$). Now $A\sigma$ can be used in a DD query,

The same procedure can be applied to T in case (b), it results in a pseudo-proof tree T' with root $A\sigma$, more general than $A\theta$. The tree corresponds to executing A alone (in a context without any pending delayed sub-queries). Formally, T' corresponds to a prefix D of an SLD-derivation for A; the last query Q of D consists of the delayed sub-queries. So any computed answer for A obtained by completing the derivation is an instance of $A\sigma$. Thus $A\sigma$ can be used instead of the unknown answer in a DD query, and be treated like a pseudo-answer for which (i) has not occurred (cf. footnote 2).

As previously, it makes sense to execute Q under a time limit. If it terminates, then all the derivations with D as a prefix have been found. They provide computed answers for A, to be used in a DD query (instead of the pseudo-answer $A\theta$).

To summarize, generalizing DD to take care of tabulation and coroutining seems rather obvious, except for diagnosing incompleteness for coroutining. In this case some DD queries cannot be extracted from the computation of the program. We discussed how to deal with this problem. For most of cases we showed a way to obtain a sound answer for such an undetermined DD query. There is one exception (in case (b) above, when the delayed Q does not terminate under a given time limit, one cannot determine that the DD query is not a symptom). We do not pursue here another possible approach, based on detailed monitoring of variable bindings during the computation.

5 Conclusion

Comments. Due to lack of space, we do not discuss various issues related to pragmatics of DD (declarative diagnosis), and to Prolog features apart basic ones. The importance of a user friendly interface is obvious (it should be possible to view big terms effectively, to conveniently inspect DD search trees, etc). Experience from actual debugging is needed to evaluate diagnosis approaches, and to better understand which features a useful diagnosis tool should have. The author is interested in buggy programs that could be used in such experiments, preferably with challenging or otherwise interesting bugs.

DD diagnosis turns out to be applicable also to programs with non-declarative fragments. Of course this concerns only issues related to the program answers (and not e.g. the sequences of input/output actions). For instance, prototype "non-algorithmic" DD tools (cf. Section 2) for incorrectness and incompleteness have been used to debug themselves. Locating errors with these tools seems substantially simpler than with the Prolog debugger.

Debugging for ASP is a separate issue, not dealt with here (as the role of answers in ASP is different from that in standard logic programming). An interesting challenge is to generalize DD so that more symptoms (of possibly a single error) can be used to locate the error more efficiently.

Summary. This note deals with DD of logic programs. We treat incorrectness diagnosing and incompleteness diagnosing (Pereira style) as instances of the same algorithm. We point out the intended model problem as possibly the main reason for lack of acceptance of DD in practice. We also consider DD for programs with tabulation and coroutining, and propose how to cope with the difficulties that arise. We advocate that a tool which allows the user to inspect the DD search tree is more suitable to perform diagnosis than an implementation of a whole DD algorithm.

References

[Apt97] K. R. Apt (1997): *From Logic Programming to Prolog*. International Series in Computer Science, Prentice-Hall.

[CRS17] R. Caballero, A. Riesco & J. Silva (2017): *A Survey of Algorithmic Debugging*. ACM Comput. Surv. 50(4), pp. 60:1–60:35, doi:10.1145/3106740.

[DM05] W. Drabent & M. Miłkowska (2005): *Proving correctness and completeness of normal programs – a declarative approach*. TPLP 5(6), pp. 669–711, doi:10.1017/S147106840500253X.

[DN94] Mireille Ducassé & Jacques Noyé (1994): *Logic Programming Environments: Dynamic Program Analysis and Debugging*. J. Log. Program. 19/20, pp. 351–384, doi:10.1016/0743-1066(94)90030-2.

[DNTM89] W. Drabent, S. Nadjm-Tehrani & J. Małuszyński (1989): *Algorithmic Debugging with Assertions*. In H. Abramson & M. H. Rogers, editors: Meta-Programming in Logic Programming, The MIT Press, pp. 501–522.

[Dra16] W. Drabent (2016): *Correctness and Completeness of Logic Programs*. ACM Trans. Comput. Log. 17(3), pp. 18:1–18:32, doi:10.1145/2898434.

[Dra18] W. Drabent (2018): *Logic + control: On program construction and verification*. Theory and Practice of Logic Programming 18(1), pp. 1–29, doi:10.1017/S1471068417000047.

[Dra19] W. Drabent (2019): *The Prolog Debugger and Declarative Programming*. In M. Gabbrielli, editor: LOPSTR 2019, Revised Selected Papers, Lecture Notes in Computer Science 12042, Springer, pp. 193–208, doi:10.1007/978-3-030-45260-5_12.

[Dra23] W. Drabent (2023): *Implementing backjumping by means of exception handling*. arXiv. Available at `https://doi.org/10.48550/arXiv.2305.16137`.

[Fer93] G. Ferrand (1993): *The Notions of Symptom and Error in Declarative Diagnosis of Logic Programs*. In P. Fritzson, editor: AADEBUG'93, Proceedings, LNCS 749, Springer, pp. 40–57, doi:10.1007/BFb0019399.

[Hog84] C. J. Hogger (1984): *Introduction to Logic Programming*. Academic Press, London.

[Kow79] Robert A. Kowalski (1979): *Algorithm = Logic + Control*. Commun. ACM 22(7), pp. 424–436. Available at `http://doi.acm.org/10.1145/359131.359136`.

[Llo87] J. W. Lloyd (1987): *Declarative error diagnosis*. New Gener Comput 5, pp. 133–154. Available at `https://doi.org/10.1007/BF03037396`.

[Nai92] L. Naish (1992): *Declarative Diagnosis of Missing Answers*. New Generation Comput. 10(3), pp. 255–286. Available at `http://dx.doi.org/10.1007/BF03037939`.

[Nai97] L. Naish (1997): *A Declarative Debugging Scheme*. J. Funct. Log. Program. 1997(3). Available at `http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/1997/A97-03/A97-03.html`.

[Nai00] L. Naish (2000): *A Three-Valued Declarative Debugging Scheme*. In: 23rd Australasian Computer Science Conference (ACSC 2000), IEEE Computer Society, pp. 166–173. Available at `http://doi.ieeecomputersociety.org/10.1109/ACSC.2000.824398`.

[NT90] S. Nadjm-Tehrani (1990): *Debugging Prolog Programs Declaratively*. In: META90: 2nd Workshop on Meta-Programming in Logic Programming, Leuven.

[Per86] L. M. Pereira (1986): *Rational Debugging in Logic Programming*. In E. Y. Shapiro, editor: ICLP, Lecture Notes in Computer Science 225, Springer, pp. 203–210. Available at `http://dx.doi.org/10.1007/3-540-16492-8_76`. Extended version at `https://userweb.fct.unl.pt/~lmp/`.

[Sha83] E. Shapiro (1983): *Algorithmic Program Debugging*. The MIT Press.

[SS94] L. Sterling & E. Shapiro (1994): *The Art of Prolog*, 2 edition. The MIT Press. Available at `https://mitpress.mit.edu/books/art-prolog-second-edition`.

Explainable and Trustworthy Traffic Sign Detection for Safe Autonomous Driving: An Inductive Logic Programming Approach

Zahra Chaghazardi
Department of Computer Science, University of Surrey
United Kingdom
z.chaghazardi@surrey.ac.uk

Saber Fallah
Connected and Autonomous Vehicles Lab, University of Surrey
United Kingdom
s.fallah@surrey.ac.uk

Alireza Tamaddoni-Nezhad
Department of Computer Science, University of Surrey
United Kingdom
a.tamaddoni-nezhad@surrey.ac.uk

Traffic sign detection is a critical task in the operation of Autonomous Vehicles (AV), as it ensures the safety of all road users. Current DNN-based sign classification systems rely on pixel-level features to detect traffic signs and can be susceptible to adversarial attacks. These attacks involve small, imperceptible changes to a sign that can cause traditional classifiers to misidentify the sign. We propose an Inductive Logic Programming (ILP) based approach for stop sign detection in AVs to address this issue. This method utilises high-level features of a sign, such as its shape, colour, and text, to detect categories of traffic signs. This approach is more robust against adversarial attacks, as it mimics human-like perception and is less susceptible to the limitations of current DNN classifiers. We consider two adversarial attacking methods to evaluate our approach: Robust Physical Perturbation (PR2) and Adversarial Camouflage (AdvCam). These attacks are able to deceive DNN classifiers, causing them to misidentify stop signs as other signs with high confidence. The results show that the proposed ILP-based technique is able to correctly identify all targeted stop signs, even in the presence of PR2 and ADvCam attacks. The proposed learning method is also efficient as it requires minimal training data. Moreover, it is fully explainable, making it possible to debug AVs.

1 Introduction

The popularity of AVs is rising rapidly because of their potential to reduce human error on the road, leading to safer transportation. AVs are believed to make more accurate perceptions and react faster than humans. Deep Neural Networks (DNNs) play a significant role in developing perception systems for AVs. However, DNNs face significant challenges that must be addressed before AVs can be deployed safely [7]. The major challenges facing DNN-based vision systems in autonomous driving are discussed below.

DNN-based systems are often considered "black boxes" because their logic is not transparent. Since it is difficult to explain how the system makes the prediction, it is challenging to debug them when they make a wrong decision. For example, misclassifying objects, such as mistaking shadows for pedestrians,

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 201–212, doi:10.4204/EPTCS.385.21

is a common problem in AVs and making decisions based on these misclassifications can lead to fatal accidents. Considering the fatal Uber accident [23], given that the AV's DNN-based decision-making is opaque, there is no way to debug the system and ensure such mistakes do not happen again. Moreover, using algorithms with ambiguous logic makes it impossible to evaluate and trust them. This means that regulatory approval is not applicable to stochastic-based AV vehicles.

Furthermore, DNNs face significant challenges when it comes to learning from small data and achieving out-of-distribution generalizability and transferability to new domains. In real-world scenarios, particularly in security domains, there is often a lack of large, annotated, and carefully curated data sets to train these systems. This can make it difficult for DNNs to acquire knowledge from a few examples and transfer it to new domains, unlike humans, who can do so with ease. Anomaly detection tasks, in particular, are affected by this challenge due to the rarity of anomalous data. Anomalies can be caused by errors, faults, or adversarial attacks, which can lead to security and safety hazards. Adversarial examples provide evidence of a network's weakness in achieving high generalisation performance [31]. Improving generalizability is crucial for adapting models to new domains when there is insufficient data. Given the lack of generalizability, current DNNs are not able to incrementally learn and improve when deployed in real-life situations and transfer knowledge from one domain to another (multi-domain) [28].

In the real world, DNNs are vulnerable to adversarial attacks and can be deceived easily. In adversarial cases, minor perturbations will lead to misclassifications with high confidence. Adversarial attacks have been investigated for different vision tasks, such as image classification, object detection, and semantic segmentation. For example, it is possible to change the red traffic light to green for AV [35], make people invisible to AI [32] using small crafted adversarial patches held in front of the body or make the AV to misinterpret a stop sign as a speed limit sign [16].

Researchers have suggested a few solutions, such as transfer learning for transferring knowledge to another domain, to address challenges associated with DNN classifiers. However, the proposed solutions partially solve the problems and have many limitations. For example, the transfer learning approach faces a significant challenge regarding data sharing and several legal issues such as privacy and property law [21].

To strengthen the safety of autonomous driving, this paper proposes an explainable ILP-based solution focusing on traffic sign detection. The proposed method mimics human perception to recognise traffic signs by detecting high-level features, including signs' geometric shapes, colours and contents, that differentiate them from other signs. While DNNs only use low-level (pixel-level) features that can be easily misled [16] and need a large amount of data, this traffic sign detector only needs a handful of training images and is fully robust against adversarial attacks.

Several studies have investigated the application of Inductive Logic Programming (ILP) in image recognition tasks. ILP has been employed in Logical Vision [12, 11], incorporating the abductive perception technique [29] to extract high-level interpretation of objects such as classical 2D shapes by utilising low-level primitives, such as high contrast points. ILP has also been used for 3D scene analysis [17] with 3D point cloud data. However, to our knowledge, a traffic sign detection based on the ILP has not been proposed previously for traffic sign classification. Therefore, our approach is a novel contribution to this context.

The paper is structured as follows. Section 2 surveys some successful adversarial examples in AVs. Section 3 describes the framework for robust traffic sign detection using ILP. Section 4 details experiments. In this section, the Aleph-based approach is compared with the Metagol-based approach. Metagol can learn hypotheses with only one positive and one negative example, while Aleph needs at least eight positive and negative examples to have the same accuracy as Metagol. Also, the ILP-based system is compared with the DNN-based classifier on adversarial examples. The results show that the ILP-based approach is considerably more resilient to adversarial attacks. Finally, Section 5 summarises the outcomes and discusses further work.

2 Adversarial Attacks on AVs' Perception

In this section, we survey a sample of successful adversarial attacks in autonomous driving that easily deceived DNN-based vision classifiers. An adversarial attack aims to generate adversarial examples as the input for machine learning systems. However, adversarial examples are only negligibly modified from the real examples; they lead to misclassification [19].

When the fragility of deep neural networks to specific input perturbations was discovered for the first time, it was shown that an adversarial attack could turn a bus into an ostrich for an AI system [31]. Another algorithm named Show-and-Fool [8] was introduced to evaluate the robustness of an image captioning system. This method attained a 95.8% attack success rate for adversarial examples via applying a minor perturbation on image pixels which are invisible to humans, turning a stop sign into a teddy bear for the AI system.

The authors of [20] devised a method whereby semantic image segmentation could be attacked using adversarial perturbation to blend out (vanish) a desired target. They showed the existence of universal noise, which removes a target class (e.g. all pedestrians) from the segmentation while leaving it mostly unchanged otherwise. The robustness of the popular DNN-based semantic segmentation models evaluated against adversarial attacks on urban scene segmentation [2]. The results showed that the segmentation performances of all models seriously dropped after the attacks.

Later it was shown that adversarial examples could be misclassified by deep learning systems in real life [22]. Previous works have threatened the model by feeding machine learning classifiers directly, which is not always possible in the real world.

Another paper [16] proposed the Robust Physical Perturbations (RP2) technique to fool a Convolutional Neural Network (CNN) based road sign classifier in the physical world under various distances and viewpoints using different robust visual adversarial perturbations. This approach caused targeted misclassification, which changed a stop sign into a speed limit sign for the AI system. They also proposed a disappearance attack, causing a stop sign hidden from state-of-art object detectors like Mask R-CNN and YOLO [15]. An Adversarial Camouflage (AdvCam) approach [14] generated adversarial photos to fool a DNN classifier at various detecting angles and distances. With a few stains invisible to humans, this technique can cause the classifier to misclassify the objects, such as misidentifying a stop sign as a "barber shop" with .82% confidence.

Fig. 1 illustrates targeted stop signs with successful physical-world attacking approaches named RP2 and AdvCam, misleading the state-of-the-art DNN classifiers.

An Adaptive Square Attack (ASA) method [24] has been suggested that can attack the black box by generating invisible perturbation for traffic sign images, successfully leading to sign misclassification. Five adversarial attacks and four defence methods have been investigated on three driving models adopted in modern AVs [13]. They demonstrated that while these defence methods can effectively defend against a variety of attacks, none can provide adequate protection against all five attacks.

One recent work proposed three sticker application methods, namely RSA, SSA and MCSA, that can deceive the traffic sign recognition DNNs with realistic-looking stickers [4]. Another attack included painting the road, which targeted deep neural network models for end-to-end autonomous driving control [5]. Another work demonstrated a successful physical adversarial attack on a commercial classification system to deceive an AV's sign classifier[25].

Figure 1: Targeted physical perturbation by a) AdvCam and b) RP_2 misleading DNN classifiers, SL45 is speed limit 45 sign.

BadNets algorithm [18] was implemented to deceive a complex traffic sign detection system leading to maliciously misclassifying stop signs as speed-limit signs on real-world images.

These adversarial attacks on the deep-learning models pose a significant security threat to autonomous driving.

3 Robust Traffic Sign Detection Using ILP

Inductive Logic Programming (ILP) is a machine learning method which uses logic-based representation and inference. Depending on the type of logical inference and the search algorithm, there are different ILP systems, such as Aleph [3] and Metagol [10], that used in this paper.

Due to a logic-based representation and inference, ILP has the potential for human-like abstraction and reasoning. These logic-based AI approaches have the ability to learn unknown complex tasks with only a few examples. It complements deep learning because logic programs are interpretable and data-efficient, leading them towards a strong generalisation. Moreover, these rule-based approaches, which are explicitly symbolic, are sometimes considered safer than neural approaches [1].

ILP aims to learn a hypothesis (rule) using a few positive and negative examples and Background Knowledge (BK); this induced rule, alongside BK, should cover as many positive and as few negative examples as possible [26]. For inducing the rules, BK should include all essential predicates to represent the relevant information.

One of the advantages of ILP is its ability to use BK, including facts and rules in the form of logical expressions, which could be related. In ILP, choosing appropriate BK based on well-selected features is essential to obtaining good results [9]. Moreover, using BK makes ILP incremental. For example, suppose we want to learn animal signs in traffic sign detection, choosing sign "a" contains an animal(*contains(a, animal)* as a BK, which holds when traffic sign "a" has an animal symbol. Then we provide BK with various different animal shaped symbols (deer, cow, ...). In that case, if we see a new animal sign that doesn't exist in our BK, we can add it to our BK without relearning, and there is no need to change the hypothesis. This feature makes it possible to have real-time interaction with drivers towards customised autonomous driving.

Our proposed ILP-based stop sign detection system is demonstrated in Fig. 2. The first step is pre-processing all the images, including training and test images, and turning them into a symbolic representation to provide BK. In the pre-processing phase, high-level features of traffic sign images, including colour, shape, text and digits, are extracted and represented as a set of logical facts for the next step. For feature extraction, computer vision tools such as OpenCV can extract high-level features using low-level features such as pixel colours or colour gradients.

In the next step, a set of positive and negative training examples (E) and a set of logical facts as BK extracted from the previous step will be provided to the ILP system. The system aims to learn a hypothesis H such that $B, H \models E$ where $\models$ is logical entailment.

We use Aleph and Metagol as the ILP system to induce the rule for stop sign detection.

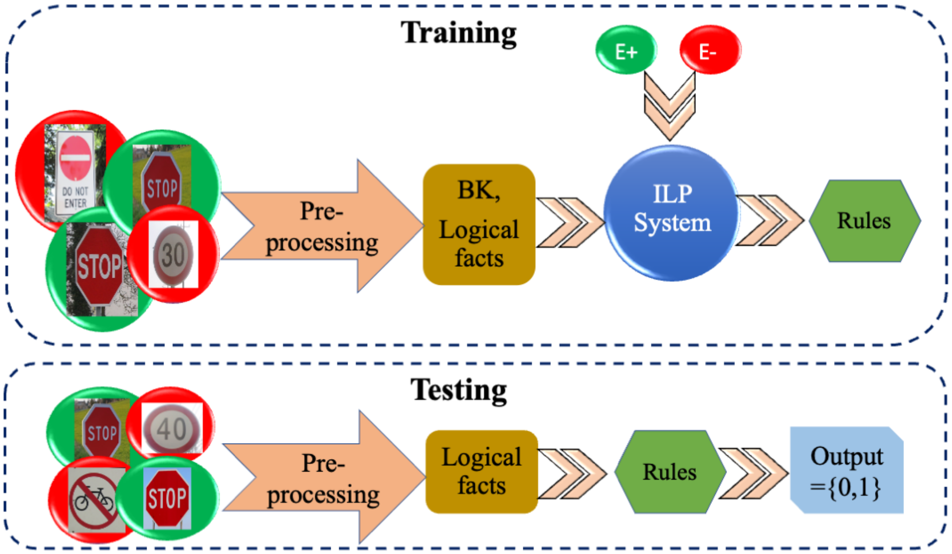


Figure 2: ILP- based traffic sign classifier

We used Aleph5 as the ILP system in one of our experiments; it is an old ILP system developed in Prolog and based on inverse entailment. Aleph's algorithm resolves the relationship between the determination predicate and the determining predicate to generate a general theory.

Metagol is employed in our other experiment. It is an ILP system based on Meta Interpretive Learning (MIL) [27] implemented in Prolog. By instantiating metarules, MIL learns logic programs from examples and BK. In addition, MIL not only learns the recursive definition and fetches higher-order meta-rules but also supports predicate invention.

4 Experimental Evaluation

This experiment aims to learn *"traffic_sign"*, which is the target predicate. For simplicity, only the stop sign is investigated; other traffic signs can be included to have a complete traffic sign classifier. We provide Aleph and Metagol with the same BK. The Aleph mode declarations are illustrated in Table 1, and the Metagol-based system is supplied with the metarules demonstrated in Table 2, uppercase letters represent predicate symbols (second-order variables), and lowercase letters represent variables.

Table 1: Mode declarations for Aleph experiments.

$: -modeh(1, traffic\_sign(+sign, \#class).$
$: -modeb(*, colour(+sign, \#colour)).$
$: -modeb(*, shape(+sign, \#shape)).$
$: -modeb(*, word(+sign, -w)).$
$: -modeb(*, closely\_match(+w, \#word)).$
$: -modeb(*, number(+sign, -n)).$
$: -modeb(*, digits(+n, \#int)).$

In Aleph mode declaration, *"modeh"* indicates that the predicate should appear in the head of the hypothesis, and *"modeb"* indicates that it should be in the body of the induced hypothesis. According to Table 1, six predicates can be used in the body of the induced hypothesis which. The meaning of each predicate is defined as follows:

- *traffic_sign(a, #class)*, which holds when the sign "a" belongs to a specific category of traffic sign determined by #class (e.g. a stop sign).
- *colour(a, #colour)*, which holds when a certain #colour(e.g. red) exists in the sign "a".
- *shape(a, #shape)*, which holds when the shape of sign "a" is a specific shape determined by # shape(e.g. circle).
- *has_word(a, a_w1)*, which holds when the sign "a" has the word a_w1 on it.
- *closely_match(a_w1, w)*, which holds when the word "a_w1" closely matches the word "w" (e.g. stop).
- *number(a, a_n1)*, which holds when the sign "a" has the number "a_n1".
- *digits(a_n1,n)*, which holds when the number "a_n1" includes "n" (e.g. 60)

Table 2: Employed metarules in Metagol experiment.

Name	Metarule
Identify	$P(x,y) \longleftarrow Q(x,y)$
Inverse	$P(x,y) \longleftarrow Q(y,x)$
Precon	$P(x, y) \longleftarrow Q(x), R(x, y)$
Postcon	$P(x, y) \longleftarrow Q(x, y), R(y)$
Chain	$P(x, y) \longleftarrow Q(x, z), R(z, y)$
Recursion	$P(x, y) \longleftarrow Q(x, z), P(z, y)$

Table 3: Extracted features for a positive (p1) and negative (n1) examples.

Pos example(p1)	Neg example(n1)
$color(p1, red).$	$color(n1, red).$
$color(p1, white).$	$color(n1, white).$
$shape(p1, octagon).$	$shape(n1, Circle).$
$has\_word(p1, p1\_w1).$	$number(n1, n1\_d1).$
$closely\_match(p1\_w1, stop).$	$digits(n1\_d1, 30).$

To further explain the process of converting images into a set of logical facts, we take one positive example, a stop sign named p1, and one negative example, a speed limit sign named n1. In the pre-processing stage, the high-level features of these traffic signs were extracted to be included in the BK. The details of these features and their corresponding logical representation are presented in Table 3.

These logical facts, together with the names of the positive and negative examples (such as sign(p1) as a positive and sign(n1) as a negative example), will enable the ILP system to induce a hypothesis (logical rule). Finally, the ILP system recognises the new traffic signs using this induced rule.

4.1 Material and Method

Base data set. Our base data set includes traffic sign images without any adversarial perturbation. They have been downloaded from Wikimedia Commons as no-restriction images. Positive examples contain stop sign images, and negative examples include other traffic signs excluding stop sign images. Normal positive and negative examples are shown in Fig. 3.

Adversarial test data set. To evaluate the robustness of the ILP stop sign detector against adversarial attacks, we used the targetted traffic signs attacked by RP_2 and AdvCam.

RP_2 is a general attack algorithm for misleading standard-architecture road sign classifiers. It generates visual adversarial perturbations, such as black and white stickers attached to a traffic sign to mislead the classifier. The RP_2 data set contains three types of perturbation, stop signs perturbed by subtle, camouflage graffiti and camouflage art attacks viewed from different angles.

AdvCam is an approach for creating camouflage physical adversarial images to fool state-of-the-art DNN-based image classifiers. This approach can make the classifier identify a stop sign as a "barber shop" with high confidence. This paper will utilise targetted stop signs with Advcam with different stain styles to evaluate the proposed ILP sign classifier.

The feature recognition element should extract high-level features, including the traffic sign's border shape, colour and text; symbol extraction can be added in future; *OpenCV* is employed for this purpose. First, the image background is removed utilising *rembg* and then pre-processed by *cv2.bilateralFilter* for noise reduction.

Colour masks were then employed using cv2.inRange for colour detection, and small areas were ignored. After that, by applying morphological operations, colour masks were post-processed.

For text and digit detection, *EasyOCR* is utilised; if the detected item is a word, it will be investigated if the detected word closely matches some common words in traffic signs; for example, it should have at least three letters in common with the word STOP to recognise as a stop word.

For shape detection, *cv2.findContours* is applied on detected colour masks, and *cv2.approxPolyDP* is utilised for polygon detection.

a) Negative Examples

b) Positive Examples

Figure 3: Base data set for training and testing

Convolutional Neural Network. To compare the results, a well-known CNN classifier [34] is utilised that is trained on the German Traffic Sign Recognition Benchmark (GTSRB) [30]. The evaluation of this architecture achieved 97.6% accuracy on the GTSRB test data set.

The base data set is utilised for training the ILP systems (Aleph and Metagol). First, we randomly select an equal number of positive and negative examples in each run, so the default accuracy is 50% for this training data set. Next, the ILP-based systems try to find a hypothesis that covers as many positive and as few negative examples as possible. Then the remaining examples in the data set are used as a test data set for evaluation to determine the accuracy. This process is repeated one hundred times, and average accuracy is calculated for each certain number of positive and negative examples in the training set. Therefore we have a fair comparison between Aleph and Metagol regarding the size of the required training data set.

The data and the code used in this experiment are available on GitHub [6].

4.2 Results and Discussion

Fig. 4 illustrates the average accuracy of Aleph and Metagol-based ILP systems with increasing training examples. According to this figure, Metagol can find a hypothesis with 100% accuracy on the test data set including only one positive and one negative example. In comparison, Aleph starts learning with at least two positive and two negative examples with around 65% accuracy. Aleph can reach the same level of accuracy as Metagol by learning from eight positive and negative examples. According to these results, Metagal is more data-efficient than Aleph. In this figure, the orange curve shows the default accuracy, which is equal to 50% because the number of negative and positive examples are equal in each run.

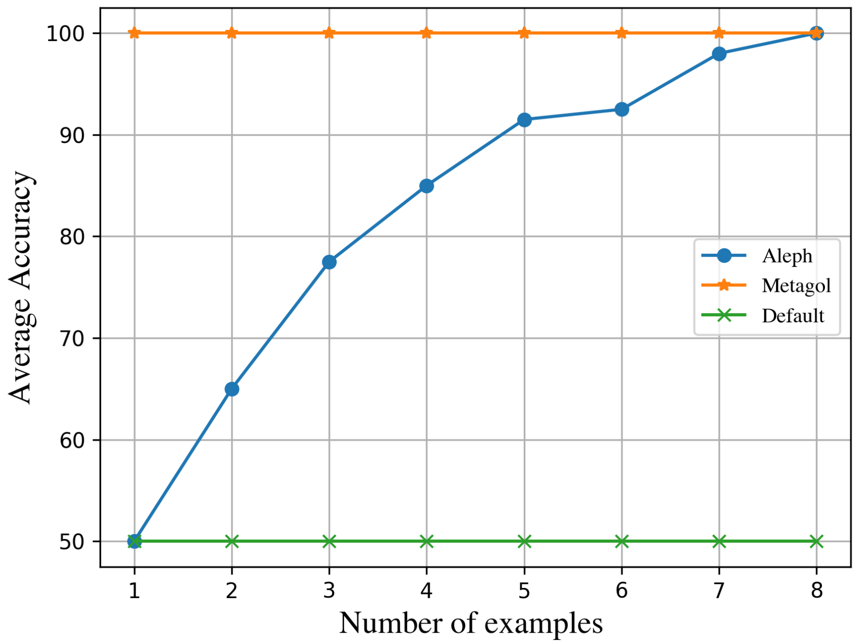


Figure 4: Average accuracy of Aleph vs Metagol with an increasing number of training examples from the base data set (equal positive and negative sets)

The hypothesis (a logic program) induced by Metagol with only one set of positive and negative examples is the same as the learned rule by Aleph with eight positive and negative examples. It is entirely accurate on the base test data set and is shown below:

```
traffic_sign(A, stop_sign):-
        has_word(A,A_w1),
        closely_match(A_w1, stop).
```

This learned rule is completely explainable and matches human interpretation. The rule says the traffic sign *"A"* is a stop sign when the two literals *has_word(A, A_w1)* and *closely_match(A_w1, stop)* hold, i.e. if the sign contains a word and that word closely matches stop, that sign would be predicted a stop sign.

The performance of this hypothesis is evaluated on the base data set and attack data sets, including RP_2 (subtle, camouflage graffiti and camouflage art attacks) and AdvCam with different stains. The accuracy of this rule on all test data sets is 100%. While the ILP-based sign detector shows a perfect performance, the DNN-based classifier shows abysmal performance facing manipulated images.

Table 4 compares the results of the DNN-based classifier and the ILP-based classifier on different data sets. The DNN-based classifier is trained on the GTSRB data set, which contains more than 50,000 images. The Aleph-based classifier is trained on eight positive and negative examples, while the Metagol approach is trained on only one negative and one positive example. It shows that while ILP-based systems can learn from small amounts of data, they are more resilient to noise and adversarial attacks.

data set		DNN-based	ILP-based
Base		100%	100%
RP_2	subtle	0	100%
	camouflage graffiti	0%	100%
	camouflage art attacks	6.6%	100%
AdvCam		66.6%	100%

Table 4: Comparing the results of the hypothesis induced by the proposed ILP-based approach (Aleph and Metagol) on different test data sets with a DNN classifier.

5 Conclusions

DNN-based traffic sign classifiers need a large amount of data for training, and it has been shown that they are vulnerable to adversarial attacks or natural noise. They are also not explainable; consequently, there is no way to debug them. While DNN-based classifiers suffer from these problems, we propose an ILP-based approach for traffic sign detection in autonomous vehicles to address these issues.

Our proposed technique mimics humans in traffic sign detection and uses high-level features of a sign, such as colour and shape, for detection. Therefore this method is data efficient, explainable and able to withstand adversarial attacks that cannot easily deceive humans.

The results indicate that our approach with only a handful of training data can induce logical rules easily understandable by humans. Furthermore, it significantly outperforms the deep learning approach regarding adversarial attacks. It shows a 100% accuracy on the data set targetted with RP_2 and AdvCam attacking approaches, while a DNN-based classifier performs poorly on these data sets.

For future works, we suggest employing DNN for high-level feature extraction (shapes or symbols) in traffic signs. Integrate Machine Learning and Logic programming in AV applications will use both the strengths of machine learning and symbolic AI (knowledge and reasoning) to address the AI obstacles.

Acknowledgments

The first author would like to acknowledge her PhD grant funding from the Breaking Barriers Studentship Award at the University of Surrey. Also, we would like to acknowledge the support of Dany Varghese with Aleph PyILP [33].

References

[1] Greg Anderson, Abhinav Verma, Isil Dillig & Swarat Chaudhuri (2020): *Neurosymbolic reinforcement learning with formally verified exploration*. *Advances in neural information processing systems* 33, pp. 6172–6183.

[2] Anurag Arnab, Ondrej Miksik & Philip HS Torr (2018): *On the robustness of semantic segmentation models to adversarial attacks*. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 888–897.

[3] Ashwin Srinivasan (2001): *The aleph manual*. `https://www.cs.ox.ac.uk/activities/programinduction/Aleph/aleph.html`.

[4] Yasin Bayzidi, Alen Smajic, Fabian Hüger, Ruby Moritz, Serin Varghese, Peter Schlicht & Alois Knoll (2022): *Traffic sign classifiers under physical world realistic sticker occlusions: A cross analysis study*. In: *2022 IEEE Intelligent Vehicles Symposium (IV)*, IEEE, pp. 644–650, doi:10.1109/CVPR.2017.634.

[5] Adith Boloor, Xin He, Christopher Gill, Yevgeniy Vorobeychik & Xuan Zhang (2019): *Simple physical adversarial examples against end-to-end autonomous driving models*. In: *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, IEEE, pp. 1–7, doi:10.1145/1081870.1081950.

[6] Zahra Chaghazardi (2022): *Traffic Sign Detection using ILP*. https://github.com/Chaghazardi/Traffic-Sign-Detection-using-ILP. Available at `https://github.com/Chaghazardi/Traffic-Sign-Detection-using-ILP`.

[7] Zahra Chaghazardi, Saber Fallah & Alireza Tamaddoni-Nezhad (2023): *A Logic-based Compositional Generalisation Approach for Robust Traffic Sign Detection*. In: *International Joint Conference on Artificial Intelligence 2023 Workshop on Knowledge-Based Compositional Generalization*.

[8] Hongge Chen, Huan Zhang, Pin-Yu Chen, Jinfeng Yi & Cho-Jui Hsieh (2017): *Attacking visual language grounding with adversarial examples: A case study on neural image captioning*. *arXiv preprint arXiv:1712.02051*.

[9] Andrew Cropper, Sebastijan Dumancic & Stephen H Muggleton (2020): *Turning 30: New Ideas in Inductive Logic Programming*. In: *IJCAI*, doi:10.24963/ijcai.2020/673.

[10] Andrew Cropper & Stephen H. Muggleton (2016): *Metagol System*. https://github.com/metagol/metagol. Available at `https://github.com/metagol/metagol`.

[11] Wang-Zhou Dai, Stephen Muggleton, Jing Wen, Alireza Tamaddoni-Nezhad & Zhi-Hua Zhou (2018): *Logical vision: One-shot meta-interpretive learning from real images*. In: *Inductive Logic Programming: 27th International Conference, ILP 2017, Orléans, France, September 4-6, 2017, Revised Selected Papers 27*, Springer, pp. 46–62, doi:10.1016/j.cviu.2007.08.003.

[12] Wang-Zhou Dai, Stephen H Muggleton & Zhi-Hua Zhou (2015): *Logical Vision: Meta-Interpretive Learning for Simple Geometrical Concepts*. In: *ILP (Late Breaking Papers)*, pp. 1–16.

[13] Yao Deng, Xi Zheng, Tianyi Zhang, Chen Chen, Guannan Lou & Miryung Kim (2020): *An analysis of adversarial attacks and defenses on autonomous driving models*. In: *2020 IEEE international conference on pervasive computing and communications (PerCom)*, IEEE, pp. 1–10, doi:10.1109/PerCom45495.2020.9127389.

[14] Ranjie Duan, Xingjun Ma, Yisen Wang, James Bailey, A Kai Qin & Yun Yang (2020): *Adversarial camouflage: Hiding physical-world attacks with natural styles*. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 1000–1008.

[15] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Florian Tramer, Atul Prakash, Tadayoshi Kohno & Dawn Song (2018): *Physical adversarial examples for object detectors*. *arXiv preprint arXiv:1807.07769* 1(3), p. 4.

[16] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno & Dawn Song (2018): *Robust physical-world attacks on deep learning visual classification*. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1625–1634.

[17] Reza Farid & Claude Sammut (2014): *Plane-based object categorisation using relational learning*. *Machine Learning* 94, pp. 3–23, doi:10.1007/s10994-013-5352-9.

[18] Tianyu Gu, Kang Liu, Brendan Dolan-Gavitt & Siddharth Garg (2019): *Badnets: Evaluating backdooring attacks on deep neural networks*. *IEEE Access* 7, pp. 47230–47244, doi:10.1109/TKDE.2009.191.

[19] Jie Gui, Zhenan Sun, Yonggang Wen, Dacheng Tao & Jieping Ye (2021): *A review on generative adversarial networks: Algorithms, theory, and applications*. *IEEE transactions on knowledge and data engineering*.

[20] Jan Hendrik Metzen, Mummadi Chaithanya Kumar, Thomas Brox & Volker Fischer (2017): *Universal adversarial perturbations against semantic image segmentation*. In: *Proceedings of the IEEE international conference on computer vision*, pp. 2755–2764.

[21] Mauritz Kop (2020): *Machine learning & EU data sharing practices*. In: *TTLF Newsletter on Transatlantic Antitrust and IPR Developments*, Stanford-Vienna Transatlantic Technology Law Forum, Transatlantic Antitrust . . . .

[22] Alexey Kurakin, Ian J Goodfellow & Samy Bengio (2018): *Adversarial examples in the physical world*. In: *Artificial intelligence safety and security*, Chapman and Hall/CRC, pp. 99–112, doi:10.1201/9781351251389-8.

[23] T. B. Lee (2018): *Software bug led to death in ubers self-driving crash*. Available at `https://arstechnica.com/tech-policy/2018/05/report-software-bug-led-to-death-in-ubers-self-driving-crash/`.

[24] Yujie Li, Xing Xu, Jinhui Xiao, Siyuan Li & Heng Tao Shen (2020): *Adaptive square attack: Fooling autonomous cars with adversarial traffic signs*. *IEEE Internet of Things Journal* 8(8), pp. 6337–6347, doi:10.1109/CVPR.2018.00957.

[25] Nir Morgulis, Alexander Kreines, Shachar Mendelowitz & Yuval Weisglass (2019): *Fooling a real car with adversarial traffic signs*. Available at `https://arxiv.org/abs/1907.00374`.

[26] Stephen Muggleton (1991): *Inductive logic programming*. *New generation computing* 8(4), pp. 295–318, doi:10.1007/BF03037089.

[27] Stephen H Muggleton, Dianhuan Lin & Alireza Tamaddoni-Nezhad (2015): *Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited*. *Machine Learning* 100(1), pp. 49–73, doi:10.1007/s10994-014-5471-y.

[28] Bukola Salami, Keijo Haataja & Pekka Toivanen (2021): *State-of-the-Art Techniques in Artificial Intelligence for Continual Learning: A Review*. *FedCSIS (Position Papers)*, pp. 23–32.

[29] Murray Shanahan (2005): *Perception as abduction: Turning sensor data into meaningful representation*. *Cognitive science* 29(1), pp. 103–134, doi:10.1207/s15516709cog2901_5.

[30] Johannes Stallkamp, Marc Schlipsing, Jan Salmen & Christian Igel (2012): *Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition*. *Neural networks* 32, pp. 323–332, doi:10.1016/j.neunet.2012.02.016.

[31] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow & Rob Fergus (2013): *Intriguing properties of neural networks*. *arXiv preprint arXiv:1312.6199*.

[32] Simen Thys, Wiebe Van Ranst & Toon Goedemé (2019): *Fooling automated surveillance cameras: adversarial patches to attack person detection*. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, pp. 0–0.

[33] Dany Varghese (2022): *PyILP*. https://github.com/danyvarghese/PyILP. Available at `https://github.com/danyvarghese/PyILP`.

[34] Vivek Yadav (2016): *German sign classification using deep learning neural networks*. `https://github.com/vxy10/p2-TrafficSigns`.

[35] Chen Yan, Zhijian Xu, Zhanyuan Yin, Xiaoyu Ji & Wenyuan Xu (2022): *Rolling Colors: Adversarial Laser Exploits against Traffic Light Recognition*. *arXiv preprint arXiv:2204.02675*.

Quantification and Aggregation over Concepts of the Ontology

Pierre Carbonnelle Matthias Van der Hallen Marc Denecker
KU Leuven
Leuven, Belgium*
pierre.carbonnelle@kuleuven.be matthias.vanderhallen@gmail.com marc.denecker@kuleuven.be

We argue that in some KR applications, we want to quantify over sets of *concepts* formally represented by symbols in the vocabulary. We show that this quantification should be distinguished from second-order quantification and meta-programming quantification. We also investigate the relationship with concepts in intensional logic.

We present an extension of first-order logic to support such abstractions, and show that it allows writing expressions of knowledge that are elaboration tolerant. To avoid nonsensical sentences in this formalism, we refine the concept of well-formed sentences, and propose a method to verify well-formedness with a complexity that is linear with the number of tokens in the formula.

We have extended FO(·), a Knowledge Representation language, and IDP-Z3, a reasoning engine for FO(·), accordingly. We show that this extension was essential in accurately modelling various problem domains in an elaboration-tolerant way, i.e., without reification.

1 Introduction

The power of a KR language for compact expression of knowledge lies for an important extent in the way it allows us to abstract over certain types of objects, e.g., to quantify, count, or sum over them. Classical first-order logic (FO) is a much stronger KR language than Propositional Calculus (PC) because it allows to quantify over domain objects. Still, the abstraction power of FO is limited, e.g., it does not allow counting or sum over domain objects satisfying some condition. Many first-order modeling languages are therefore extended to support *aggregations*. Likewise, one can only quantify over individuals in the domain, not over relations and functions. This is resolved in second-order logic (SO).

In this paper, we argue that in some KR applications, we want to abstract (i.e., quantify, count, sum, ...) not over relations and functions, but over certain *concepts* in the problem domain, as the following example will show.

Example 1 *Consider a corona testing protocol. A person is to be tested if she shows at least two of the following symptoms: fever, coughing and sneezing. The set of persons showing a symptom is represented by a unary predicate ranging over persons, i.e., $hasFever/1$, $coughs/1$, and $sneezes/1$. Leaving first-order logic behind us, assume that the set of symptoms is represented by the predicate $symptom/1$.*

Testing is expressed as a unary predicate test1/1, ranging over persons. Informally, this predicate is to be defined along the following lines (while freely extending the syntax of FO):

$$\forall p(test1(p) \Leftrightarrow 2 \leq \#\{x|symptom(x) \wedge x(p)\}) \tag{1}$$

---

[*] This research received funding from the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" programme. The authors thank Maurice Bruynooghe for his review of the drafts of this paper.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 213–226, doi:10.4204/EPTCS.385.22

The question is: what is the nature of the variable x and the predicate *symptom*? Over what sorts of values does x range?

At first sight, one might think that x is a second-order variable ranging over the set of sets of persons, and that we want to count the sets of persons having one of the symptoms and containing person p. A 'possible world' analysis refutes this idea. Consider a state of affairs where everybody has all 3 symptoms. Formally, in a structure $\mathscr{I}$ abstracting such a state, the interpretation of each symptom predicate is identical: it is the set of all persons, $hasFever^{\mathscr{I}} = coughs^{\mathscr{I}} = sneezes^{\mathscr{I}} = \{y|person(y)\}$. Thus, there is only one set that contains person p: the cardinality in Eq. (1) would be one, and the condition for the first test would not be satisfied for any p, even though everyone should be tested. A different idea is that x is ranging over *symbols*, and that we want to count the number of symptom symbols whose interpretation contains person p. This sort of variable is generally found in meta programming, of the kind investigated in, e.g., Logic Programming [4] or Hilog [10]. While this view has merits, it is not totally satisfying. Assume, for insight, that the team working on this application is international, and the French group has introduced the predicate $estFievreux/1$, French for "has fever". Thus, *hasFever*/1 and *estFievreux*/1 are *synonymous*. Consider *John* whose only symptom is fever. Both $isFievreux(John)$ and $hasFever(John)$ are true. The number of symptom symbols whose interpretation contains *John* would be two, leading to the erroneous decision that *John* needs to be tested.

The view that we elaborate in this paper, is that the abstraction in this example is over *concepts*. For context, consider that the first phase of a rigorous approach to building a KR specification is the selection of a formal ontology Σ of symbols representing relevant *concepts* of the application field. These concepts can be identified with the user's *informal interpretations* of the symbols in the ontology of the domain. The informal interpretation is in practice a crucial concept in all knowledge representation applications: it is the basis of all actions of formal knowledge representation, and of all acts of interpreting formal results of computation in the application domain. Connecting concepts to informal interpretations provides a good intuitive explanation of what concepts are and how they are relevant in a KR context.

In classical first-order logic, a structure only has the extension of symbols. The extension of symbol σ in a structure is its *formal* interpretation, a.k.a value, in that structure. It is a relation or function, of a precise arity, over the universe of discourse; in formulae, it is denoted simply by σ.

For our purpose, we extend a structure to also have the intension of symbols, i.e., an object in the domain of discourse representing its *informal* interpretation. This intension is *rigid* in the sense that it is the same in every extended structure. Since the informal meaning of symbols is known, the synonym relation $\sim_s$ is known too, and it is an equivalence relation on Σ. Synonymous symbols have the same arity and type. Synonymous symbols σ_1, σ_2 have the same intension, and, in each structure $\mathscr{I}$, the same extension. In formulae, the intension of σ is denoted by $`\sigma$. Finally, we introduce the *value functor* $\$(\cdot)$: when applied to an intension, it returns its formal interpretation in the current structure.

In the example, the intension of the symbols $hasFever/1$ and $estFievreux/1$ are the same object denoted by $`hasFever$ (or $`estFievreux$) which is the abstraction of the concept of people having fever. The symbol *symptom* denotes the set containing the intensions $`hasFever$, $`coughs$ and $`sneezes$. In any particular structure, the $hasFever/1$ symbol has an interpretation which is a set of elements abstracting the set of people having fever in the state of affairs. This set is also the value of the $`hasFever$ intension, i.e., of $\$(`hasFever)$. Suppose we want to count the concepts in *symptom* whose value (a set) contains person p. The following formula expresses the testing protocol:

$$\forall p(test1(p) \Leftrightarrow 2 \leq \#\{x|symptom(x) \land \$(x)(p)\}) \tag{2}$$

In the current state of the art of Knowledge Representation, one might consider, as we once did, applying the common KR technique of *reification*, rather than laboriously extending KR languages as we do. In this particular case, one could introduce object symbols $Fever, Sneezing, Coughing$ (under the Unique Name Assumption), and a predicate, $Has/2$. We can now represent that Bob has fever by the atom $Has(Bob, Fever)$. The testing protocols can then be formalized (in a proper extension of FO) as:

$$\forall x(symptom(x) \Leftrightarrow x = Fever \lor x = Sneezing \lor x = Coughing) \tag{3}$$

$$\forall p(test1(x) \Leftrightarrow 2 \leq \#\{x|symptom(x) \land Has(p,x)\}) \tag{4}$$

Still, we believe that our study is valuable for two reasons. First, we improve the scientific understanding of the problem, thus helping avoid using KR encoding tricks to circumvent it. Second, our logic is more *elaboration tolerant*. Per [26], "a formalism is elaboration tolerant to the extent that it is convenient to modify a set of facts expressed in the formalism to take into account new phenomena or changed circumstances". Because it helps reduce development costs, elaboration tolerance is a desirable feature of formalisms used in AI applications, and in particular in knowledge representation. Consider a knowledge base that, initially, does not contain rule (1). Per standard KR practice, it would use predicates such as $hasFever/1$ to encode symptoms. Then, circumstances change, and rule (1) must be introduced. A formalism that requires rewriting the knowledge base using reified symptoms $Has/2$ is not elaboration tolerant. Our formalism, by contrast, would not require any change to the initial knowledge base.

To evaluate our approach, we have extended FO(·) to allow quantification and aggregation over concepts, and extended IDP-Z3, a reasoning engine for FO(·), accordingly. It proved essential to accurately model the knowledge of various problem domains in an elaboration tolerant way, i.e., without reification.

To summarize, the main contributions of the paper are as follows. We argue that in some KR applications, it is useful to abstract over concepts in quantification and aggregations. We analyze the connection to well-known intensional logic (Section 2), and explain how to distinguish this from second-order quantification and meta-programming (Section 6). We explain how to extend a simple predicate logic so that every ontology symbol has an intensional and an extensional component (Section 3). We show that this may lead to nonsensical sentences, and we restrict our syntax to avoid them. In Section 4, we extend FO(·), a model-based knowledge representation language, and IDP-Z3, a reasoning engine for FO(·), as we propose, and illustrate the benefits of this extension in different applications (Section 5).

2 "Concepts" in Logic

The notions that we study here, i.e., the notions of a "concept" and of its "value" in a particular state of affairs, are closely related, if not identical, to the broadly studied distinction between *intension* and *extension*, *meaning* and *designation*, or *sense* and *reference* as studied in philosophical and intensional logic. The close relation between the two urges us to detail the role of concepts or "intensions" in intensional logic, and to make the comparison with our approach.

In philosophical logic, the concepts of *intension* and *extension* have been investigated at least since Frege (for a historical overview, see [22]). A prototypical example concerns the "morning star" and "evening star", i.e., the star visible in the east around sunrise, and in the west around sunset. These words have two different intensions (i.e., they refer to two different concepts), but as it happens, in the current state of affairs, their extensions (i.e., their interpretation) are the same: the planet also known as Venus.

Early key contributors in this study are Frege, Church, Carnap and Montague. The intensional logics of Montague, of Tichý, and of Gallin, are typed modal logics based on *Kripke semantics* in which expressions at any level of the type hierarchy have associated intensions. They provide notations to access the intension and the extension of expressions. They also provide modal operators to talk about the different values that expressions may have in the *current* and in *accessible* worlds.

The principles of intensions of higher-order objects are similar to those of the base level (i.e., domain elements), leading Fitting in several papers [21, 22] to develop a simplified logic, called FOIL, where only expressions at the base level of the type hierarchy have intensions.

In intensional logic, the intension of an expression is modelled as a mapping from possible worlds to the interpretation of the expression. For example, the intension of *EveningStar* is the mapping from possible worlds to the extension of *EveningStar* in that world. We can express that the *extension* of *MorningStar* in the current world, equals the *extension* of *EveningStar* in a (possibly different) *accessible* world:

$$[\lambda x(\diamond(EveningStar = x))](MorningStar) \tag{5}$$

Here the lambda expression binds variable x to the extension of *MorningStar* in the current world, and the "$\diamond$" modal operator is used to indicate an accessible world. This statement would be true, e.g., in the view of a scientist that may not have evidence that both are the same but *accepts it as a possibility*.

A comparison between the work presented in this paper and the intensional logics in philosophical logic, is not easy. Even if the notion of intension versus extension is underlying the problems that we study here, the focus in our study is on very different aspects of intensions than in philosophical logic. As a result, the logic developed here differs strongly from intensional logics.

An important difference is the lack of modal logic machinery in our logic to analyze the difference between intensions and extensions. In our logic, a symbol has an extension (a.k.a interpretation) in different possible worlds, but there are no modal operators to "talk" about the extension in other worlds than the current one.

Another important difference is that we provide an abstraction mechanism through which we can quantify over, and count, intensional objects (a.k.a concepts). In KR applications, problems such as the symptom example easily occur (see Section 5). They can be analyzed and demonstrated and solved in a logic much closer to standard logic.

Compared to FOIL, we associate intensions to predicate and function symbols of any arity even though, as hinted earlier, it poses additional challenges to ensure the syntactical correctness of formulae. We will address these by introducing guards. By contrast, FOIL associates intensions only to intensional objects of arity 0, like *Morningstar*, avoiding the issue. Finally, our research also includes the extension of an existing reasoning engine for the proposed logic, capable of reasoning with aggregates over concepts such as in Eq.(3).

3 FO(Concept)

Below, we describe how first-order logic, in its simplest form, can be extended to support quantification over concepts. The language, FO(Concept), is purposely simple, to explain the essence of our ideas. We present the syntax (3.1) of the extension, its semantics (3.2), and discuss its complexity and some alternative formulations (3.3).

3.1 Syntax

A vocabulary Σ is a set of symbols with associated arity. We want to extend the FO syntax of terms over vocabulary Σ with these four new construction rules:

- n is a term if n is a numeral, i.e., a symbol denoting an integer;

- $\#\{x_1,\dots,x_n:\phi\}$ is a term if $x_1,\dots,x_n$ are variables and ϕ a formula;
- $`\sigma$ is a term if $\sigma \in \Sigma$,
- $\$(x)(t_1,\dots,t_n)$ is a term if x is a variable, and $t_1,\dots,t_n$ are terms over Σ.

The last rule is problematic, however. In a structure $\mathscr{I}$ extended as we propose, x ranges not only over the domain of $\mathscr{I}$, but also over the set of intension of the symbols in Σ: when the value of x is such an intension, $\$(x)$ is the extension of the associated symbols; but when x is not such an intension, $\$(x)$ is undefined. Furthermore, the value assigned to x may be the intension of a predicate symbol: in that case, $\$(x)(t_1,\dots,t_n)$ is not a term. Finally, the value of x may be the intension of a function of arity $m \neq n$: in that case, $\$(x)(t_1,\dots,t_n)$ is not a well-formed term.

For example, $\$(x)()$ is not a well-formed term in the following cases (among others):

. $[x = 1]$ where 1 denotes a numeric element of the domain of discourse;

. $[x = `p]$ where p is a predicate symbol;

. $[x = `f]$ where f is a function symbol of arity 1.

Essentially, whereas in FO, arities of function and predicate symbols are known from the vocabulary, and taken into account in the definition of well-formed composite terms and atoms by requesting that the number of arguments matches the arity, this now has become impossible due to the lack of information about $\$(x)$. Thus, to define well-formed terms, we need additional information about the variables occurring in them. We formalize that information in a typing function.

We call γ a *typing function* if it maps certain variables x to pairs (k,n) where k is either PRED or FUNC, and n is a natural number. Informally, when a variable x is mapped to (k,n) by γ, we know that x is a concept, of kind k and arity n. For a given typing function γ, we define $\gamma[x:(k,n)]$ to be the partial function γ' identical to γ except that $\gamma'(x) = (k,n)$.

This allows us to define the notion of well-formed term and formula:

Definition 1 (Well-formed term) *We define that a string e is a* well-formed term *over Σ given a typing function γ (denoted $\gamma \vdash_t e$) by induction:*

. $\gamma \vdash_t x$ if x is a variable;

. $\gamma \vdash_t f(t_1,..,t_n)$ if f is an n-ary function symbol of Σ and for each i, $\gamma \vdash_t t_i$;

\+ $\gamma \vdash_t$ n if n is a numeral, a symbol denoting the corresponding natural number n;

\+ $\gamma \vdash_t \#\{x_1,\dots,x_n:\phi\}$ if $x_1,\dots,x_n$ are variables and $\gamma \vdash_f \phi$;

\+ $\gamma \vdash_t `\sigma$ if $\sigma \in \Sigma$;

\+ $\gamma \vdash_t \$(x)(t_1,\dots,t_n)$ if x is a variable, $\gamma(x) = (\text{FUNC}, n)$ and for each $i \in [1,n]$, $\gamma \vdash_t t_i$.

(The rules with a "+" bullet are those we add to FO's definition of terms.)

Definition 2 (Well-formed formula) *We define that a string ϕ is a* well-formed formula *over Σ given γ (denoted $\gamma \vdash_f \phi$) by induction:*

. $\gamma \vdash_f$ **t**, $\gamma \vdash_f$ **f**;

. $\gamma \vdash_f p(t_1,\dots,t_n)$ if p is an n-ary predicate of Σ and for each $i \in [1,n]$, $\gamma \vdash_t t_i$;

. $\gamma \vdash_f (\neg\phi)$ if $\gamma \vdash_f \phi$;

. $\gamma \vdash_f (\phi \vee \psi)$ if $\gamma \vdash_f \phi$ and $\gamma \vdash_f \psi$;

. $\gamma \vdash_f \exists x : \phi$ if x is a variable and $\gamma \vdash_f \phi$;

. $\gamma \vdash_f t_1 = t_2$ if $\gamma \vdash_t t_1$ and $\gamma \vdash_t t_2$;

\+ $\gamma \vdash_f t_1 \leq t_2$ if $\gamma \vdash_t t_1$ and $\gamma \vdash_t t_2$ and t_1 and t_2 are numeric terms or cardinality aggregates;

\+ $\gamma \vdash_f (\mathbf{if}\, x :: k/n\, \mathbf{then}\, \phi\, \mathbf{else}\, \psi)$ if x is a variable, k is either PRED or FUNC,
n is a natural number, $\gamma[x : (k,n)] \vdash_f \phi$ and $\gamma \vdash_f \psi$;

\+ $\gamma \vdash_f \$(x)(t_1,\ldots,t_n)$ if x is a variable, $\gamma(x) = (\text{PRED}, n)$ and for each $i \in [1,n]$, $\gamma \vdash_t t_i$.

Notice that, because of the aggregate term rule, these definitions are mutually recursive. Also, due to their constructive nature, well-formed terms and formulae can be arbitrarily large, but always finite.

Formulae of the form $\phi \wedge \psi, \phi \Rightarrow \psi, \phi \Leftrightarrow \psi, \forall x : \phi$ are shorthand for these formulae, and are not further discussed:

$$\neg(\neg\phi \vee \neg\psi), \neg\phi \vee \psi, (\phi \wedge \psi) \vee (\neg\phi \wedge \neg\psi), \neg\exists x : \neg\phi$$

The other comparison operators, $<, >, \geq$, can be defined similarly.

Let $\varnothing$ be the typing function with empty domain. We say that ϕ is a *well-formed formula* over Σ if $\varnothing \vdash_f \phi$.

Example 2 *Here is a well-formed version of the cardinality sub-formula of Equation 2:*

$$\#\{x | \mathbf{if}\, x :: \text{PRED}/1\, \mathbf{then}\, Symptom(x) \wedge \$(x)(p)\, \mathbf{else}\, false\}$$

We say that $Symptom(x) \wedge \$(x)(p)$ is *guarded* by the $x :: \text{PRED}/1$ condition, and that the formula is *well-guarded*.

Note that, in this logic, a value $\$(x)$ cannot occur in a formula without immediately being applied to a tuple of arguments.

3.2 Semantics

Definition 3 (Intensional ontology and equivalence class $|\sigma|_{\mathscr{O}}$) *We define an* intensional ontology $\mathscr{O}$ *as a pair of a vocabulary Σ and a synonym relation $\sim_s$ between symbols of same arity in the vocabulary. We denote the equivalence class of symbol σ in $\mathscr{O}$ by $|\sigma|_{\mathscr{O}}$, and the set of such equivalence classes (or witnesses) by $\mathscr{C}_{\mathscr{O}}$.*

In the philosophical papers about intensional logic, the set of concepts is open. Here, however, we want to quantify only over the concepts that are interpretations of symbols in Σ, not over any concepts. In essence, we want to quantify over $\mathscr{C}_{\mathscr{O}}$.

Hence, to define the semantics of FO(Concept), we extend the notion of a structure to include $\mathscr{C}_{\mathscr{O}}$ and an additional mapping from concepts to their value.

Definition 4 (Total structure) *A* (total) structure $\mathscr{I}$ *over ontology $\mathscr{O}$ consists of:*

. an object domain D containing the set of natural numbers $\mathbb{N}$,

. a (total) mapping from predicate symbols p/n in Σ to n-ary relations $p^{\mathscr{I}}$ over $D \cup \mathscr{C}_{\mathscr{O}}$,

. a (total) mapping from function symbols f/n in Σ to n-ary functions $f^{\mathscr{I}}$ over $D \cup \mathscr{C}_{\mathscr{O}}$,

\+ a (total) mapping $\$^{\mathscr{I}}$ from concepts in $\mathscr{C}_{\mathscr{O}}$ to relations and functions over $D \cup \mathscr{C}_{\mathscr{O}}$.

$\$^{\mathscr{I}}$ interprets the value function "$\$(\cdot)$".

We now define a notion of *coherency* for structures $\mathscr{I}$, expressing that synonyms have the same value or, more formally, that $\$(\text{‘}\sigma) = \sigma$ for any σ.

Definition 5 (Coherent structure) *A total structure $\mathscr{I}$ over $\mathscr{O}$ is* coherent *iff*

+ for every predicate symbol $p \in \Sigma$, $\$^{\mathscr{I}}(|p|_{\mathscr{O}}) = p^{\mathscr{I}}$;
+ for every function symbol $f \in \Sigma$, $\$^{\mathscr{I}}(|f|_{\mathscr{O}}) = f^{\mathscr{I}}$.

From now on, we consider only coherent structures.

We define a *variable assignment* ν as a mapping of variables to elements in $D \cup \mathscr{C}_{\mathscr{O}}$. A variable assignment extended so that the mapping of x is d is denoted $\nu[x : d]$.

We introduce the ternary *valuation* function, that maps terms t (resp. formulae ϕ), structures $\mathscr{I}$ and variable assignment ν to values $v \in D \cup \mathscr{C}_{\mathscr{O}}$ (resp. to Booleans).

Definition 6 (Value of a term) *We partially define the value v of well-formed t in ($\mathscr{I}$, ν) (denoted $[\![t]\!]^{\mathscr{I}}_{\nu} = v$) by induction:*

. $[\![x]\!]^{\mathscr{I}}_{\nu} = \nu(x)$ if x is a variable in the domain of ν;
. $[\![f(t_1,\dots,t_n)]\!]^{\mathscr{I}}_{\nu} = f^{\mathscr{I}}([\![t_1]\!]^{\mathscr{I}}_{\nu},\dots,[\![t_n]\!]^{\mathscr{I}}_{\nu})$ if f is an n-ary function symbol and $[\![t_1]\!]^{\mathscr{I}}_{\nu},\dots,[\![t_n]\!]^{\mathscr{I}}_{\nu}$ are defined;
+ $[\![\mathrm{n}]\!]^{\mathscr{I}}_{\nu} = n$ if n is the integer denoted by n;
+ $[\![\#\{x_1,\dots,x_n : \phi\}]\!]^{\mathscr{I}}_{\nu} = m$ if $[\![\phi]\!]^{\mathscr{I}}_{\nu[x1:d1]\dots[xn:dn]}$ is defined for every $d1,\dots,dn \in D \cup \mathscr{C}_{\mathscr{O}}$, m is an integer, and $\#\{x_1,\dots,x_n : [\![\phi]\!]^{\mathscr{I}}_{\nu[x1:d1]\dots[xn:dn]} = \mathbf{t}\} = m$
+ $[\![\text{‘}S]\!]^{\mathscr{I}}_{\nu} = |S|_{\mathscr{O}}$;
+ $[\![\$(x)(t_1,\dots,t_n)]\!]^{\mathscr{I}}_{\nu} = \$^{\mathscr{I}}([\![x]\!]^{\mathscr{I}}_{\nu})([\![t_1]\!]^{\mathscr{I}}_{\nu},\dots,[\![t_n]\!]^{\mathscr{I}}_{\nu})$ if $[\![x]\!]^{\mathscr{I}}_{\nu}$ is a concept in $\mathscr{C}_{\mathscr{O}}$ mapped by $\$^{\mathscr{I}}$ to an n-ary function, and $[\![t_1]\!]^{\mathscr{I}}_{\nu},\dots,[\![t_n]\!]^{\mathscr{I}}_{\nu}$ are defined;
+ $[\![t]\!]^{\mathscr{I}}_{\nu}$ is undefined in all the other cases.

Definition 7 (Truth value of a formula) *We partially define the truth value v of well-formed ϕ in ($\mathscr{I}$, ν) (denoted $[\![\phi]\!]^{\mathscr{I}}_{\nu} = v$) by induction:*

. $[\![\mathbf{t}]\!]^{\mathscr{I}}_{\nu} = \mathbf{t}$; $[\![\mathbf{f}]\!]^{\mathscr{I}}_{\nu} = \mathbf{f}$;
. $[\![p(t_1,\dots,t_n)]\!]^{\mathscr{I}}_{\nu} = p^{\mathscr{I}}([\![t_1]\!]^{\mathscr{I}}_{\nu},\dots,[\![t_n]\!]^{\mathscr{I}}_{\nu})$ if p is an n-ary predicate symbol and $[\![t_1]\!]^{\mathscr{I}}_{\nu},\dots,[\![t_n]\!]^{\mathscr{I}}_{\nu}$ are defined;
. $[\![\neg\phi]\!]^{\mathscr{I}}_{\nu} = \neg[\![\phi]\!]^{\mathscr{I}}_{\nu}$ if $[\![\phi]\!]^{\mathscr{I}}_{\nu}$ is defined;
. $[\![\phi \vee \psi]\!]^{\mathscr{I}}_{\nu} = [\![\phi]\!]^{\mathscr{I}}_{\nu} \vee [\![\psi]\!]^{\mathscr{I}}_{\nu}$ if $[\![\phi]\!]^{\mathscr{I}}_{\nu}$ and $[\![\psi]\!]^{\mathscr{I}}_{\nu}$ are defined;
. $[\![\exists x : \phi]\!]^{\mathscr{I}}_{\nu} = (\exists d \in D \cup \mathscr{C}_{\mathscr{O}} : [\![\phi]\!]^{\mathscr{I}}_{\nu[x:d]})$ if $[\![\phi]\!]^{\mathscr{I}}_{\nu[x:d]}$ is defined for every $d \in D \cup \mathscr{C}_{\mathscr{O}}$;
. $[\![t_1 = t_2]\!]^{\mathscr{I}}_{\nu} = ([\![t_1]\!]^{\mathscr{I}}_{\nu} = [\![t_2]\!]^{\mathscr{I}}_{\nu})$ if and $[\![t_1]\!]^{\mathscr{I}}_{\nu}$ and $[\![t_2]\!]^{\mathscr{I}}_{\nu}$ are defined;
. $[\![t_1 \leq t_2]\!]^{\mathscr{I}}_{\nu} = ([\![t_1]\!]^{\mathscr{I}}_{\nu} \leq [\![t_2]\!]^{\mathscr{I}}_{\nu})$ if and $[\![t_1]\!]^{\mathscr{I}}_{\nu}$ and $[\![t_2]\!]^{\mathscr{I}}_{\nu}$ are defined and integers;
+ $[\![\mathbf{if}\, x :: k/n\, \mathbf{then}\, \phi\, \mathbf{else}\, \psi]\!]^{\mathscr{I}}_{\nu} = [\![\phi]\!]^{\mathscr{I}}_{\nu}$ if $[\![x]\!]^{\mathscr{I}}_{\nu}$ is a concept in $\mathscr{C}_{\mathscr{O}}$ mapped by $\$^{\mathscr{I}}$ to an n-ary predicate or function according to k, and $[\![\phi]\!]^{\mathscr{I}}_{\nu}$ is defined;
+ $[\![\mathbf{if}\, x :: k/n\, \mathbf{then}\, \phi\, \mathbf{else}\, \psi]\!]^{\mathscr{I}}_{\nu} = [\![\psi]\!]^{\mathscr{I}}_{\nu}$ if $[\![x]\!]^{\mathscr{I}}_{\nu}$ is not a concept in $\mathscr{C}_{\mathscr{O}}$ mapped by $\$^{\mathscr{I}}$ to an n-ary predicate or function according to k, and $[\![\psi]\!]^{\mathscr{I}}_{\nu}$ is defined;

- + $[\![\$(x)(t_1,\dots,t_n)]\!]^{\mathscr{I}}_{\nu} = \$^{\mathscr{I}}([\![x]\!]^{\mathscr{I}}_{\nu})([\![t_1]\!]^{\mathscr{I}}_{\nu},\dots,[\![t_n]\!]^{\mathscr{I}}_{\nu})$ if $[\![x]\!]^{\mathscr{I}}_{\nu}$ is a concept in $\mathscr{C}_{\mathscr{O}}$ mapped by $\$^{\mathscr{I}}$ to an n-ary predicate, and $[\![t_1]\!]^{\mathscr{I}}_{\nu},\dots,[\![t_n]\!]^{\mathscr{I}}_{\nu}$ are defined;
- + $[\![\phi]\!]^{\mathscr{I}}_{\nu}$ is undefined in all the other cases.

Theorem 1 *The truth value $[\![\phi]\!]^{\mathscr{I}}_{\nu}$ is defined for every well-formed formula ϕ over Σ, every total structure $\mathscr{I}$ over $(\Sigma, \sim_s)$, and every variable assignment ν that assigns a value to all free variables of ϕ.*

This can be proven by parallel induction over the definitions of well-formed formulae (and terms) and of their value, applying the properties of $\mathscr{I}$ and ν when needed. Indeed, the conditions in the inductive rules of the valuation function match the conditions in the inductive rules of well-formed formulae (and terms).

A *sentence* is a well-formed formula without free variables. We say a total, coherent structure $\mathscr{I}$ *satisfies* sentence ϕ iff $[\![\phi]\!]^{\mathscr{I}}_{\nu} = \mathbf{t}$ for any ν. This is also denoted $\mathscr{I} \models \phi$. Coherent structures $\mathscr{I}$ that satisfy sentence ϕ are called *models* of ϕ.

3.3 Complexity and alternative formulations

The determination of the well-formedness of a formula (Section 3.1) can be performed by backward reasoning, using the appropriate inductive rule at each step based on the syntactic structure of the formula under consideration. Each inductive step consists of simple syntactical analysis, and, for formulae having sub-formulae, of determining the well-formedness of each sub-formula; formula $\$(x)(t_1,\dots,t_n)$ requires an additional lookup of x in γ. Notice that, at each step, the complexity of the reasoning does not depend on the size of the domain of discourse. Bounding the cost of both the syntactical analysis and of the variable lookup by α, the cost C of the computation for any formula ϕ having sub-formulae ϕ_i is bounded as follows: $C(\phi) \leq \alpha + \sum_i C(\phi_i) \leq \alpha \times N(\phi)$, where $N(\phi)$ is the number of tokens in the formula ($N(\phi) = 1 + \sum_i N(\phi_i)$). Thus, the complexity is linear with the number of tokens in the formula.

The complexity of decision and search problems in FO(Concept) is the same as in FO. This is because the domain D of structures is extended only with a fixed, constant-sized set $\mathscr{C}_{\mathscr{O}}$ of elements (and not over higher-order objects). For example, deciding the existence of a model of an FO (resp. FO(Concept)) theory with an input domain D (resp. $D \cup \mathscr{C}_{\mathscr{O}}$) is an NP problem measured in the size of D (resp. $D \cup \mathscr{C}_{\mathscr{O}}$).

FO could be extended to support quantification over concepts in other ways than the one presented above. Instead of the **if** .. **then** .. **else** .. construct, we could use $\lor$ (and $\land$) with non-strict evaluation. Well-guarded quantifications would then be written as follows:

$$\exists x((x :: \text{PRED}/1) \land \$(x)(p)) \tag{6}$$

$$\forall x((x :: \text{PRED}/1) \implies \$(x)(p)) \tag{7}$$

Also, instead of giving *partial* definitions of values (Def. 6 and 7), we could give *total* definitions by assigning an arbitrary value when the term (resp. formula) is undefined. We would then show that this arbitrary value is not relevant in well-formed formula.

4 Extending FO(·) with intensions

We now discuss how we extended the Knowledge Representation language called FO(·) [17, 16] to support quantification over intensions.

FO(·) is first-order logic extended with language constructs to make it more expressive:

- types: the vocabulary may include custom types, in addition to the built-in types (e.g., `Int, Bool`). Each symbol has a type signature of the form `T1×...×Tn→T` specifying their domain and range. The range of a predicate is the set of Booleans `Bool`. Formula must be well-typed, i.e., predicates and functions must be applied to arguments of the correct type.
- equality: `t1=t2` is a formula, where `t1` and `t2` are terms of the same type.
- arithmetic over integers and rationals: arithmetic operators (`+,-,*,/`) and comparisons (e.g., $\leq$) are interpreted functions.
- binary quantification: `∃x ∈ P: p(x).` (where `P` is a type or predicate) is equivalent to `∃x: P(x) ∧ p(x).`
- aggregations, such as `#{x ∈ P: p(x)}` (count of `x` in `P` satisfying `p`) and `sum{{f(x) | x ∈ P}}` (sum of `f(x)` over `x` in `P`).
- (inductive) definitions [17]: FO(·) theory consists of a set of logic sentences and a set of (potentially inductive) definitions. Such a definition is represented as a set of rules of the form:
  `∀ x1 in T1,..,xn in Tn: p(x1,..,xn) ← F.`
  where F is an FO(·) formula.

Thanks to the support of inductive definitions, FO(·) is unique in combining the expressivity of classical logic and logic programming [16]. The reference manual of FO(·) is available online[1].

A Knowledge Base (KB) written in FO(·) cannot be run: like human knowledge, it is just a "bag of information", formally describing models in a problem domain. Knowledge bases do not distinguish inputs from outputs, and allow reasoning in any direction. They are used to perform a variety of reasoning tasks (each with a particular set of inputs and outputs), using generic methods provided by reasoning engines, such as IDP-Z3[2] and FOLASP [20]: they can find relevant questions to solve a particular problem, derive the consequences of new information, explain how they derived these consequences, find possible models, and find a model that minimizes a cost function [15].

These generic capabilities are used to easily build knowledge-based interactive systems that assist users in finding solutions to problems in a problem domain [13, 19].

To allow reasoning about concepts, we have extended FO(·) with the "`‘.`" operator (to refer to the intension of a symbol) and the "`$(.)`" operator (to refer to the interpretation of a concept), as described in Section 3 for FO.

An issue arises in expressions of the form

```
$(x)(t1,...,tn)
```

x must be a `Concept`, the arguments `t1,...,tn` must be of appropriate types and number for the predicate or function `$(x)`, and `$(x)(t1,...,tn)` must be of the type expected by its parent expression.

To address this issue and to support the writing of well-guarded formulae, we have introduced types for the concepts having a particular type signature:

```
Concept[T1 × .. × Tn → T]
```

The interpretation of this type is the set of concepts with signature `T1×..×Tn→T`. Note that `T1,..,Tn,T` themselves can be conceptual types.

The well-formedness and semantics of quantifications over a conceptual type :

---

[1] `https://fo-dot.readthedocs.io/`

[2] `http://idp-z3.be/`

```
∃x ∈  Concept[T1 × .. × Tn  →  T]: $(x)(t1, .., tn).
```

is defined by extending the concept of guards (Section 3.1), and considering the following equivalent statement with guards:

```
∃x: if x::[T1 × .. × Tn  →  T] then $(x)(t1,..,tn) else false.
```

where `x::[T1×..×Tn → T]` is a guard specifying the types of the arguments of *x*, and the type of `$(x)(t1,...,tn)`. The definitions of the typing function γ (Section 3.1), and of well-formed formulae and their semantics are updated accordingly.

The syntax of FO(·) allows applying the "`$(.)`" operator to expressions (not just to variables), e.g., in atom `$(expr)()`. The type of `expr` must be a conceptual type with appropriate signature.

We have updated the IDP-Z3 reasoning engine[3] to support such quantification over concepts. IDP-Z3 transforms FO(·) theories into the input language of the Z3 SMT solver [29] to perform various reasoning tasks. A quantification

```
∃ x in Concept[T1 × .. × Tn→T]: expr(x).
```

is transformed into a disjunction of `expr(‘c)` expressions, where `‘c` is a `Concept[T1×..×Tn→T]`. Occurrences of `$(‘c)(t1,..tn)` within `expr(‘c)` are transformed into `c(t1,..tn)` where `c` is the symbol denoting concept `‘c` (Our implementation currently does not support synonymous concepts). The resulting formula is an FO sentence that can be submitted to Z3.

5 Examples in FO(·)

In our practice, we have identified a few examples where quantifications over concepts proved essential to accurately model the knowledge available within a domain in an elaboration tolerant way, i.e., without reification. These examples come from a broad range of applications. They are:

- the May 2021 DM Community challenge[4] about deciding to perform additional testing of patients, based on a set of symptoms (similar to the example we used in the introduction);
- the "International Law to fight money laundering" example, where we want to represent the general rule that the national laws must be stricter than the EU directive.
- the "Word disambiguation" example, in which the word "child" in a statutory law could represent either the biological or legal child.
- the "template" example, in which intensional objects are used to define templates.

The examples are available online[5], and can be run using the IDP-Z3 reasoning engine. Below, we discuss the International law example for illustration purposes.

Since 1990, the European Union has adopted legislation to fight against money laundering and terrorist financing. It creates various obligations for the parties in a business relationship, such as verifying the identity of the counter-party. The member states have to transpose the directive into national laws. The national laws must meet the minimum obligations set forth in the EU directive.

---

[3] `http://idp-z3.be/`

[4] `https://dmcommunity.org/challenge/challenge-may-2021/`

[5] `https://tinyurl.com/Intensions`

In our simplified example, the EU directive requires the verification of identity in any transaction with a value above 1M€; a national law might set the threshold at 500K€instead. Similarly, the EU directive might require a bank to send a report to their authority at least quarterly, but a country might require a monthly report.

Our goal is then to express the requirement that the national obligations are stricter than the EU ones. We choose an ontology in which "has a lower value" means "stricter". We also use a mapping from the parameters of the national laws to their equivalent parameter in the EU law.

```
vocabulary {
    type Country
    threshold, period: Country → Int
    obligation: Concept[Country→Int]→Bool
    thresholdEU, periodEU: () → Int
    mapping: Concept[Country→Int]→ Concept[()→Int]
}
theory {
    obligation := {‘threshold, ‘period}
    mapping := {‘threshold →‘thresholdEU, ‘period→‘periodEU}

    // national law must be stricter than European law.
    ∀o ∈ obligation: ∀c ∈ Country: $(o)(c) ≤ $(mapping(o))().
}
```

Notice that an expression (not a variable) is applied to the value operator: `$(mapping(o))()`. Because `o` is a `Concept[Country→Int]` by quantification, it is in the domain of `mapping`, and `mapping(o)` has type `Concept[()→Int]`. Its value is thus a nullary function of range `Int`.

6 Related Work

We discussed the relation between intensional logic and our work in Section 2. We now discuss the relation with other work.

6.1 Second-order quantification and relations

A contribution of this work is to clarify the utility of being able to quantify over concepts in the vocabulary and to show how it differs from quantifying over sets or functions as in second-order quantification. Certainly in the case of predicate or function intensions, it is our own experience that it is easy to confuse the two. We showed in the introduction the need for quantification over intensions as opposed to over second or higher-order objects (relations or functions). However, the quantification over intensions cannot replace quantification over second-order. A clear-cut example of second-order quantification and relation occurs in the *graph mining problem* [23].

Example 3 *A graph* homomorphism *h is a function from the nodes of a graph, called the* pattern *p, to nodes of a graph g, that preserves the edges. The following expression intends to define hom/1 as the set of homomorphisms of p in g, where $p(x,y)$ (resp. $g(x,y)$) denotes an edge between nodes x and y.*

$$\forall h(hom(h) \Leftrightarrow \forall x \forall y(p(x,y) \Leftrightarrow g(h(x),h(y))))$$

The question is: over what sorts of values does h range (in the context of a structure $\mathscr{I}$)? Over the set of intensions of unary function symbols in Σ? There may be none! Or over the set of all binary functions from nodes of the pattern to nodes of the graph in $\mathscr{I}$? Clearly, over the latter. Hence, h is a second-order variable and $hom/1$ a second-order predicate symbol.

6.2 Metaprogramming

Variables ranging over concepts, as we propose, are similar to variables ranging over symbols, as found in meta programming. Indeed, meta programming also brings the benefit that we seek: formulating knowledge about concepts in an elaboration tolerant way (i.e, without reification).

Meta programming has been proposed in logic-based languages such as OWL [28, 30], as well as in rule-based languages such as Prolog [4], HiLog [10], or webdamlog [1], but not in languages integrating the two types of formulation, such as FO(·). This integration is needed to support all the use cases that we describe. For example, the International Law example encodes a logic constraint that cannot be encoded in a rule-based system; the Template example encodes an inductive definition that cannot be encoded in logic-based systems. By contrast, our reasoning engine does support the two types of formulation, and all the use cases.

Furthermore, we formalize the guarding mechanism that ensures that sentences are well-formed. To the best of our knowledge, this mechanism is missing in all meta-programming implementations (and in particular in HiLog and webdamlog). Without this mechanism, the developer of a knowledge base cannot benefit from the automatic detection of syntactical errors.

7 Summary

As our examples illustrate, it is often useful in knowledge-intensive applications to quantify over concepts, i.e., over the intensions of the symbols in an ontology. An intension is an atomic object representing the informal interpretation of the symbol in the application domain.

First-order logic and FO(·) can be extended to allow such quantifications in an elaboration tolerant way. Appropriate guards should be used to ensure that formulae in such extensions are well-formed: we propose a method to verify such well-formedness with a complexity that is linear with the number of tokens in the formula.

While related to modal logic, the logic introduced here differs strongly from it. It has no modal operators to talk about the extension of a symbol in other worlds than the contextual one, but it offers mechanisms to quantify and count intensional objects.

References

[1] Serge Abiteboul, Meghyn Bienvenu, Alban Galland & Émilien Antoine (2011): *A rule-based language for web data management*. In Maurizio Lenzerini & Thomas Schwentick, editors: *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece*, ACM, pp. 293–304, doi:10.1145/1989284.1989320.

[2] Harvey Abramson & M. H. Rogers, editors (1989): *Meta-Programming in Logic Programming, Workshop on Meta-Programming in Logic, META 1988, University of Bristol, 22-24 June, 1988*. MIT Press.

[3] Chitta Baral, Juraj Dzifcak & Hiro Takahashi (2006): *Macros, macro calls and use of ensembles in modular answer set programming*. In: *International Conference on Logic Programming*, Springer, pp. 376–390,

doi:10.1007/11799573_28. Available at `https://link.springer.com/chapter/10.1007/11799573_28`.

[4] Jonas Barklund, Pierangelo Dell Acqua, Stefania Costantini & Gaetano Aurelio Lanzarone (2000): *Reflection principles in computational logic*.

[5] Sergey Berezin, Clark W. Barrett, Igor Shikanian, Marsha Chechik, Arie Gurfinkel & David L. Dill (2005): *A Practical Approach to Partial Functions in CVC Lite*. Electron. Notes Theor. Comput. Sci. 125(3), pp. 13–23, doi:10.1016/j.entcs.2004.06.064.

[6] Bart Bogaerts, Joachim Jansen, Maurice Bruynooghe, Broes De Cat, Joost Vennekens & Marc Denecker (2014): *Simulating Dynamic Systems Using Linear Time Calculus Theories*. Theory Pract. Log. Program. 14(4-5), pp. 477–492, doi:10.1017/S1471068414000155.

[7] P. Carbonnelle (2021): *Links for this paper*. `https://tinyurl.com/Intensions`. Accessed: 2021-12-10.

[8] Pierre Carbonnelle (2021): *IDP-Z3*. `https://idp-z3.be`. Accessed: 2021-03-15.

[9] Pierre Carbonnelle, Simon Vandevelde, Joost Vennekens & Marc Denecker (2022): *IDP-Z3: a reasoning engine for FO(.)*. arXiv:2202.00343.

[10] Weidong Chen, Michael Kifer & David S Warren (1993): *HiLog: A foundation for higher-order logic programming*.

[11] Patrick Cousot & Radhia Cousot (1992): *Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation*. In Maurice Bruynooghe & Martin Wirsing, editors: Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92, Leuven, Belgium, August 26-28, 1992, Proceedings, Lecture Notes in Computer Science 631, Springer, pp. 269–295, doi:10.1007/3-540-55844-6_142.

[12] Ingmar Dasseville, Matthias van der Hallen, Gerda Janssens & Marc Denecker (2015): *Semantics of templates in a compositional framework for building logics*. Theory Pract. Log. Program. 15(4-5), pp. 681–695, doi:10.1017/S1471068415000319.

[13] Ingmar Dasseville, Laurent Janssens, Gerda Janssens, Jan Vanthienen & Marc Denecker (2016): *Combining DMN and the Knowledge Base Paradigm for Flexible Decision Enactment*. In Tara Athan, Adrian Giurca, Rolf Grütter, Mark Proctor, Kia Teymourian & William Van Woensel, editors: Supplementary Proceedings of the RuleML 2016 Challenge, Doctoral Consortium and Industry Track hosted by the 10th International Web Rule Symposium, RuleML 2016, New York, USA, July 6-9, 2016, CEUR Workshop Proceedings 1620, CEUR-WS.org. Available at `http://ceur-ws.org/Vol-1620/paper3.pdf`.

[14] Benjamin Lent Davis & Diane Maclagan (2003): *The card game SET*. The Mathematical Intelligencer 25(3), pp. 33–40, doi:10.1007/BF02984846.

[15] Broes De Cat, Bart Bogaerts, Maurice Bruynooghe, Gerda Janssens & Marc Denecker (2018): *Predicate logic as a modeling language: the IDP system*. In Michael Kifer & Yanhong Annie Liu, editors: Declarative Logic Programming: Theory, Systems, and Applications, ACM / Morgan & Claypool, pp. 279–323, doi:10.1145/3191315.3191321.

[16] Marc Denecker (2000): *Extending Classical Logic with Inductive Definitions*. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv & Peter J. Stuckey, editors: Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings, Lecture Notes in Computer Science 1861, Springer, pp. 703–717, doi:10.1007/3-540-44957-4_47.

[17] Marc Denecker & Eugenia Ternovska (2008): *A logic of nonmonotone inductive definitions*, doi:10.1145/1342991.1342998. Available at `https://doi.org/10.1145/1342991.1342998`.

[18] Marc Denecker & Joost Vennekens (2014): *The well-founded semantics is the principle of inductive definition, revisited*. In: Fourteenth International Conference on the Principles of Knowledge Representation and Reasoning.

[19] Marjolein Deryck, Joost Vennekens, Jo Devriendt & Simon Marynissen (2019): *Legislation in the Knowledge Base Paradigm: Interactive Decision Enactment for Registration Duties*. In: 13th IEEE International

Conference on Semantic Computing, ICSC 2019, Newport Beach, CA, USA, January 30 - February 1, 2019, IEEE, pp. 174–177, doi:10.1109/ICOSC.2019.8665543.

[20] Kylian Van Dessel, Jo Devriendt & Joost Vennekens (2021): *FOLASP: FO(.) as Input Language for Answer Ser Solvers*. arXiv:2108.04020.

[21] Melvin Fitting (2004): *First-order intensional logic*.

[22] Melvin Fitting (2020): *Intensional Logic*. In Edward N. Zalta, editor: *The Stanford Encyclopedia of Philosophy*, Spring 2020 edition, Metaphysics Research Lab, Stanford University.

[23] Matthias van der Hallen, Sergey Paramonov, Gerda Janssens & Marc Denecker (2019): *Knowledge representation analysis of graph mining*, doi:10.1007/s10472-019-09624-y. Available at `https://doi.org/10.1007/s10472-019-09624-y`.

[24] Patricia Hill & John Gallagher (1998): *Meta-programming in logic programming*. *Handbook of Logic in Artificial Intelligence and Logic Programming* 5, pp. 421–497.

[25] Giovambattista Ianni, Giuseppe Ielpa, Adriana Pietramala, Maria Carmela Santoro & Francesco Calimeri (2004): *Enhancing answer set programming with templates*. In: *NMR*, pp. 233–239.

[26] John McCarthy (1998): *Elaboration tolerance*. In: *Common Sense*, 98, Citeseer.

[27] Richard Montague (1970): *Pragmatics and intensional logic*. *Synthèse* 22, p. 68–94, doi:10.1007/BF00413599.

[28] Boris Motik (2007): *On the Properties of Metamodeling in OWL*, doi:10.1093/logcom/exm027. Available at `https://doi.org/10.1093/logcom/exm027`.

[29] Leonardo Mendonça de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, *Lecture Notes in Computer Science* 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[30] Jeff Z. Pan, Ian Horrocks & Guus Schreiber (2005): *OWL FA: A Metamodeling Extension of OWL DL*. In Bernardo Cuenca Grau, Ian Horrocks, Bijan Parsia & Peter F. Patel-Schneider, editors: *Proceedings of the OWLED*05 Workshop on OWL: Experiences and Directions, Galway, Ireland, November 11-12, 2005*, *CEUR Workshop Proceedings* 188, CEUR-WS.org. Available at `http://ceur-ws.org/Vol-188/sub15.pdf`.

[31] Nikolay Pelov & Eugenia Ternovska (2005): *Reducing Inductive Definitions to Propositional Satisfiability*. In Maurizio Gabbrielli & Gopal Gupta, editors: *Logic Programming, 21st International Conference, ICLP 2005, Sitges, Spain, October 2-5, 2005, Proceedings*, *Lecture Notes in Computer Science* 3668, Springer, pp. 221–234, doi:10.1007/11562931_18.

[32] Terrance Swift & David S Warren (2012): *XSB: Extending Prolog with tabled logic programming*. *Theory and Practice of Logic Programming* 12(1-2), pp. 157–187, doi:10.1017/S1471068411000500.

[33] Guizhen Yang, Michael Kifer & Chang Zhao (2003): *Flora-2: A Rule-Based Knowledge Representation and Inference Infrastructure for the Semantic Web*. In Robert Meersman, Zahir Tari & Douglas C. Schmidt, editors: *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE - OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003*, *Lecture Notes in Computer Science* 2888, Springer, pp. 671–688, doi:10.1007/978-3-540-39964-3_43.

[34] Guizhen Yang, Michael Kifer, Chang Zhao & Vishal Chowdhary (2005): *FLORA-2: User's manual*. *Version 0.94 (Narumigata). April* 30.

A Logic Programming Approach to Global Logistics in a Co-Design Environment

Emmanuelle Dietz[1], Tobias Philipp[2], Gerrit Schramm[1], Andreas Zindel[1]*

[1] Airbus Central Research & Technology, Germany
[2] secunet Security Networks AG, Germany
`{firstname.lastname}@airbus.com,secunet.com`

In a co-design environment changes need to be integrated quickly and in an automated manner. This paper considers the challenge of creating and optimizing a global logistics system for the construction of a passenger aircraft within a co-design approach with respect to key performance indicators (like cost, time or resilience). The product in question is an aircraft, comprised of multiple components, manufactured at multiple sites worldwide. The goal is to find an optimal way to build the aircraft taking into consideration the requirements for its industrial system. The main motivation for approaching this challenge is to develop the industrial system in tandem with the product and making it more resilient against unforeseen events, reducing the risks of bottlenecks in the supply chain. This risk reduction ensures continued efficiency and operational success. To address this challenging and complex task we have chosen Answer Set Programming (ASP) as the modeling language, formalizing the relevant requirements of the investigated industrial system. The approach presented in this paper covers three main aspects: the extraction of the relevant information from a knowledge graph, the translation into logic programs and the computation of existing configurations guided by optimization criteria. Finally we visualize the results for an effortless evaluation of these models. Internal results seem promising and yielded several new research questions for future improvements of the discussed use case.

1 Introduction

Co-design is a holistic approach ([23]) for the development of new products in tandem with an existing industrial system, and is essential for the development of innovative and complex products. Consider the scenario that a product development department chooses a material that cannot be processed by the existing industrial system. This may lead to significant cost increases and bottlenecks later in the lifecycle. In contrast, a co-design enabled environment encourages trade-offs and suggests to adapt either the product or the industrial system. The results of that trade-off are integrated into the future developments, before the product is ready-to-market.

The first step towards co-design is to determine if a product can be produced by the existing industrial system. This assessment is especially difficult for products — like aircrafts — where not only the product itself is complex but which is also manufactured by a global industrial system in an intricate logistics network. Therefore, we propose the following approach: a data model to represent the industrial system and the product, means to capture the data about these systems, a method to create valid system architectures, ways to calculate key performance indicators and lastly visualize the results.

Creating a unified data model is a challenging task since the co-design approach requires to combine different data domains together. We propose an Resource Description Framework (RDF) based ontology to create the data model. This ontology needs to be instantiated with the available company knowledge

---

[*] The authors are in alphabetical order.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 227–240, doi:10.4204/EPTCS.385.23

captured in multiple sources, resulting in a co-design knowledge graph. Since the data can change anytime the attached domains develop their respective systems further the knowledge graph needs to be as lean as possible to quickly adapt to the evolving environment. After each iteration of the 'to be developed' product new variants of the industrial system need to be created and evaluated. A naive approach would expand all possible variants and evaluate each of them. However, given a sufficiently complex system, this would soon extend the capabilities of even the most advanced computer systems.[1]

We address this challenging task by applying ASP, an expressive, logic-based declarative modeling language and problem-solving framework for hard computational problems ([24, 11]). Given that these techniques require deep knowledge about the deployed technologies this process needs to be as seamless and automated as possible to be suitable for a co-design framework. Thus, it is supported by a Continuous Integration (CI) based process to integrate company knowledge into ASP statements automatically. For a straightforward evaluation the results are visualized.

The paper is structured as follows: After the preliminaries, Section 3 illustrates the structure of the data as a knowledge graph, the extraction of the relevant information and its conversion to logic program facts. Next, the (choice) rules and integrity constraints for finding possible logistics configurations optimized with respect to certain performance indicators and the visual representation of these configurations are presented. Section 4 summarizes some lessons learned.

2 Background

The proposed framework is based on a variety of modules that exploit the technological advancements in Software Engineering and Knowledge Representation and Reasoning. This section introduces the preliminaries for knowledge extraction and computation.

2.1 Semantic Web Technologies

The World Wide Web Consortium (W3C) defined a standardized technology stack for representing data in a semantic way. These technologies can be used to create ontologies that model the investigated system and populating them with entities to build a knowledge graph.

In our context an ontology is defined as *a formal, explicit specification of shared conceptualization* ([14]). *Formal* because an ontology needs to be machine-readable, *explicit specification* demands the capability of defining concepts, properties, constraints and axioms, *shared* as it represents consensual knowledge while *conceptualization* defines an ontology as an abstract model, a simplified view of the system.

The W3C technology stack proposes RDF as the standard model to represent information in the form of Subject Predicate Object (SPO) triples ([6]). To be valid RDF each part of the triple has to be an Internationalized Resource Identifier (IRI), a data type literal — that stores discrete values — or a blank node. Blank nodes are either subjects or objects that do not have an IRI or literal. RDF can be serialized into a variety of formats such as *Turtle*, *RDF/XML* or *JSON-LD* depending on the consuming interface ([1]).

To foster re-usability several standardized RDF vocabularies are available. The Web Ontology Language is an RDF vocabulary that allows the creation of ontologies ([33]). Web Ontology Language (OWL) implements classes (`owl:Class`), properties (e.g. `owl:ObjectProperty`, , `owl:DatatypeProperty`), constraints (e.g. `owl:minCardinality`) and axioms (e.g. `owl:equivalentClass`). An

---

[1] A product with 70 parts, 16 possible manufacturers and 2 manufactures per part yields 10^{166} variants.

ontology forms one part of a knowledge graph, called the terminological component (*TBox*). The other part, that contains instances of the classes defined in the *TBox*, is called assertion component (*ABox*) ([14]). Tools like *Protégé* support the creation of OWL ontologies by providing a graphical editor to implement concepts and relations ([31]).

OWL can be split into several fragments with varying levels of expressiveness. These fragments, like OWL-DL, provide a subset of constructs of the full implementation of OWL. Since OWL-DL implements the $\mathscr{SROIQ}(\mathscr{D})$ Description Logic (hence the suffix DL) ([16]) reasoning engines can ingest knowledge graphs backed by an OWL-DL ontology in order to infer new knowledge based on existing facts.

Semantic Web Rule Language (SWRL) provides a way to express these inferences rules. These rule sets can be used by the reasoning engines to create new knowledge ([17]). OWL-DL (like all OWL fragments) is using the Open World Assumption (OWA) for reasoning. Possibly, reasoners might not terminate, since OWL and its fragments are undecidable.

In order to access data stored in the knowledge graphs specialized data stores and query languages have been developed. SPARQL Protocol and RDF Query Language (SPARQL) is the W3C recommendation for querying knowledge graphs implemented with RDF ([15]). SPARQL provides expressions and features of traditional query languages for relation databases like the Structured Query Language (SQL). However, since SPARQL is implemented as a graph pattern matching language, it provides additional methods for traversing RDF graphs, querying multiple graphs at once and constructing sub-graphs. SPARQL is running under the Closed World Assumption (CWA), either it matches a graph pattern or not. Executing a query will always yield a result, even if empty. An extension of SPARQL called SPARQL Update allows to manipulate the data in the knowledge graph by inserting, updating and deleting triples with the help of a query ([9]).

Specialized data stores for RDF data are called triple stores. These stores come in two varieties: they are either storing RDF data natively, or convert them into their own, internal data format. Usually these storage solutions come with tools, Application Programming Interfaces (APIs) and SPARQL endpoints to extract and manipulate RDF triples. One such storage solution is MarkLogic ([22]). MarkLogic is a NoSQL document store that stores all data in Extensible Markup Language (XML) documents. In order to store RDF data the RDF triples are put into documents. Internally the provided SPARQL endpoints access the data stored in the documents.

MarkLogic provides reasoning at query run time; so called backward-chaining inference ([22]). However, this is resource intensive. To facilitate fast query execution times, reasoning via ontologies or SWRL rules should be done before executing the query, explicitly materializing the inferred facts. Not only due to the additional run time of the reasoning step but also because of the undecidability of OWL.

We focus on two areas of Semantic Web technologies: unify data from several diverse sources supported by an ontology and generate new knowledge with the help of reasoning to keep the resulting knowledge graph as concise and easy to maintain as possible.

2.2 Logic Programming and Stable Model Semantics

We introduce the general notation and terminology used in this paper and assume the reader to be familiar with ASP ([24, 11]) and the stable model semantics ([12]). The interested reader is referred to [3] and [18].

We consider the countable set of *terms* $\mathscr{T} = \{t_1, \ldots, t_n\}$ that consists only of constants and variables with a total order $\leq$ over the elements in $\mathscr{T}$. An *atom* is an expression $p(t_1, \ldots t_m)$ where p is a predicate, $m \geq 0$ and $t_1, \dots t_m \subseteq \mathscr{T}$. $\mathscr{A}$ is a fixed, finite and non-empty set of *atoms*. A *ground atom* is an atom with only constants. If A is an atom, then A and *not* A are *literals*, the *positive literal* and the *negative literal*, respectively. A *ground formula* is a formula with only constants. Consider the example where productionLoc and warehouseLoc are predicates, hamburg and hamburgHarbor are constants and X is a variable: productionLoc(hamburg) and warehouseLoc(hamburgHarbor) are ground atoms, *not* productionLoc(hamburgHarbor) is a negative ground literal and productionLoc(X) is a non-ground atom. A (normal) *rule* r is of the form

$$A \leftarrow A_1, \dots, A_m, not\, A_{m+1}, \dots, not\, A_n. \tag{1}$$

where A and A_i with $1 \leq i \leq n$ are atoms. The *head* of r is denoted as $H(r) = A$. The subformula to the right of the implication symbol is called *body* of r and is denoted as $B(r) = A_1, \dots, A_m, not\, A_{m+1}, \dots, not\, A_n$. The set of all positive literals in $B(r)$ is denoted as $B^+(r) = \{A_1, \dots, A_m\}$ and the set of all negative literals in $B(r)$ is denoted as $B^-(r) = \{not\, A_{m+1}, \dots, not\, A_n\}$. A rule is *safe* if each variable in r occurs in $B^+(r)$. A rule is *ground* if no variable occurs in r. A *fact* is a ground rule with empty body, i.e. $n = 0$. The rule $\mathsf{loc}(X) \leftarrow \mathsf{productionLoc}(X)$ is safe, but neither ground nor a fact.

A *logic program* $\mathscr{P}$ is a (finite) set of (normal) rules. For any program $\mathscr{P}$, let $\mathscr{C}$ be the set of all constants appearing in $\mathscr{P}$. In the sequel, we assume for all $\mathscr{P}$ it holds that $\mathscr{C}_{\mathscr{P}} \neq \emptyset$. The ground program $\mathrm{g}\mathscr{P}$ is the set of rules $r\sigma$ obtained by applying to each rule $r \in \mathscr{P}$, all possible substitutions σ from the variables in r to the elements of $\mathscr{C}_{\mathscr{P}}$.

Program $\mathscr{P}_{\mathsf{loc}}$ consists of the following rules, where $\mathscr{C}_{\mathscr{P}_{\mathsf{loc}}} = \{\text{hamburg}, \text{hamburgHarbor}\}$:

$$\mathsf{loc}(X) \leftarrow \mathsf{productionLoc}(X). \quad \mathsf{productionLoc}(\text{hamburg}). \quad \mathsf{warehouseLoc}(\text{hamburgHarbor}).$$

The ground program $\mathrm{g}\mathscr{P}_{\mathsf{loc}}$ consists of the following rules:

$$\begin{array}{rll} \mathsf{loc}(\text{hamburg}) & \leftarrow \mathsf{productionLoc}(\text{hamburg}). & \mathsf{productionLoc}(\text{hamburg}). \\ \mathsf{loc}(\text{hamburgHarbor}) & \leftarrow \mathsf{productionLoc}(\text{hamburgHarbor}). & \mathsf{warehouseLoc}(\text{hamburgHarbor}). \end{array}$$

The set of all atoms in $\mathrm{g}\mathscr{P}$ is denoted by $\mathscr{A}_{\mathscr{P}}$. An *interpretation* I of a program $\mathscr{P}$ is a mapping of $\mathscr{A}_{\mathscr{P}}$ to the set of truth values $\{\top, \bot\}$, where $\top$ means *true* and $\bot$ means *false*. Given a ground rule r, $I(r) = \top$ denotes that the interpretation I maps r to $\top$ according to the corresponding logic. An interpretation I is a *model* of $\mathscr{P}$ where for each rule r occurring in $\mathrm{g}\mathscr{P}$ it holds that $I(r) = \top$. Consider $\mathrm{g}\mathscr{P}_{\mathsf{loc}}$ again.

$$\begin{aligned} \mathscr{A}_{\mathscr{P}_{\mathsf{loc}}} = \{ & \mathsf{loc}(\text{hamburg}), \mathsf{productionLoc}(\text{hamburg}), \mathsf{loc}(\text{hamburgHarbor}), \\ & \mathsf{productionLoc}(\text{hamburgHarbor}), \mathsf{warehouseLoc}(\text{hamburgHarbor}) \}. \end{aligned}$$

$I_1 = \mathscr{A}_{\mathscr{P}_{\mathsf{loc}}}$, $I_2 = \{\mathsf{loc}(\text{hamburg}), \mathsf{productionLoc}(\text{hamburg}), \mathsf{warehouseLoc}(\text{hamburgHarbor})\}$ and $I_3 = \emptyset$ are interpretations of $\mathscr{P}_{\mathsf{loc}}$, but only I_1 and I_2 are models of $\mathscr{P}_{\mathsf{loc}}$. An interpretation $I \subseteq \mathscr{A}_{\mathscr{P}}$ *satisfies* a ground rule r iff $H(r) \cap I \neq \emptyset$ whenever $B^+(r) \subseteq I$ and $B^- \cap I = \emptyset$. I satisfies a ground program $\mathscr{P}$, if each $r \in \mathscr{P}$ is satisfied by I. A non-ground rule r (resp. a program $\mathscr{P}$) is satisfied by an interpretation I if I satisfies all ground instances of r (resp. $\mathrm{g}\mathscr{P}$). I is an *answer set* (also called *stable model*) of $\mathscr{P}$ iff I is the subset-minimal set satisfying the *Gelfond-Lifschitz reduct*: $\mathscr{P}^I = \{H(r) \leftarrow B^+(r) \mid I \cap B^-(r) = \emptyset, r \in \mathrm{g}\mathscr{P}\}$. I_2 is the only answer set of $\mathscr{P}_{\mathsf{loc}}$.

The following two syntactic extensions are commonly used in ASP.

$$\leftarrow A_1, \dots, A_m, not\, A_{m+1}, \dots, not\, A_n. \tag{2}$$

$$\min\{A : A_1, \dots, A_m, not\, A_{m+1}, \dots, not\, A_n\} \max. \tag{3}$$

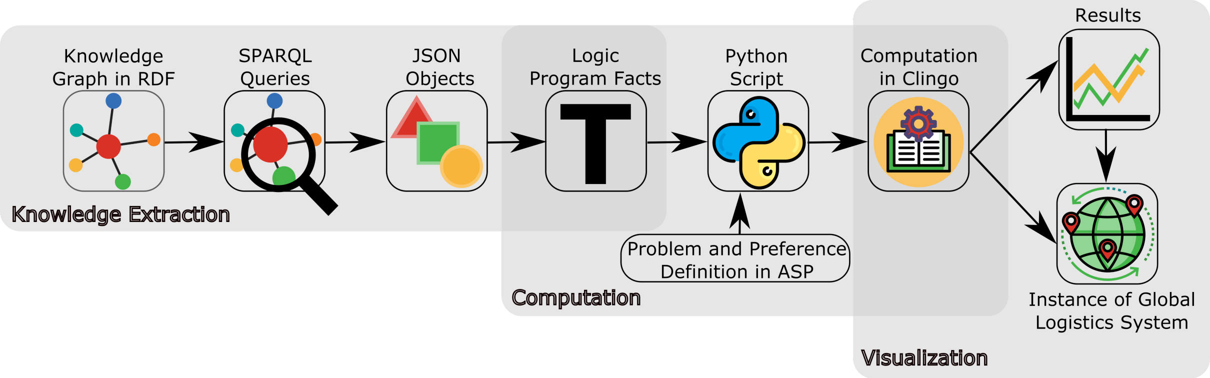


Figure 1: Knowledge extraction, computation and visualization for a global logistics framework.

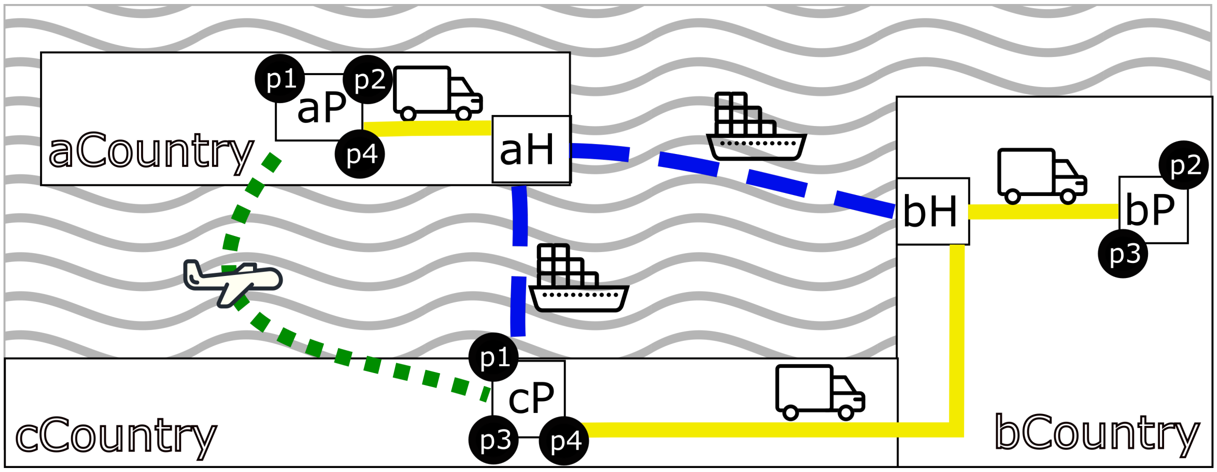


Figure 2: A logistics system where the gray waves, the black rectangles and the black squares represent a sea, countries and locations, respectively. The lines represent transport routes.

(2) and (3) are called *integrity constraint* and *choice rule*, respectively. Similar as above, A and A_i with $1 \leq i \leq n$ are atoms. Intuitively, an integrity constraint represents an undesirable situation, i.e. $A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n$ should be evaluated to *false*. *No location can be both a warehouse and a production location* can be expressed as $\leftarrow \mathsf{productionLoc}(X), \mathsf{warehouseLoc}(X)$. In choice rules, min and max are non-negative expressions, and $\{A : A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n\}$ denotes the set of all ground instantiations of A, governed through $\{A : A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n\}$. Intuitively, an interpretation satisfies a choice rule if $\min \leq N \leq \max$ holds, where N is the cardinality of any subset of $\{A : A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n\}$. *At least one location to be a production location* can be expressed as $1\{\mathsf{productionLoc}(X) : \mathsf{loc}(X)\}$.

3 Logistics in Action

Figure 1 shows the proposed framework: The first box on the left contains the information about the global logistics system as a knowledge graph in RDF on a triple store. Predefined queries specified in SPARQL allow the extraction of relevant data resulting in JavaScript Object Notation (JSON) objects that are transposed with JSONata for the conversion into logic program facts. The endpoints of an express server ingested through a Representational State Transfer (REST) interface are called by a Python script via MarkLogic to receive the facts from the knowledge graph. This Python script invokes the clingo ASP tool chain to compute the stable models. Finally, a graph shows a scatter-matrix according to different performance indicators of the computed models. Each model can be visualized as a global logistics configuration in a Python Dash Plotly environment.

The discussed concept of co-design development in Section 1 includes considering new variants of the industrial system for the product optimized towards some key performance indicators. Here, we mainly focus on the modeling part of the industrial system, i.e. the logistics facts and their requirements. The real industrial system is about the logistics of constructing an aircraft from multiple components. The goal of the task is to build a complete aircraft. Each component can be constructed from smaller components at some production site. These components need to be transported to the production sites where they can be further processed, possibly via warehouses. Several production sites exist in various countries and are reachable via various transport means. Not all production sites can construct all components, not all components can be transported with all transport means and not all production or warehouse sites can be reached by all transport means.

Because of the obligation of non-disclosure, we discuss here an adapted and simplified logistics system, which covers all relevant components that are necessary to compute configurations for the real industrial system: Figure 2 shows a fictional map. The production locations aP, bP and cP are located in aCountry, bCountry, and cCountry, respectively. The warehouse locations aH and bH are located in aCountry and bCountry, respectively. Existing transport means are plane, truck and ship. Possible transport means at each location and transport routes between each two locations are denoted by a line: A continuous yellow line denotes a truck route, a dashed blue line denotes a ship route and a dotted green line denotes a plane route. The black circles at the corner of the squares of the production locations denote which of the four parts (p1, p2, p3 and p4) is produceable at which site. The production plan that tells us the order in which the parts have to be assembled is as follows: p1 is the final product and needs p2 for its assembly. p3 is necessary for the production of p2 and in turn, p4 is necessary for the production of p3.

3.1 From Knowledge Graph to Logic Program Facts

The data for the global logistics scenario was modeled as an RDF knowledge graph. The implementation follows the technology stack presented in Section 2.1, The knowledge is scattered over multiple different sources is collected in manually created spreadsheets for the main bulk of the data but also captured as unstructured data in the form of documents. Additional assumptions of the system were not captured in any written form but were acquired via interviews with subject-matter experts for the industrial system. As a first step we modeled the ontology to capture all the the necessary knowledge.

In the next step we instantiated the concepts from the ontology with the help of collected data from the subject-matter experts. In order to minimize the workload we decided that not every fact had to be added to the knowledge graph but rather be created with the help of reasoning and SWRL rules whenever possible.

For example since the number of production locations in our industrial network can change, routes between these locations would have to be updated not only to the new location but also extended to all the already existing locations in our knowledge graph. This would mean an effort of creating and maintaining $n(n-1)/2$ for the n location nodes in the network ([27]). These efforts multiply if different transport means (like ships, trucks, airplanes) are taken into account, that all travel on different routes.

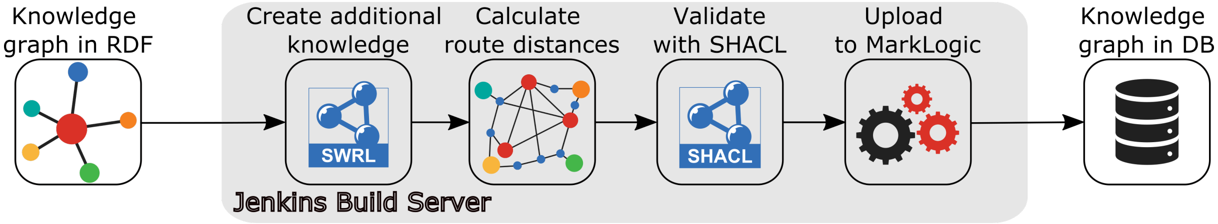


Figure 3: CI process for deploying the global logistics knowledge base.

In order to better reflect the actual industrial system — and to reduce the number of possible routes — the knowledge graph was extended so that some locations support restricted transportation means. For example locations without a harbor cannot handle ships. Additionally, certain transportation means are only available in certain geographic regions of the world. Further we implemented concepts for continental and intercontinental transport means to accurately model the SWRL rules for overseas transport routes. This had to be done due to the Open World Assumption used for reasoning.

Since the ontology has no means to accurately determine the distance between locations for each transport mean an attached routing service calculates them before uploading the knowledge graph into MarkLogic.

Finally, reasoning on the properties of the ontology was preformed to materialize additional facts. Utilizing the `owl:inverseOf` object property automatically creates the reverse connection between entities. The knowledge graph only indicates which transport means can carry certain parts (`can_transport`). The reasoner then materializes the inverse connection: which part can be transported by what transport mean (`is_transported_by`). This was implemented in order to create shortcuts in the knowledge graph to easier extract the data for the creation of the logic program facts.

The goal was to provide as much information prior to the grounding of the logic program. This reduced the facts that had to be derived in ASP and improved its performance for the model computation. It also provides a mechanism to perform consistency checks while modeling with the *Protégé* graphical editor.

The data for the ASP client was accessed through a MarkLogic data store (Section 2.1). In order to reduce the amount of work the knowledge graph developer has to put into the deployment in MarkLogic a Jenkins task ([5]) which is deployed to automatically push the data to the data store. This happens every time a new push to the git repository is detected by a webhook ([21]). Before the data is ultimately pushed into the store several scripts are being executed by Jenkins. First the different files are combined into a single file, for easier processing. Then all the SWRL rules are executed to materialize the additional facts. Next, the data is completed by accessing the REST service that calculated the distance between two locations. As a last step a process checks via predefined Shapes Constraint Language (SHACL) ([20]) shapes if all the entities in the knowledge graph conform to predefined shapes. If all the entities conform to the SHACL shapes they will be pushed into the data store. Figure 3 gives an overview over the process.

To extract the data from the data store we use the provided SPARQL endpoint of MarkLogic (Section 2.1). For each set of logic program facts a SPARQL SELECT query is implemented. SPARQL SELECT queries produce a tabular output which needs to be transformed into such facts so they can be consumed by the clingo client. For this we deploy a facade of REST web services with Node.js that uses the JSONata library ([19]) to convert the resulting tabular data from the SELECT query into logic program facts. Table 1 shows a mapping from the entities in the knowledge graph to the facts and gives

Entities in Knowledge Graph	Logic Program Fact Example	Description
Instance of `Country` class	country(aCountry).	aCountry is a country
Instance of `ProductionLocation` class and its sub-classes `Locations` that can manufacture `Products`	productionLoc(aP).	aP is a production location
Instance of `WarehouseLocation` classes and its sub-classes	warehouseLoc(bH).	bH is a warehouse location
Instance from `is_located_in` `Locations` in `Country`	locatedIn(aP, aCountry).	aP is located in aCountry
Entities that can transport parts via `TransportationResource`	transportMean(ship).	ship is a transport mean
via `has_terminal` and `can_handle` what `TransportationResource` accepted at location	transportMeanAtSite(bH, ship).	ship is a transport mean at bH
Correlated to instances of `Product` class and its sub-classes	part(part2).	part2 is a part to be produced
What `TransportationResource` can transport what `Product`	canTransport(ship, part2).	ship can transport part2
Via `ProductionResource` that links via `has_location` and `can_produce`	partProduceableAt(part2, aP).	part2 is produceable at aP
`Routes` via `has_destination`, `has_source` and `has_transport_mean`	transportRoute(aH, bH, ship, 7).	Distance from aH to bH by ship is 7
combination of `Product` instances and `has_part` object properties	productionPlan(part1, part2).	part2 necessary for production of part1

Table 1: One example for each type of fact that is extracted from the knowledge graph.

one example for each (predicate) type.[2]

In the next section we describe how the facts from the industrial system knowledge graph are used for the definition of requirements in ASP.

3.2 Representation of Logistics Requirements in ASP

Facts Eleven different (predicate) types of logic program facts with in total more than 70.000 facts, are extracted from the knowledge graph as described in Section 3.1.

Rules The assembly of the aircraft is specified by a production plan (last row in Table 1). For the example introduced at the beginning of this section, part1 is the final product (or root). The following

---

[2] In Figure 2 part is abbreviated with p.

rule defines this root:

$$\mathsf{root(X)} \leftarrow \mathsf{productionPlan(X,\_)}, \mathit{not}\ \mathsf{productionPlan(\_,X)}. \tag{4}$$

For all X, if X is a $\mathsf{productionLoc}$ or $\mathsf{warehouseLoc}$, then X is a $\mathsf{location}$:

$$\mathsf{location(X)} \leftarrow \mathsf{productionLoc(X)}. \qquad \mathsf{location(X)} \leftarrow \mathsf{warehouseLoc(X)}. \tag{5}$$

All $\mathsf{transportRoute}$ relations are symmetric:

$$\mathsf{transportRoute(From,To,TM,D)} \leftarrow \mathsf{transportRoute(To,From,TM,D)}. \tag{6}$$

All locations have an $\mathsf{intrasiteTransport}$ which can transport all parts with distance 0:

$$\mathsf{transportMeanAtSite(X,intrasiteTransport)} \leftarrow \mathsf{location(X)}. \tag{7}$$
$$\mathsf{canTransport(intrasiteTransport,X)} \leftarrow \mathsf{part(X)}. \tag{8}$$
$$\mathsf{transportRoute(X,X,intrasiteTransport,0)} \leftarrow \mathsf{location(X)}. \tag{9}$$

At all locations all parts can be transported intra-site with distance 0:

$$\mathsf{canBeTransportedFromTo}(X,X,\mathsf{Part},\mathsf{intrasiteTransport},0) \leftarrow \mathsf{location}(X),\mathsf{part(Part)}. \tag{10}$$

A part can be transported from one location to another location by a certain transport mean with a given distance if the following holds:

$$\begin{aligned}&\mathsf{canBeTransportedFromTo(From,To,Part,TM,D)} \leftarrow \mathsf{canTransport(TM,Part)},\\ &\quad \mathsf{transportMeanAtSite(From,TM)},\mathsf{transportMeanAtSite(To,TM)},\\ &\quad \mathsf{transportRoute(From,To,TM,D)}.\end{aligned} \tag{11}$$

A part can be transported from its original $\mathsf{location}$ (where it is produced) to its final $\mathsf{location}$ (where it is further assembled), directly, via one or via two locations:

$$\mathsf{direct(Part,From,To,TM,D)} \leftarrow \mathsf{canBeTransportedFromTo(From,To,Part,TM,D)}. \tag{12}$$

$$\begin{aligned}&\mathsf{via1(Part,From,(Via1,To),(TM1,TM2),D)} \leftarrow\\ &\quad \mathsf{canBeTransportedFromTo(From,Via1,Part,TM1,D1)},\mathsf{From!=To},\\ &\quad \mathsf{canBeTransportedFromTo(Via1,To,Part,TM2,D2)},\mathsf{D=D1+D2}.\end{aligned} \tag{13}$$

$$\begin{aligned}&\mathsf{via2(Part,From,((Via1,Via2),To),(TM1,TM2,TM3),D)} \leftarrow\\ &\quad \mathsf{canBeTransportedFromTo(From,Via1,Part,TM1,D1)},\mathsf{From!=To}\\ &\quad \mathsf{canBeTransportedFromTo(Via1,Via2,Part,TM2,D2)},\mathsf{Via1!=Via2},\\ &\quad \mathsf{canBeTransportedFromTo(Via2,To,Part,TM3,D3)},\mathsf{D=D1+D2+D3}.\end{aligned} \tag{14}$$

Choice Rules The requirements for the logistics system are specified through choice rules. Each part needs to be produced at some production location at which it is produceable (15) and each production location needs to produce at least one part (16):

$$1\{\mathsf{partProducedAt(Part,PL)} : \mathsf{partProduceableAt(Part,PL)}\}1 \leftarrow \mathsf{part(Part)}. \tag{15}$$
$$1\{\mathsf{partProducedAt(Part,PL)} : \mathsf{part(Part)}\} \leftarrow \mathsf{productionLoc(PL)}. \tag{16}$$

These choice rules lead ASP to guess $\mathsf{partProducedAt(Part,PL)}$ for each part and production location. After that, the transport paths and the transport means are guessed. If the part is the final product, i.e. the root part, then no further transportation is needed:

$$\begin{aligned}&1\{\mathsf{path}(\mathsf{Part},\mathsf{From},(),\mathsf{intrasiteTransport},0):\mathsf{partProducedAt}(\mathsf{Part},\mathsf{From})\}1\leftarrow\\&\quad\mathsf{partProducedAt}(\mathsf{Part},\mathsf{From}),\mathsf{root}(\mathsf{Part}).\end{aligned}\tag{17}$$

Otherwise, one of the other route types (direct, $\mathsf{via1}$ or $\mathsf{via2}$) applies:

$$\begin{aligned}&1\{\mathsf{path}(\mathsf{Part},\mathsf{From},\mathsf{To},\mathsf{TM},\mathsf{D}):\mathsf{direct}(\mathsf{Part},\mathsf{From},\mathsf{To},\mathsf{TM},\mathsf{D});\\&\quad\mathsf{path}(\mathsf{Part},\mathsf{From},(\mathsf{Via1},\mathsf{To}),(\mathsf{TM1},\mathsf{TM2}),\mathsf{D}):\\&\quad\mathsf{via1}(\mathsf{Part},\mathsf{From},(\mathsf{Via1},\mathsf{To}),(\mathsf{TM1},\mathsf{TM2}),\mathsf{D});\\&\quad\mathsf{path}(\mathsf{Part},\mathsf{From},((\mathsf{Via1},\mathsf{Via2}),\mathsf{To}),(\mathsf{TM1},\mathsf{TM2},\mathsf{TM3}),\mathsf{D}):\\&\quad\mathsf{via2}(\mathsf{Part},\mathsf{From},((\mathsf{Via1},\mathsf{Via2}),\mathsf{To}),(\mathsf{TM1},\mathsf{TM2},\mathsf{TM3}),\mathsf{D})\}1:-\\&\mathsf{partProducedAt}(\mathsf{Part},\mathsf{From}),\mathsf{productionPlan}(\mathsf{Super},\mathsf{Part}),\mathsf{partProducedAt}(\mathsf{Super},\mathsf{To}).\end{aligned}\tag{18}$$

Integrity Constraint So far, we have assumed a single sourcing strategy, i.e. every part is produced at exactly one location. The resilience of a logistics system can be strengthened by applying a multi sourcing strategy. For double sourcing, every part is required to be produced at two different locations, thus the min and max 1 constraints in choice rule (15) need to be replaced by 2. It is then desirable that these two locations are not located in the same country.

$$\begin{aligned}&\leftarrow\mathsf{part}(\mathsf{Part}),\mathsf{partProducedAt}(\mathsf{Part},\mathsf{Location1}),\mathsf{locatedIn}(\mathsf{Location1},\mathsf{Country}),\\&\mathsf{partProducedAt}(\mathsf{Part},\mathsf{Location2}),\mathsf{locatedIn}(\mathsf{Location2},\mathsf{Country}),\mathsf{Location1}!=\mathsf{Location2}.\end{aligned}\tag{19}$$

Optimization The next statement minimizes the overall distance of the transport paths:

$$\#\mathsf{minimize}\{\mathsf{D},\mathsf{part},\mathsf{From},\mathsf{To},\mathsf{TM}:\mathsf{path}(\mathsf{part},\mathsf{From},\mathsf{To},\mathsf{TM},\mathsf{D})\}.\tag{20}$$

Assertions In total we defined 25 assertions, which are rules that test if the extracted facts are correctly specified. Here are two examples: It should not be the case that a location is not located in a country (21) and it should not be the case that locations are located in different countries (22).

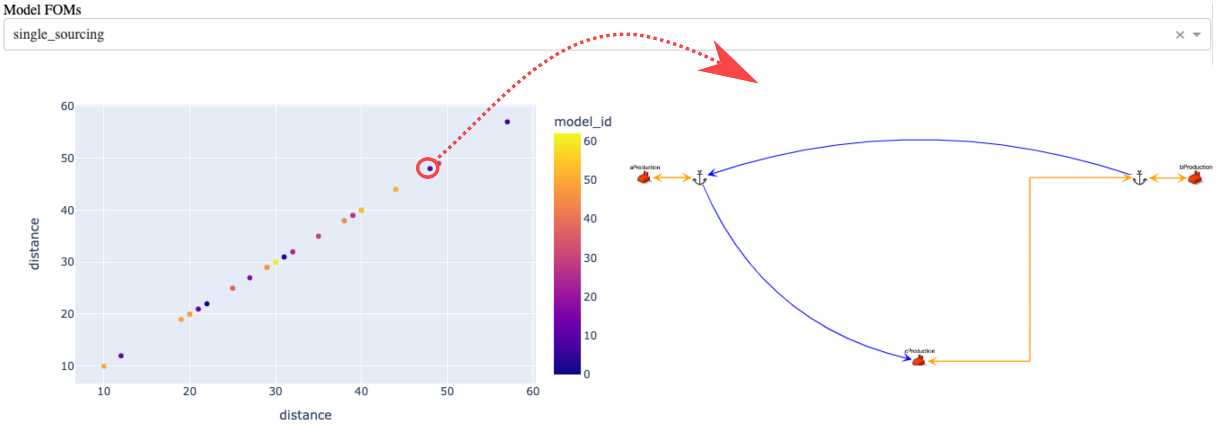
$$\mathsf{invalidLocatedIn}(\mathsf{X})\leftarrow\mathsf{location}(\mathsf{X}),\mathit{not}\ \mathsf{locatedIn}(\mathsf{X},\_).\tag{21}$$

$$\begin{aligned}&\mathsf{invalidLocatedInTwoCountries}(\mathsf{X})\leftarrow\mathsf{location}(\mathsf{X}),\mathsf{country}(\mathsf{Y1}),\mathsf{country}(\mathsf{Y2}),\\&\quad\mathsf{locatedIn}(\mathsf{X},\mathsf{Y1}),\mathsf{locatedIn}(\mathsf{X},\mathsf{Y2}),\mathsf{Y1}!=\mathsf{Y2}.\end{aligned}\tag{22}$$

These assertions could also be expressed as integrity constraints. However, in case an integrity constraint is violated, ASP would simply lead to UNSAT. The expression as assertion leads the head of the respective ‘invalid’ rule to be true in all models and allows us to immediately identify invalid specifications.

3.3 Visual Model Representation

A visual representation seems to be the simplest way for humans to communicate a model. Post-processing was done with the computed models by translating the relevant facts into .csv tables and



Model FOMs: single_sourcing

Figure 4: Overview of all models with respect to the total distance of necessary transportation.

one overview table showing key performance indicators. The graphical representation of the overview table and the individual configurations was done in a Python application using Plotly and Dash ([26, 29]). This toolkit produces an interactive representation of the already available .csv tables. Figure 4 shows an example for the simplified logistics system. The user interface is split into three parts: The drop-down menu at the top shows that ‘single_sourcing’ is selected. When selected, a scatter matrix appears (left). Each scatter plot represents the tradeoff between two indicators. As this example has only one performance indicator (‘distance’) it is simply plotted against itself. Each dot in the plot represents one computed model. The user can then hover on any dot in order to show details of one model (right), which consists of a map, showing production locations (red small factories) and warehouses locations (black small anchors). The line between these locations represent instantiated routes, where colors and shapes of the lines denote the mode of transport. Hovering over each route shows the part that is being transported. The red dotted arrow indicates that the red circled dot in the scatter matrix corresponds to the configuration shown in the map. This configuration shows a model with three indirect transport paths: part 4 is transported from cP to bP via bH (anchor next to bP), part 3 is transported from bP to aP via bH and aH and part 2 is transported from aP to cP via aH (anchor next to aP) where part 1 (root) is assembled into the final product.

4 Variations & Evaluation

Experiments on a 2.3 GHz Quad-Core Intel Core i7+16 GB RAM 3733 MHz type DDR4 with 29 locations (13 productionLoc facts and 16 productionLoc facts), 182 transportMeanAtSite facts and 34 part facts (including one root), show that variations in the encoding can have a strong impact on grounding and solving time: Rules (6) to (10) in Section 3.2 were replaced with almost 30.000 canBeTransportedFromTo facts directly derived from the ontology, omitting the instantiation of irrelevant rules. The second row in Table 2, Baseline, shows the resulting number of (choice) rules, times (in minutes) for grounding and finding the first model. The third row, PL Choice as IC, refers to the variation where choice rule (16) is replaced by the following rule and integrity constraint:

$$\mathsf{producedAtLoc}(\mathsf{PL}) \leftarrow \mathsf{productionLoc}(\mathsf{PL}), \mathsf{partProducedAt}(\mathsf{Part}, \mathsf{PL}). \tag{23}$$

$$\leftarrow \mathsf{productionLoc}(\mathsf{PL}), \mathit{not}\ \mathsf{producedAtLoc}(\mathsf{PL}). \tag{24}$$

LP Variation	# Rules (%)	# Choice Rules (%)	Grounding in min.	1st model in min.
Baseline	63148530 (100)	4475 (100)	14.47	5.67
PL Choice as IC	63142310 (99)	3098 (69)	14.24	5.42
Loc Type Req	2873012 (4.5)	4307 (96)	1.14	0.24
TM Type Req	1313162 (2)	3107 (69)	0.19	0.05
All	194129 (0.3)	2667 (60)	0.04	0.01

Table 2: Configurations wrt # (choice) rules, time for grounding and 1st model.

The fourth row, Loc Type Req, refers to the extension of the `via1` (13) and `via2` (14) rules by requiring `From`, `To` and `Via1`, `Via2` to be production locations and warehouse locations, respectively. As all warehouses in the current ontology are harbours, the transport mean between `via1` and `via2` can only be a ship. The results by extending the `via2` rule to require `TM2` to be a ship, i.e. `ship(TM2)`, is shown in the fifth row, TM Type Req. The last row, All, shows the results when all the replacements and extension were applied.

5 Conclusions, Lessons Learned & Future Work

The representation as RDF ontology and the computation of configurations in ASP seems promising for co-design development. With the help of the graphical Protégé editor it was easy to discuss the architecture, concepts and meta-data of the global industrial system with the industrial architects to crosscheck if we understood their provided constraints and data sets properly. For the first time we were able to devise a machine readable model of the global industrial system.

The construction of additional facts with the help from SWRL rules and reasoning was beneficial in two regards. Firstly, it allowed the creation of ancillary knowledge for the configuration of the logic programs. It also provided a mechanism to quickly adapt the product and industrial system to new requirements from the stakeholders. Secondly, some entities like valid `Routes` for the global logistics system and additional facts were generated automatically, which reduced the number of entities in the graph that had to be added or adapted considerably. Reasoning conducted under the Open World Assumption resulted in the creation of more concepts and properties to capture everything with the SWRL and reasoning rules. It was still manageable for our use case but could become a hindrance when scaling up the industrial system knowledge graph. Alternative approaches like the Ontological Modeling Language (OML), that automatically create additional statements to close the world, could be deployed ([7]). The CI pipeline implemented with Jenkins made the deployment of new iterations of the knowledge graph seamless. Through the git webhook the CI pipeline was always informed when new changes had to be deployed in MarkLogic. The SHACL shapes ensured that no invalid data was integrated into the knowledge graph. The knowledge graph could locally implement changes while the computational intensive tasks was handled by the Jenkins server. The computed models in ASP were validated by a fully fledged simulation environment in AnyLogic ([32]).

As Table 2 in Section 4 shows, adding *obvious* information to the rules accelerated the search process in ASP significantly: The extension of the `via1` (13) and `via2` (14) rules with a requirement on the location type (row 4, Loc Type Req) and a requirement on the transport mean type between warehouses (row 5, TM Type Req), reduced grounding from minutes to seconds. A surprising feature of ASP compared to other approaches that we investigated so far, is that it was very simple to specify partial configurations, when they were known (by simply adding them as facts) or modify requirements, as shown by the simple extension for the multi sourcing strategy in Section 3 (on page 236).

In the future, we intend to extract certain requirements directly from OWL, such that choice rules and integrity constraints can be automatically built from templates. Another aspect would be to produce a wider range of variations by means of the multishot feature in ASP. [8] suggest the fixed-domain semantics, an alternative formal semantics for an intuitive understanding in OWL and description logics that might be accessible for logically less skilled practitioners and provide a translation into ASP. Even though our purpose was to reduce the grounding time by deriving relevant facts from the ontology directly, their approach might be interesting for us.

Acknowledgments

Thanks to the anonymous reviewers for valuable feedback. It helped improving the paper.

References

[1] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux & Gavin Carothers (2014): *RDF 1.1 Turtle*. Https://www.w3.org/TR/turtle/.

[2] Barry B. Blake (1988): *Russell S. Tomlin, Basic word order. Functional principles. London: Croom Helm, 1986. Pp. 308.* Journal of Linguistics 24(1), p. 213–217, doi:10.1017/S0022226700011646.

[3] G. Brewka, T. Eiter & M. Truszczyński (2011): *Answer set programming at a glance*, doi:10.1145/2043174.2043195.

[4] Gerhard Brewka, James P. Delgrande, Javier Romero & Torsten Schaub (2015): *asprin: Customizing Answer Set Preferences without a Headache*. In Blai Bonet & Sven Koenig, editors: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA, AAAI Press, pp. 1467–1474, doi:10.1609/aaai.v29i1.9398.

[5] Continuous Delivery Foundation (2023): *Jenkins User Documentation*. Https://www.jenkins.io/doc/.

[6] Richard Cyganiak, DERI NUI Galway, David Wood, 2 Round Stones, Lanthaler MArkus, Graz University of Technology, Graham Klyne, Jeremy J. Carrol & Brian McBride (2014): *RDF 1.1 Concepts and Abstract Syntax*. Https://www.w3.org/TR/rdf11-concepts/.

[7] Maged Elaasar & Nicolas Rouquette (2022): *Ontological Modeling Language 1.4*. Http://www.opencaesar.io/oml/.

[8] Sarah Alice Gaggl, Sebastian Rudolph & Lukas Schweizer (2016): *Fixed-Domain Reasoning for Description Logics*. In Gal A. Kaminka, Maria Fox, Paolo Bouquet, Eyke Hüllermeier, Virginia Dignum, Frank Dignum & Frank van Harmelen, editors: Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI 2016), Frontiers in Artificial Intelligence and Applications 285, IOS Press, pp. 819 – 827, doi:10.3233/978-1-61499-672-9-819.

[9] Paula Gearon, Alexandre Passant & Axel Polleres (2013): *SPARQL 1.1 Update*. Https://www.w3.org/TR/sparql11-update/.

[10] M. Gebser, B. Kaufmann & T. Schaub (2012): *Conflict-Driven Answer Set Solving: From Theory to Practice* 187–188. doi:10.1016/j.artint.2012.04.001.

[11] Martin Gebser, Roland Kaminski, Benjamin Kaufmann & Torsten Schaub (2012): *Answer Set Solving in Practice*. Morgan & Claypool, doi:10.2200/S00457ED1V01Y201211AIM019.

[12] Michael Gelfond & Vladimir Lifschitz (1991): *Classical Negation in Logic Programs and Disjunctive Databases*, doi:10.1007/BF03037169.

[13] github (2023): *GitHub*. Https://github.com/.

[14] Thomas R. Gruber (1993): *A translation approach to portable ontology specifications*, doi:10.1006/knac.1993.1008.

[15] Steve Harris, Garlik & Andy Seaborne (2013): *SPARQL 1.1 Query Language*. Https://www.w3.org/TR/sparql11-query/.

[16] Ian Horrocks, Oliver Kutz & Ulrike Sattler (2006): *The Even More Irresistible SROIQ*. In: Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning, KR'06, AAAI Press, p. 57–67.

[17] Ian Horrocks, Peter Patel-Schneider, Harbold Boley, Said Tabet, Benjamin Grosof & Mike Dean (2004): *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. Https://www.w3.org/Submission/SWRL/.

[18] Tomi Janhunen & Ilkka Nimelä (2016): *The Answer Set Programming Paradigm*, doi:10.1609/aimag.v37i3.2671.

[19] JSONata.org (2021): *JSONata Documentation*. Http://docs.jsonata.org/overview.html.

[20] Holger Knublauch, TopQuadrant, Inc., Dimitris Kontokostas & University of Leipzig (2017): *Shapes Constraint Language (SHACL)*. Https://www.w3.org/TR/shacl/.

[21] Jeff Lindsay (2007): *Web hooks to revolutionize the web*. Https://progrium.github.io/blog/2007/05/03/web-hooks-to-revolutionize-the-web/.

[22] MarkLogic (2019): *Semantic Graph Developer's Guide*. Https://docs.marklogic.com/guide/semantics.

[23] Val Mitchell, Tracy Ross, Andrew May, Ruth Sims & Christopher Parker (2015): *Empirical investigation of the impact of using co-design methods when generating proposals for sustainable travel solutions*, doi:10.1080/15710882.2015.1091894.

[24] Ilkka Niemelä (1999): *Logic programs with stable model semantics as a constraint programming paradigm*, doi:10.1023/A:1018930122475.

[25] OMG (2019): *Semantics Of Business Vocabulary And Business Rules*. Https://www.omg.org/spec/SBVR/1.5/About-SBVR/.

[26] Plotly Technologies Inc. (2015): *Collaborative data science*. Https://plot.ly.

[27] J.P. Rodrigue, C. Comtois & B. Slack (2017): *The Geography of Transport Systems*. Routledge, doi:10.4324/9780429346323. Available at `https://books.google.de/books?id=qimgjwEACAAJ`.

[28] Torsten Schaub & the Potassco Team (2022): *Potassco, the Potsdam Answer Set Solving Collection*. `https://potassco.org/`.

[29] Carson Sievert (2020): *Interactive Web-Based Data Visualization with R, plotly, and shiny*. Chapman and Hall/CRC, doi:10.1201/9780429447273. Https://plotly-r.com.

[30] Patrik Simons, Ilkka Niemelä & Timo Soininen (2002): *Extending and implementing the stable model semantics*. Artificial Intelligence 138, pp. 181–234, doi:10.1016/S0004-3702(02)00187-X.

[31] Stanford University (2020): *Protégé a free, open-source ontology editor and framework for building intelligent systems*. Https://protege.stanford.edu.

[32] The AnyLogic Company (2023): *AnyLogic Documentation*. Https://anylogic.help/.

[33] W3C OWL Working Group (2012): *OWL 2 Web Ontology Language Document Overview (Second Edition)*. Https://www.w3.org/TR/owl2-overview/.

The *Janus* System: Multi-paradigm Programming in Prolog and Python

Theresa Swift
Johns Hopkins Applied Physics Lab
theresasturn@gmail.com

Carl Andersen
carl.andersen@gmail.com

Python and Prolog express different programming paradigms, with different strengths. Python is wildly popular because it is well-structured, easy to use, and mixes well with thousands of scientific and machine learning programs written in C. Prolog's logic-based approach provides powerful reasoning capabilities, especially when combined with constraint evaluation, probabilistic reasoning, well-founded negation, and other advances. Both languages have commonalities as well: both are usually written in C, both are dynamically typed, and both use data structures based on a small number of recursive types.

This paper describes the design and implementation of *Janus*, a system that tightly combines Prolog and Python into a single process. *Janus* bi-translates data structures and offers performance of many hundreds of thousands of round-trip inter-language calls per second. Although *Janus* is still new, it has been used in commercial applications including natural language processing, visual query answering and robotic automation. *Janus* was developed for XSB, but porting *Janus* code to a second Prolog has been straightforward, indicating that *Janus* is a tool that other Prologs may easily adopt.

1 Introduction

Modern Prolog systems offer a rich array of functionality that reflects a strong research community. There are well over a dozen Prolog systems that are actively developed and/or maintained, including SWI([15]), SICStus ([3]), Ciao ([7]), YAP ([11]), ECLIPSe ([12]), GNU Prolog ([4]), Picat ([16]), Trealla, [1] Tau, [2] and XSB ([13]). Most of these Prologs are compliant with Prolog's ISO standard ([8]), but also offer diverse functionality far exceeding the standard.[3] We offer several examples. Picat, SICStus, and ECLIPSe are excellent for constraint-evaluation; Ciao is particularly strong for program analysis and a flexible syntax; YAP is especially fast for SLDNF; SWI offers a fast and stable multi-threaded engine; and XSB has pioneered a wide array of tabling strategies, many of which are now offered by other Prologs.

Altogether, the dialects of these modern Prolog systems offer numerous useful features that are either unavailable or rarely available in other programming languages. Despite these features the author's experience is that Prolog is rarely used in industry – certainly not as often as it might be. Perhaps the major factor behind this is the lack of programmers trained in Prolog, but there are also issues of usability. The first of these is embeddability. Many Prologs can be called from C or Java, but fewer can be called from other languages popular in industry such as Python or C#. Fewer still can be called from emerging languages such as Rust or Julia.[4] While the above problems are more acute for research-oriented Prologs like XSB than for more heavily used Prologs like SWI or SICStus, the paucity of packages can make

---

[1] https://github.com/trealla-prolog/trealla

[2] http://tau-prolog.org

[3] Picat has a non-ISO syntax, but it is based on the B-Prolog engine ([17]).

[4] Scryer is an emerging Prolog implemented in Rust.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 241–255, doi:10.4204/EPTCS.385.24

development in any Prolog uneconomical. Languages like Java, Python and Javascript each have package repositories with well over 100,000 packages, perhaps two orders of magnitude more than the total packages available from currently maintained Prologs. As a result, the use of Prolog in a commercial system may be limited to a few modules – or most commonly, avoided altogether.

The *Janus* system combines Prolog and Python into a single system running in a single process, with development possible from either the Prolog or the Python command line (or IDE). Because Prolog and Python are tightly connected, round trip calls between languages can be made many hundreds of thousands of times per second, so that there is little need to implement functionality in one language if it is present in the other.

Several features of Python make it a productive choice of language for Prolog integration. Python is currently one of the most popular of all programming languages both for commercial and scientific computing, with large package managers that are easy to use. Like most Prologs, the most popular version of Python, CPython is written in C, with a well-written C-API.[5] Furthermore, Python is dynamically typed, as is Prolog. Nearly all of Python's data structures are recursively formed using lists, tuples, sets, and dictionaries; as discussed in Section 2 these data structures map well into Prolog's logical terms.

In addition to establishing a tight, fast connection between Python and Prolog, the authors put substantial effort towards making *Janus* 's configuration as easy as possible. *Janus* was originally developed for XSB and its first version is available as part of the XSB distribution. XSB's installation process generates code enabling either Python and Prolog to load the other language on demand. Alternately, a test site on the `PyPI` repository allows *Janus* and XSB to be installed via a Python `pip` command.[6] Once *Janus* is installed, Prolog is at a Python programmer's fingertips, and vice versa.

In the less than a year since *Janus* ' release, it has been used in commercial applications including reasoning over large knowledge graphs, natural language processing, visual query answering, and situational awareness of autonomous agents. Furthermore, our experience in porting *Janus* to `LVM` Prolog developed by Permion, Inc.[7] has shown that *Janus* is easy for a developer to port. [8]

This paper focuses on the design and architecture of *Janus*. First, Section 2 describes the design of the Prolog and Python APIs of *Janus* . Section 3 describes implementation of the heart of *Janus* : its bi-translation between Python and Prolog data structures. This section also discusses implementation of calls in the *Janus* interface as well as *Janus* performance. Section 5 lists applications using *Janus* and describes in detail the implementation of one of a dozen or so libraries that use *Janus*.[9]

1.1 Related Work

Most of the Prologs mentioned above are written largely in C and so offer bi-directional C/C++ interfaces. Many also offer bi-directional Java interfaces using JPL (SWI and YAP), Jasper (SICStus), Interprolog ([2]) (XSB) and other frameworks (including Ciao, YAP, ECLIPSe and GNU Prolog). Further, some Prologs also have Python interfaces (e.g, SWI), Javascript interfaces (e.g., Tau, Ciao) or Rust interfaces (Scryer). In addition, the Natlog interpreter ([14]), written in Python provides a form Prolog-style evaluation within Python.

Three factors distinguish the *Janus* interfaces and how they are used. First, *Janus* tightly combines

---

[5] `https://docs.python.org/3/c-api/index.html`

[6] Installation via the `PyPi` site currently works only for Linux.

[7] `http://permion.ai`

[8] Although all functionality described in this is available in XSB, its first version used non-generic `xsbpy` and `px` within function and module names. These names are in the process of being changed.

[9] These applications are described in more detail in a companion paper about *Janus* applications [1].

Prolog and Python into a single process. As will be discussed in subsequent sections, the result is that *Janus* is extremely fast. Second, its automatic bi-translation (Section 3.1) allows large and complex data structures to be copied from one language to another without requiring special declarations. Third, compiling and linking is handled by the Janus configuration process, allowing transparent, on-demand compilation for both Prolog and Python modules, and on-demand loading of Python modules when called by Prolog. These features are fully supported for Python and Prolog modules even when they call other languages, such as C.

2 The *Janus* Programmer's API

This section describes the bi-directional *Janus* API.

2.1 Calling Python from Prolog

Janus is consulted (loaded) by Prolog just as any other module (currently, XSB and `LVM` Prolog are supported). When *Janus* is loaded, Python is also loaded and initialized within the same process as Prolog; paths to *Janus* and its subdirectories are added both to the library directories of Python (`sys.path`) and Prolog; and core Prolog modules are loaded. Subsequent calls from Prolog cause the Python interpreter to search for its modules and packages in the same manner as if Python were stand-alone.

We introduce further *Janus* functionality through a series of examples.

Calling Python Functions The *Janus* Prolog predicates `pyfunc/3` and `pyfunc/4` are used to call Python functions. We invoke the Python standard library `json`, which reads and writes JSON strings, to illustrate Python calls via `pyfunc/3`. The Python function `json.loads(string)` inputs a JSON-formatted string, and returns a Python dictionary. A simple call of `json.loads()` from Prolog and subsequent write has the form:

```
?- pyfunc(json,loads(’{"name":"Bob", "langs":["English", "GERMAN"]}’),Dict),
   writeq(Dict).
```

and prints out:

```
pyDict([’’(name,’Bob’),’’(languages,[’English’,’GERMAN’])])
```

Here, Prolog invokes the Python call

```
json.loads(’{"name":"Bob", "langs":["English", "GERMAN"]}’)
```

which parses the string and outputs a Python dictionary, which is then translated into a Prolog term and bound to the `Dict` variable.[10]

More specifically, *Janus* translates a Python dictionary (a set of key-value pairs) to a Prolog term `pyDict(List)`, where `List` is a list of key-value tuples. *Janus* represents each such tuple as a Prolog term `''(Key,Value)`, in which the functor `''` (the empty string) is chosen to make the pairs resemble Python tuples.[11] Because Python-Prolog translations can be complex, multi-level Prolog terms, *Janus* provides Prolog tools to traverse these data structures that are analogous to the Python methods `values` and `keys()`.

A slightly different call shows *Janus*' support for Python keyword arguments.

---

[10] Any errors encountered in the Python execution are caught and wrapped in a Prolog exception.

[11] The choice of functor `''` used for tuples is configurable, as are most *Janus* functors.

```
?- pyfunc(json,loads('{"name":"Bob", "langs":["English","GERMAN"]}'),pyDict(L)),
   append(L,[''(gpa,3.5)],L1),
   pyfunc(json,dumps(pyDict(L1)),[sort_keys=1],NewString).
```

Here, the call to `json.dumps()` is made via the Prolog predicate `pyfunc/4`, which takes an additional parameter specifying the optional Python keyword parameter `sort_keys`. In this case, the output is sorted by key, producing

```
{"gpa": 3.5, "languages": ["English", "GERMAN"], "name": "Bob"}
```

Using Python Objects and Methods *Janus* also supports the use of Python class objects and methods. We demonstrate these capabilities via calls to the `fasttext` Python package for Natural Language Processing, specifically its `lid.176.bin` language identification model ([5]). The model is loaded by calling an initializer directly from the `fasttext` package:

```
pyfunc(fasttext,load_model('./lid.176.bin'),Obj).
```

This call unifies `Obj` with a Prolog reference to the (Python) model object, represented for this example as `pyObj(p0x7faca3428510)`. Using the reference, the object's class methods can be called, or its class attributes obtained, via `pydot/[3,4]`. For example, a call to the class method `predict`:

```
pydot(pyObj(p0x7faca3428510),predict(
       'Janus is a really useful addition to Prolog! But language detection
        requires a longer string than I usually want to type.'),Lang).
```

returns the detected language and confidence value.

In the preceding examples, once *Janus* was consulted, Python functions and methods were immediately available to Prolog. In practice, it is sometimes convenient to write libraries containing small amounts of Prolog and Python wrapper code. For instance, the `jns_rdflib.P` library transforms rdf triples, which are maintained as objects in the Python `rdflib` library, into Prolog terms that can be asserted as facts. In other cases, it may make sense to provide an access from Prolog that is a bit higher level than that provided by Python functions or methods, as will be illustrated in Section 5.

2.2 Calling Prolog from Python

When used from Python, *Janus* is loaded just like any other Python package. When *Janus* is loaded into a Python session, the XSB executable and its core libraries are loaded and initialized, and XSB's paths configured. For convenience, Python functions for compiling and consulting user Prolog code are also provided, as in the following invocation of Prolog's `consult`:

```
>>> import janus
>>> consult(prolog_file)
```

When Prolog calls Python, a goal of *Janus* is to make the Python data structures and calls as natural as possible for a Prolog programmer. Similarly, when Python calls Prolog, the goal is to make Prolog data structures and calls as "Pythonic" as possible. Several Prolog features make a Pythonic translation challenging. Python has no analog of Prolog's logical variables, nor a direct analog to the named functors of Prolog terms.[12] Additionally, the non-determinism of Prolog predicates must be accounted for; and in

---

[12] Using Python's `namedtuple` library, classes can be created whose instances syntactically resemble terms with named functors, but both a class and its instances have to be specifically instantiated to be used.

Prologs like XSB and SWI that support the well-founded semantics (WFS), each answer must must be associated with its truth value, which may be *true* or *undefined*.

Prolog Deterministic Goals We first describe how Prolog deterministic goals (goals returning exactly one answer) can be called via the *Janus* Python function `jns_qdet()`. Specifically, consider a call from Python to reverse a list:

```
>>> (Answer,TV) = jns_qdet('basics','reverse',[1,2,3,('mytuple'),{'a':{'b':'c'}}])
```

which calls the Prolog goal

```
basics:reverse([1,2,3,('mytuple'),{a:{b:c}}],Return)
```

The first two parameters of `jns_qdet()` are the module (`basics`) and the name (`reverse`) of the Prolog predicate to be called . The remaining parameters are a (possibly empty) sequence of ground arguments that are passed to the Prolog predicate: in this case only one argument is passed, the list to be reversed. The Prolog predicate that is called must have one additional argument not included in the Python call: in this case it is the second argument of `reverse/2` bound to the variable `Return`.[13] Here, the call returns to Python the tuple `(Answer, TV)`, in which `Answer` is the binding of the Return variable (the reversed list) and `TV` is the call's returned truth value (here, `1`). The *Janus* convention is that a truth value in Python may be `1` representing *true*, `0` representing *false*, or 2 representing *undefined*. (If a goal fails, `None` is returned for the answer).[14] `jns_qdet()` is designed to be extremely fast, and so is suitable for uses like calling Prolog as part of a neuro-symbolic loss function for machine learning.

Prolog Non-deterministic Goals Non-deterministic Prolog goals can be called via the *Janus* Python function `jns_comp()`, which uses the same parameters as `jns_qdet()` but also offers optional keyword parameters. We introduce a small Prolog knowledge base, `jns_test` to illustrate the behavior of `jns_comp()`:

```
test1(a,b,1).                      test1(a,c,2).
test1(a,d,5):- unk(something).     unk(X):- tnot(unk(X)).
```

If `jns_test` is already loaded in Prolog, then the Python call:

```
>>>  jns_comp('jns_test','test1','a',vars=2)
```

calls the Prolog goal:

```
?- jns_test:test1(a,V1,V2).
```

The keyword argument `vars=2` causes a Prolog goal to be formed with two free variables at the end. The call returns a list of answer bindings to `V1` and `V2`, each of which has a distinct truth value:

```
[((b,1),1), ((c,2),1), ((d,5),2) ]
```

The answer tuple `((d,5),2)` has an *undefined* truth value because it is proved via the subgoal

```
unk(something)
```

---

[13] A function `jns_cmd()` handles similarly restricted goals where no answer binding needs to be returned.

[14] The version of *Janus* using XSB does not use the Python keywords `True` and `False` since three truth values are needed and it was thought best for all truth values to have the same class.

which is *undefined* in WFS. Under the hood, XSB's tabled evaluation of WFS semantics maintains a *delay list* for each answer that tracks potential proofs of the answer that are delayed until their undefined dependencies are resolved. In this case, the answer `(d,5)` has a non-empty delay list (i.e. its undefined dependency is never resolved) [13].

`jns_comp()` has other useful keyword arguments. The keyword argument `comprehension=set` returns the answers as a set instead of a (default) list. The keyword argument `truth_vals=no_truthvals` drops the truth value from the answer tuple. Alternately, `truth_vals=delay_lists` causes the full delay lists of answers to be returned.

Janus also supports an alternative means of calling complex Prolog goals using a Python string to hold the arguments of the goal. The Python string is first translated to a Prolog atom, and then the argument terms are read from the atom. For example, a system of numerical constraints could be invoked through a call such as

```
>>> jns_cmnd('jns_constraints','check_entailed','[[X  > 3*Y + 2,Y>0],[X > Y]]')
```

Here, the predicate `check_entailed/1` reads the input atom, sets up its constraints, and calls a constraint evaluator such as `clpr`.[15] In fact, a *Janus* -based interface from Logtalk ([10]) expands on this approach by returning logical variables and their bindings to Python in a dictionary.

Because `jns_comp()` returns a list or set, it can be used in Python anywhere an iterable object can be used, for instance in the Python code fragment:

```
>>> for answer in jns_comp(...):
```

Although iterable objects are arguably central to Pythonic programming, the current *Janus* implementation constructs answer lists/sets eagerly, returning all answers at once, which can cause performance issues when answer sets are very large. A more incremental alternative would make use of Python's `generator` type, a cursor-like mechanism through which answers can be returned lazily via Python's `yield` statement. This approach is in fact taken by the Natlog interpreter, written in Python ([14]). However, multiple generators cannot directly be supported by a single-threaded Prolog engine since this corresponds to multiple top-level backtracking points. Future versions of *Janus* may provide support for *table cursors*, using a lightweight mechanism to backtrack through completed tables.[16]

Finally, *Janus* also supports round-trip callbacks from Python to Prolog to Python, enabling flexible combinations of the two languages' capabilities and ecosystems. For example, Prolog reasoning could be leveraged by Python-based graphical interface libraries like `Tkinter`, web interfaces libraries like `django`, or IDE plug-ins. Here, round-trip callbacks enable both languages to participate in the orchestration of the overall program.

3 Implementation and Performance

The *Janus* code base is small: approximately 1500 lines of C code and 750 lines Prolog code, apart from that used for configuration, which has made it straightforward to port. It is also designed to be as fast as possible. In this section, we first describe the implementation of bi-translation, followed by the implementation of sample *Janus* predicates and functions. ¡

---

[15] See Volume 2, Section 18.2.4 of the XSB manual.

[16] Multi-threaded Prologs can support multiple backtrack points, but with a likely space overhead, since a Prolog thread requires more memory than a cursor.

3.1 Bi-Translation of Data Structures

Bi-translation is at the heart of *Janus*, which offers one translator to traverse a Prolog term to create a corresponding object on the Python heap; and another to traverse a Python object to create a term on the Prolog heap.

Translating a Prolog Term to a Python Object. Prolog to Python translation is performed by the function `translate_prTerm_pyObj()` which recursively traverses a Prolog term to convert atoms, integers, floats, and lists to their Python form. The function also uses translation conventions described earlier, such as translating terms having the `''/n` outer functor to tuples, `pySet/1` to sets, and `pyDict/1` to dictionaries. A domain error is thrown if a term is encountered having some other outer functor.

Figure 1 shows a fragment of C-like pseudo-code for the translation of `pyDict/1` terms (via the macro `PYDICT_C`). We note several points. First, the fragment begins with the `else if`' conditional since `translate_prTerm_pyObj()` is structured as a series of conditions rather than as a switch statement. This choice was made because the number of possible checks is small, and the conditions can be ordered by expected frequency. Conditions are ordered to first check for integers, atoms, floats and then lists. If the term is a structure, checks are made for tuples, dictionaries and finally sets.

Second, the code shown largely consists of Python C-API calls that create and assemble Python objects (`PyDIct_New()` and `PyDict_SetItem()`) along with calls that interact with the Prolog heap. In the latter category, (`get_functor_string()` returns the functor of a term as a string, `is_list()`, `get_arg()` returns the nth argument of a structure, while `get_car()` and `get_cdr()` return the head and tail of a list). When *Janus* is ported to another Prolog, only the calls that interact with the Prolog heap need to be redefined. (E.g., by using macros to redefine functions like `get_arg()` to corresponding functions in the target Prolog.)

Third, although `translate_prTerm_pyObj()` is recursive, recursion depth has never been a problem in practice, in part because iteration is used to traverse through the top-level elements of data structures as in the while statement of Figure 1. Furthermore, any Prolog term traversed must correspond to a Python data structures and in our experience Python data structures like dictionaries or tuples are rarely nested with depth of more than a couple dozen. Instead, lists and sets may have a large number of elements, and dictionaries a large number of associations at a given level.

```
else if (get_functor_string(prTerm) == PYDICT_C)
   PyObject pykey, pyval;
   prolog_term list, elt;
   list = get_arg(prTerm, 1);
   pydict = PyDict_New();
   while (is_list(list))
      elt = get_car(list);
      pykey = translate_prTerm_pyObj(get_arg(elt,1));
      pyval = translate_prTerm_pyObj(get_arg(elt,2));
      PyDict_SetItem(pydict,pykey,pyval);
      list = get_cdr(list);
   Pyobj = pydict;
   return TRUE;
```

Figure 1: Pseudo-C code for fragment of `translate_prTerm_pyObj()`

```
else if(PyDict_Check(pyObj))
   prolog_term head, tail;
   prolog_term P = new_heap_cell();
   create_beap_functor(PYDICT_C,1,P);
   tail = get_arg(P, 1);
   PyObject* tup;
   PyObject** key, value;
   Py_size_t* pos = 0;
   while ( PyDict_Next(pyObj, pos, key, value))
      create_beap_list(tail);
      head = get_car(tail);
      tup = PyTuple_New(2);
      PyTuple_SET_ITEM(tup,0,key);
      PyTuple_SET_ITEM(tup,1,value);
      head = translate_pyObj_prTerm(tup);
      tail = get_heap_cdr(tail);
   set_heap_nil(tail);
```

Figure 2: Pseudo-C code for fragment of `translate_pyObj_prTerm()`

For memory management, Python maintains reference counts of all objects, and objects with a positive reference count cannot be garbage collected. The block of code in Figure 3.1 does not need to explicitly decrement Python object reference counts (via `Py_DECREF()`) since object references are automatically decremented when control exits the end of a code block.[17]

Translating a Python Object to a Prolog Term. Figure 2 shows a fragment of

`translate_pyObj_prTerm()`

which builds Prolog `pyDict/1` terms from Python dictionaries. As when translating Prolog to Python, `translate_pyObj_prTerm()` is structured as an ordered series of conditions, where some conditions call the function recursively. The code first sets up the outer `pyDict` functor, and then builds a list in its first argument, iterating through the dictionary via the Python C API function `PyDict_Next()` that sets *key* and *value* point to the key and value, respectively, of each association. For convenience, a tuple is built with the *key* and *value* variables, and `translate_pyObj_to_prTerm()` called recursively.

One key difference of Python from Prolog is Python's support for complex classes and class instances. To translate these to Prolog, a Python object reference is built and returned to Prolog if the type of a Python object is not otherwise handled (e.g., for Python binary objects or complex numbers).

3.2 Implementation of *Janus* Calls

This section describes at a high level the implementation of *Janus* calls using `pyfunc/[3,4]` and `jns_comp()` as representative examples.

---

[17] `https://docs.python.org/3/c-api/refcounting.html`

```
1) int pyfunc(module,prologTern)
   PyErr_Clear()
   modulePtr = PyImport_ImportModule(module)
   if modulePtr is null throw janus_error/* module cannot be found */
5) Obtain functor and arity from prologTerm
   funcPtr = PyObject_GetAttrString(modulePtr, functor)
   if(not PyCallable_Check(funcPtr)) throw janus_error
   funcTuple = PyTuple_New(arity)
   for(int i = 1; i <= arity; i++)
10)    Set the translated $i^{th}$ argument of funcTuple to the $i^{th}$ argument of prologTerm
   Set up dictionary of keyword arguments if called by pyfunc/4
   retValue = PyObject_CallObject(funcPtr, funcTuple,dictionary)
   if (retValue == NULL) throw janus_error
   return translate_pyobj_prTerm(retValue)
```

Figure 3: Pseudo-C code for `pyfunc/[3,4]`

Calling Python from Prolog: pyfunc/[3,4]. Figure 3 provides pseudo-code for `pyfunc/[3,4]` (i.e., with and without keyword arguments). Much of the work is done by `translate_prTerm_to_pyObj()` to translate the call, and `translate_pyObj_to_prTerm()` to return the results. The pseudo-code provides a taste of how Python's C API is used. The Prolog module, functor name, and (if applicable) the list of keyword arguments are translated to Python objects via C API calls. The module loading in lines 2-3 of Figure 3.2 has the benefit that Python modules are loaded *on-demand*: if the goal `pyfunc(`*module*`,...)` is called, *module* will be loaded if necessary without needing any special import declarations. Note that since Python functions are variadic, the function name can be resolved in line 6 without specifying its cardinality. To properly handle errors, the Python error buffer is cleared in line 1; any Python error "caught" in line 13 is re-thrown as a Prolog error containing the Python error message and backtrace in addition to the Prolog message and backtrace.

Calling Prolog from Python: jns_comp. The implementation of a call

```
jns_comp(module,functor,arg_1,...,arg_n,varnum=m)
```

consists of a C function and a Prolog predicate. First the C function creates the goal

```
jns_comp(PrologGoal,Flag,Comp)
```

on the Prolog heap, where `PrologGoal` is the goal

```
module:functor(arg_1,...,arg_n,var_1,...var_m)
```

created from the arguments of `jns_comp()` as described in Section 2.2. Here, `Flag` is an integer encoding of keyword arguments, and `Comp` is unified with the list or set constructed by Prolog. Due to space constraints, we present in detail only the Prolog predicate `jns_comp/3`.

Since `jns_comp()` is treated as a list or set comprehension in Python, the Prolog answers must be gathered into a list or set. As shown in Figure 4, if no truth values are needed, this can be done directly through `findall/3`. In this case each answer tuple (Section 2.2) consists solely of answer bindings. Within the findall, after each success of `PrologGoal` with substitution θ, the answer bindings must be extracted from the last *varnum* arguments of `PrologGoal`θ.

```
1) jns_comp(PrologGoal,Flag,Result)     /* Pseudo-Prolog code */
   If no_truthvals(Flag)
        findall(Binding,(PrologGoal,strip_bindings(PrologGoal,Flag,Binding),InterimResult)
   Else if tabled(PrologGoal)
5)      findall(Binding,(PrologGoal,table_bindings(PrologGoal,Flag, Binding),InterimResult)
   Else abolish_table_pred(jns_table_goal(_))
        findall(AnswerTuple,(jns_table_goal(PrologGoal),
                              table_bindings(PrologGoal,Flag,AnswerTuple),InterimResult)
   If set_comp(Flag) Result = pySet(sort(InterimResult)) else Result = InterimResult
```

Figure 4: Pseudo-Prolog code for `jns_comp()`

If truth values are to be returned along with the answer bindings, tabled calls may need to be used to retrieve undefined results. If `PrologGoal` isn't a tabled predicate, the findall calls `PrologGoal` through the tabled meta-call `jns_table_goal/1`, otherwise `PrologGoal` can be called directly. In lines 5 and 7 the findall first calls `PrologGoal` and then backtracks through the completed table using the predicate `get_residual/2` which binds `PrologGoal` with each answer substitution θ and returns the answer's delay list. Answer bindings are taken from `PrologGoal`θ as in the non-tabled case. The answer tuple is constructed from these bindings along with either a numeric truth value or the delay list of the answer. If set comprehension is required, the list of tuples is sorted and then clothed in `pySet/1` which signals to the translation that a set is to be returned to Python.

4 Performance

Initially, *Janus* was coded using the Python `ctypes` package, which did not provide adequate performance, so Python's lowest level C-API was used instead. Table 1 shows the average times for both intra- and inter-language calls.[18] As an aside, for standard Prolog benchmarks the performance of XSB is neither the fastest nor the slowest of the Prologs listed in Section 1.

Table 1 compares the time within and between each language. The first row shows the time per iteration of a simple loop that calls a function or predicate to decrement a counter. The second row calls the haversine function, [19] a small function that makes multiple trigonometric and exponentiation calls. The third row tests creation of 20 element lists, via list comprehension based on a set of 20 elements in Python and `findall/3` calling a predicate in XSB. From these timings, XSB is generally faster than native Python, except for list comprehension/findall: this is likely due to the difference between calling a predicate and reading a list from a set. Table 1 also shows that for Python calling XSB, while there is an overhead for inter-language calls, this overhead is still small: many hundreds of thousands of round-trip Prolog/Python calls can be made per second. Even for the small haversine function, Prolog calling Python is only about 3x slower than a Python-only loop.

Python calling XSB is slower than XSB calling Python. For the first two rows, timings are reported for `jns_qdet()` (the timings for `jns_cmd()` are nearly the same). The third row uses `jns_comp()` with

---

[18] All benchmarks were run within a VM on a server with an AMD EPYC 7542 32-Core Processor running Ubuntu 20.04, using Python 3.9.16 and the repo version of XSB of February, 2023, available at `xsb.sourceforge.net`. The smallest time over three runs was taken for each benchmark, but since these tests were done on a shared server all times should be regarded as approximate. Benchmark tests used in this paper are available from the XSB repo version under `xsbtests/python_tests`.

[19] `https://en.wikipedia.org/wiki/Haversine_formula`

	XSB Only	Python Only	XSB/Python	Python/XSB
Simple Loop	30 ns	78 ns	1.98 μs	79 μs
Haversine	675 ns	1.5 μs	5.07 μs	81 μs
List Comp.	92 μs	586 ns	5.69 μs	584 μs

Table 1: Timings of Intra- and Inter-languaage Calls

its default values, which incurs an overhead due to need to use `findall/3`, to return only the bindings of calls, as described in the previous section. Surprisingly, tests that set `truth_vals=None` did not run significantly faster than the default. Nonetheless, even the slowest timing in Table 4 is still fairly fast, supporting thousands of round trips calls per second.

The timings in Table 4 transferred small amounts of data between the languages. To illustrate the performance of bi-translation, Table 4 shows the transfer time per element of tuples, lists, sets and dictionaries of varying sizes, each of which consisted only of integers. For these tests, XSB called the Python function

```
def bitranslate(X):
   return(X)
```

so that Table 4 represents the average of times to translate a data structure from Prolog to Python and from Python to Prolog. Because XSB allows a maximum arity of 64K for predicates, the last two columns for tuples were not tested. While there was unavoidable variance in the timings, Translation times for sets and dictionaries, which in Python are based on hash tables, are sometimes higher than for lists and tuples. Nonetheless, the cost of per-element transfer is always small.

	10 elts	100 elts	1000 elts	10000 elts	100000 elts	1000000 elts
Tuple	30	28.9	36.1	48.7	*	*
List	53.8	54.3	73.4	58.4	77.8	73.5
Set	109	104	78.1	85.5	62.0	86.1
Dict.	113	40	221	58.5	90.5	72.5

Table 2: Per-element round-trip transfer cost for data structures of increasing size. All times in ns.

Python's garbage collection works by maintaining reference counts to each object: only when an object's reference count is 0 can the object be safely abolished and its space reclaimed. Because of *Janus*' use of bi-translation, it does not create persistent references to Python objects, except when a Python object reference is returned to Prolog, for use in e.g., `pydot/4`. Otherwise, the bi-translation code of *Janus* is able to execute without causing a memory leak due to stray references. This has been verified by tests of `jns_cmd()`, `jns_qdet()` and `jns_comp()` where the heap analysis toolset `guppy.heapy`[20] was used to check for uncollected Python objects.

5 Libraries

To support the use of *Janus* in applications, we have steadily built a library of *Janus* interfaces to Python packages. The XSB distribution includes interfaces to Elsasticsearch[21], Faiss (Facebook AI Similarity

---

[20] `https://pypi.org/project/guppy3`

[21] `https://www.elastic.co`

Search)[22] fastText word vectors ([5]), the GIS library Geopy[23], Google Maps[24], Google Translate[25], RDFlib (for `hdt`, `jsonld`, `ntriples`, `nquads`, and `turtle`)[26], shapely (an interface to GIS files)[27], SpaCy[28], Wikidata-hdt[29] , Wikidata Integrator[30]and others.

Because of the mapping of data structures between Python and Prolog, some of these interfaces need not be large. For instance, the `jns_elastic` library contains predicates to create and search Elasticsearch indices in various ways. Because the Python interface to Elasticsearch relies on sending and receiving JSON structures over a simple REST interface, much Elasticsearch functionality can be called directly using `pyfunc/[3,4]` or `pydot/4`. As an example, given a Python connection `conn`, searching an Elasticsearch index `ind` via a search condition `bdy` (represented as a Python dictionary) can be done via the Python statement

```
>>> result = conn.search(index=ind, body=bdy)
```

For a Prolog connection reference `Conn`, index atom `Ind` and Prolog dictionary term `Bdy`, the analogous statement is quite similar:

```
pydot(Conn.search(),[index=Ind, body=Bdy],Result)
```

Indeed, a few of the *Janus* libraries are little more than starting examples showing how common calls to a Python package are formed in *Janus*.

Case Study of a Library: jns_spacy. Some *Janus* libraries, such as `jns_spacy`, implement useful Prolog functionality on top of the basic Prolog-Python interface. SpaCy is a widely used Python tool that exploits neural language models to analyze text via tools such as dependency parses, named entity recognition, relation extraction, and sentence or span vectors based on Roberta ([9]) or other transformer-models.

To use `jns_spacy`, a SpaCy model is first loaded via the Prolog convenience predicate:

```
?- load_model(en_core_web_trf).
```

which loads SpaCy's English transformer model. Multiple models can be loaded when text from more than one language must be analyzed. Once a model is loaded, analytics commands can be called:

```
?- proc_string(en_core_web_trf,'She was the youngest of the two daughters
of a most affectionate, indulgent father; and had, in consequence of
her sister's marriage, been mistress of his house from a very early period.',Doc)
```

The above command uses `en_core_web_trf` to analyze the atom in the second argument, returning a Python object reference `Doc` to the SpaCy result. Text files can be analyzed in a similar manner. At this stage, either the `Doc` object can be queried directly, or the object's dependency graph can be asserted into Prolog via

```
?- token_assert(Doc).
```

The Prolog form of the dependency graph has binary relations connecting nodes of the form

```
token_info(Index,Text,Lemma,Pos,Tag,Dep,EntType)
```

---

[22] `https://engineering.fb.com/2017/03/29/data-infrastructure`

[23] `https://geopy.readthedocs.io/en/stable`

[24] `https://developers.google.com/maps`

[25] `https://cloud.google.com/translate`

[26] `https://rdflib.readthedocs.io/en/stable`

[27] `https://shapely.readthedocs.io/en/stable/manual.html`

[28] `https://spacy.io`

[29] `https://www.rdfhdt.org/datasets`

[30] `https://github.com/SuLab/WikidataIntegrator`

Such a node contains in order: the character offset of the token within the string, the token itself, the lemma of the token (the base form of a verb or singular form or a noun), the token's part of speech, the fine-grained part of speech, the tokens relation with it's parent in the dependency graph, and the entity type if any, for a span containing the token.

A highly simplified example gives a taste of how Prolog can exploit the above interface to better use the results of neural models. Here, a Prolog rule infers a spatial proximity relation from the span information in the dependency graph:

```
trigger_relation(class(proximity),Span,rel(SibSpan,ChildSpan),[spatial]):-
    dg_subord_conj(Span,SibSpan,ChildSpan),
    span_EREid(SibSpan,physical),
    span_EREid(ChildSpan,physical).
```

The rule infers spatial proximity between entities (here represented as entity textual spans) `Span` and `ChildSpan`. The rule infers proximity if (i) `Span` has a dependency graph sibling `SibSpan` that is a physical entity (e.g., person, place, object); and (ii) `SibSpan` is the head of a subordinating conjunction to another entity `ChildSpan`. Both entities are required to be of the class `physical` by `span_EREid/2` which checks the classes of relation and event ids as well as entity ids. Other clauses (not shown) represent the different forms of a subordinating conjunctions using the Prolog form of the SpaCy dependency graph.

The above rule is a simplified example of actual rules used in an textual analytic application assessing how changing information affected the behavior of human agents. In our work, we use rule-based reasoning as a complement to neural models for neural extraction and annotation (e.g. SpaCy). Neural models are the standard for entity and relation extraction, but their results are often noisy, especially if they were trained on data different than the run-time domain. Accordingly rules like the above are useful to refine the meaning of neural models, to map a model trained on one ontology to a different ontology, to disambiguate the results of relation extraction (perhaps using defeasibility), and to assign the extraction a useful likelihood. [31] While we have not done so, similar rules could also be used for weak supervision of neural models using data labeling, as in the Snorkel framework.[32]

6 Discussion

As described in a companion paper ([1]), while *Janus* is still new, it has formed an essential basis for several commercial and institutional applications using XSB Prolog; it has also formed the basis for several commercial applications using `LVM` Prolog. The largest of the XSB applications involved reasoning over a large scale knowledge graph ($\mathcal{O}(10^8)$ edges) of entities, relations and events created by analyzing thousands of documents, images and videos. In this applications, XSB both performed reasoning, and orchestrated the integration of various Python tools, such as SpaCY and FastText, along with external KBs (e.g. Wikidata) accessed via Python. Orchestration was also a feature of a commercial research project in which Ergo, ([6] a system based on XSB, orchestrated and reasoned about natural language analysis and geo-spatial data; as well as a feature of a separate commercial research project on visual query answering. While these applications used Prolog calling Python, a more recent project used Python bindings to the Robot Operating System (ROS)[33] to call XSB to perform situational assessment reasoning for an

---

[31] The direct use of certainty measures from many neural models is unreliable, since the softmax layer of neural models may artificially reduce the entropy of its output distribution.

[32] `https://snorkel.ai/`

[33] `https://www.ros.org`

autonomous agent.

This small flurry of applications is no coincidence. Prolog's reasoning power and scalability has advanced over the last two decades: Prolog systems now include high-quality finite-domain and numerical constraint systems, event-action rules, defeasible logics based on the well-founded semantics with priorities and integrity constraints, and probabilistic and T-norm based reasoning. Yet one may argue that the application of these reasoning methods has been throttled by a lack of data. Most Prolog systems provide interfaces to relational databases, but few interface to the current generation of no-SQL databases such as Elasticsearch or MongoDB, or to systems like Faiss or Annoy that store high-dimensional numerical knowledge vectors. Most Prolog systems can be called by a language like C or Java, but it is not always easy to leverage these interfaces to embed Prologs in frameworks like ROS. Indeed, the types of data that applications need and the types of frameworks that applications use is constantly changing. The ability to read any new form of data or fit into any new framework is only possible for the most heavily used languages like Java, C#, Javascript and Python. However, the use of the *Janus* framework has allowed two Prologs to view Python as a term-server for nearly any kind of data, and for Python to view Prolog as a reasoning mechanism working over its standard data types.

Acknowledgements The authors would like to thank Michael Kifer for huge help with *Janus* configuration, David Warren for porting the Prolog to Python bridge to Windows, Muthukumar Suresh who contributed code used in the Prolog to Python bridge, and Paulo Moura who answered numerous questions about different Prologs.

References

[1] Carl Andersen & Theresa Swift (2023): *The Janus System: Combining Prolog and Python in Applications*. In: *Prolog - The Next 50 Years*, Springer, doi:`10.1007/978-3-031-35254-6_8`. ISBN 978-3-031-35253-9.

[2] M. Calejo (2004): *Interprolog: Towards a Declrative Embedding of Logic Programming in Java*. In: *JELIA*, Springer-Verlag, pp. 714–717, doi:`10.1007/978-3-540-30227-8_64`.

[3] M. Carlsson & P. Mildner (2012): *SICStus Prolog – the first 25 years*. *Theory and Practice of Logic Programming* 12(1-2), pp. 35–66, doi:`10.48550/arXiv.1011.5640`.

[4] D. Diaz, S. Abreu & P. Codognet (2012): *On the implementation of GNU Prolog*. *Theory and Practice of Logic Programming* 12(1-2), pp. 253–282, doi:`10.48550/arXiv.1012.2496`.

[5] E. Grave, P. Bojanowski, P. Gupta, A. Joulin & T. Mikolov (2018): *Learning Word Vectors for 157 Languages*. In: *ICLRE*. Available at `https://aclanthology.org/L18-1550`.

[6] B. Grosof, M. Kifer, T. Swift, P. Fodor & J. Bloomfield (2023): *Ergo: A Quest for Declarativity in Logic Programming*. In: *Prolog - The Next 50 Years*, Springer, doi:`10.1007/978-3-031-35254-6_18`. ISBN 978-3-031-35253-9.

[7] M. V. Hermenegildo, F. Bueno, M. Carro, P. López-Garciá, E. Mera, F. Morales & G. Puebla (2012): *An Overview of Ciao and its Design Philosophy*. *Theory and Practice of Logic Programming* 12(1-2), pp. 219–252, doi:`10.48550/arXiv.1102.5497`.

[8] ISO working group JTC1/SC22 (1995): *Prolog International Standard ISO-IEC 13211-1*. Technical Report, International Standards Organization.

[9] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M.Lewis, L. Zettlemoyer & V. Stoyanov (2019): *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. *CoRR* abs/1907.11692, doi:`10.48550/arXiv.1907.11692`.

[10] Paulo Moura (2022): *The Logtalk Handbook*, Release 3.61.0 edition.

[11] V. Santos Costa, L. Damas & R. Rocha (2012): *The YAP Prolog System*. *Theory and Practice of Logic Programming* 12(1-2), pp. 5–34, doi:`10.48550/arXiv.1102.3896`.

[12] J. Schimpf & K. Shen (2012): *ECLiPSe – From LP to CLP*. Theory and Practice of Logic Programming 12(1-2), pp. 127–156, doi:`10.48550/arXiv.1012.4240`.

[13] T. Swift & D.S. Warren (2012): *XSB: Extending the Power of Prolog using Tabling*. Theory and Practice of Logic Programming 12(1-2), pp. 157–187, doi:`10.48550/arXiv.1012.5123`.

[14] P. Tarau (2021): *Natlog: a Lightweight Logic Programming Language with a Neuro-symbolic Touch*. In: ICLP, doi:`10.48550/arXiv.2109.08291`.

[15] Jan Wielemaker, T. Schrijvers, M. Triska & T. Lager (2012): *SWI-Prolog*. Theory and Practice of Logic Programming 12(1-2), pp. 67–96, doi:`10.48550/arXiv.0711.0917`.

[16] N. Zhou, H. Kjellerstrand & J. Fruhman (2015): *Constraint Solving and Planning with Picat*. Springer Briefs in Intelligent Systens, doi:`10.1007/978-3-319-25883-6`.

[17] N.F. Zhou (2012): *The Language Features and Architecture of B-Prolog*. Theory and Practice of Logic Programming 12(1-2), pp. 189–218, doi:`10.48550/arXiv.1103.0812`.

The ICARUS-System for Interactive Coherence Establishment in Logic Programs

Andre Thevapalan
Technische Universität Dortmund, Dortmund, Germany
andre.thevapalan@tu-dortmund.de

Jesse Heyninck
Open Universiteit, the Netherlands
jesse.heyninck@ou.nl

When modelling expert knowledge as logic programs, default negation is very useful, but might lead to there being no stable models. Detecting the exact causes of the incoherence in the logic program manually can become quite cumbersome, especially in larger programs. Moreover, establishing coherence requires expertise regarding the modelled knowledge as well as technical knowledge about the program and its rules. In this demo, we present the implementation of a workflow that enables knowledge experts to obtain a coherent logic program by modifying it in interaction with a system that explains the causes of the incoherence and provides possible solutions to choose from.

1 Introduction

Due to its declarative nature, and with the availability of both strong and default negation, answer set programming offers valuable features for the usage in real-world applications. But often knowledge bases are subject to change. Adding rules to a logic program is not as straightforward as it might seem. Due to its declarative nature, adding new information can easily lead to contradictory knowledge. Such contradictions can result from rules whose conclusions are contradictory while both their premises can potentially hold simultaneously. However, by using default negation, another form of inconsistency can appear which is called *incoherence*. Incoherent answer set programs do not possess answer sets. In many cases, though, the cause of the incoherence generally comprises only one or a couple of rules. Several approaches show how these causes can be found and which part of the programs are affected and thus have to be revised in order to restore the coherence of a program [5, 4, 10]. These approaches do not describe, however, how to establish coherence. In recent work [11], we have defined a method that offers strategies for computing suitable solutions without the need for additional information from the user and which in turn can be used to enable implementations where coherence can be interactively established in collaboration with an expert. This method is based on results on coherence in argumentation graphs [10] for normal logic programs which in turn enables us to locate those parts of the program that are responsible for the incoherence.

2 The ICARUS-system

We implemented the framework from [11] in a prototype system[1]. In this section, we describe and illustrate this implementation.

Figure 1 shows the general workflow that is implemented in this prototype for establishing the coherence in a given incoherent program $\mathscr{P}$.

---

[1] The implementation is available on icarus.cs.tu-dortmund.de.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D’Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 256–259, doi:10.4204/EPTCS.385.25

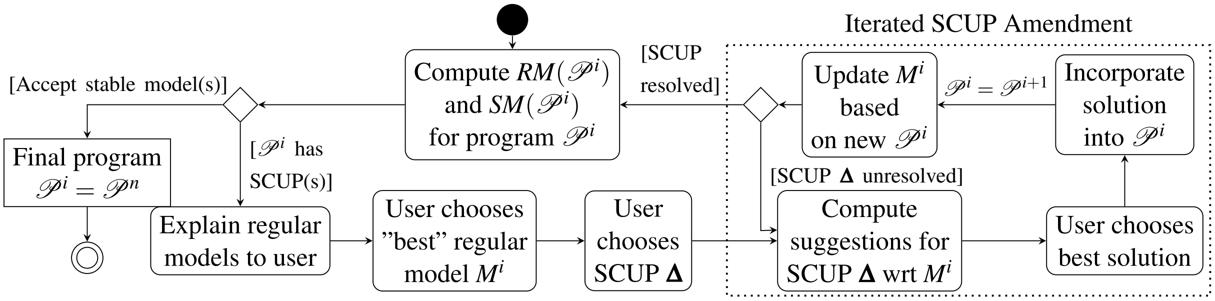


Figure 1: Amendment Workflow

In the following, we will describe the general workflow for this application, and how it helps the knowledge expert to gradually restore coherence. For this consider the following program $\mathscr{P}^1$ inspired by the medical domain will be used as a running example[2]:

r_1: $\mathit{fever} \leftarrow \mathit{highTemp}, {\sim}\mathit{sunstrk}.$
r_2: $\mathit{fatigue} \leftarrow \mathit{lowEnergy}, {\sim}\mathit{stress}, {\sim}\mathit{sports}.$
r_3: $\mathit{highTemp} \leftarrow \mathit{temp} > 37.$
r_4: $\mathit{sunstrk} \leftarrow \mathit{highTemp}, {\sim}\mathit{fever}.$
r_5: $\mathit{allrg} \leftarrow \mathit{pollenSeason}, \mathit{fatigue}, {\sim}\mathit{flu}, {\sim}\mathit{cold}.$
r_6: $\mathit{cold} \leftarrow \mathit{fatigue}, \mathit{soreThroat}, {\sim}\mathit{migrn}.$
r_7: $\mathit{flu} \leftarrow \mathit{fatigue}, {\sim}\mathit{migrn}.$
r_8: $\mathit{migrn} \leftarrow \mathit{headache}, {\sim}\mathit{allrg}.$
r_9: $\mathit{soreThroat}.$ r_{10}: $\mathit{headache}.$ r_{11}: $\mathit{pollenSeason}.$ r_{12}: $\mathit{lowEnergy}.$ r_{13}: $\mathit{temp} > 37.$

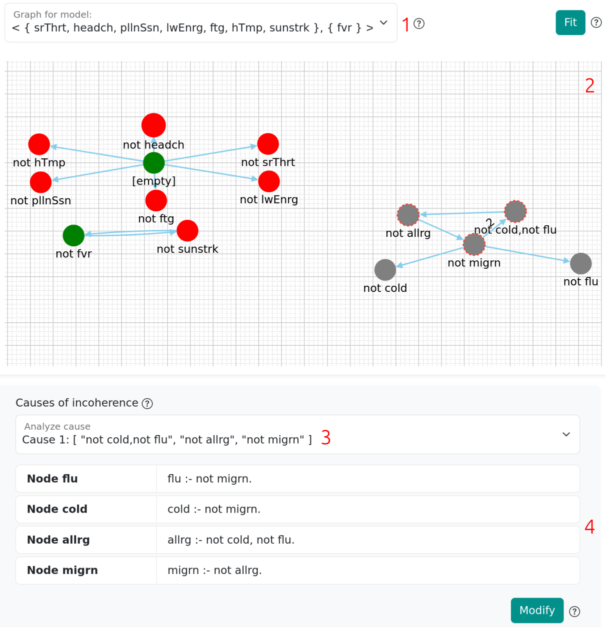
The process starts with the knowledge expert providing an extended logic program [7], e.g. $\mathscr{P}^1$, via the interface[3]. For each $\mathscr{P}^i$, the regular models are computed using an ASP encoding for abstract argumentation frameworks presented in [8]. If the program is already coherent, the total stable interpretations are shown accordingly. If not, the user can choose the regular model M^i that is closest to an intended total stable interpretation of $\mathscr{P}$ (see Figure 2 (1)). On the basis of the labelling that corresponds to M^i, the argumentation graph for $\mathscr{P}^i$ is computed and shown to the user alongside the subgraphs responsible for incoherence (called *SCUPS*) (see Figure 2 (2)). The system then displays the argumentation graph, where nodes are coloured according to their label where gray nodes stand for undefined atoms, green ones for true atoms, and red nodes for false atoms. Likewise, nodes that are part of a SCUP have red dashed border. Figure 2 shows that $\mathscr{P}^1$ contains a SCUP involving the atoms *allrg*, *cold*, *flu*, and *migrn*. The user is then shown the list of SCUPs and can choose the SCUP they want to resolve (first) (see Figure 2 (3)). For the chosen SCUP, the interface shows all involved atoms and rules (see Figure 2 (4)).

Advantage is taken of the high explanatory potential of computational argumentation by implementing explanation methods inspired by the dialogue games for argumentation [2, 3] to explain (non-)acceptability of atoms in a regular model, via the explanation of the (non-)acceptability of an argument for that atom w.r.t. the corresponding preferred labelling of the argumentation framework. This component contains explanations for the label of each atom (set) in $\mathsf{AF}(\mathscr{P}^i)$. It uses an approach derived from argumentative dialogue games to tackle the following explanation task: Given the regular interpretation $\langle I, I' \rangle$ of $\mathscr{P}$ and $A \in \mathscr{A}$, explain $\texttt{Int2Lab}(\langle I, I' \rangle)({\sim} A)$. If the user wishes to see explanations for the labels of certain atom sets, they can choose the respective set in this component (see Figure 4 (1)). For the chosen atom set, the application will present the list of their attackers and explain each of their labels. It furthermore shows the rules that are associated with the atom set and their labelling. Figure 4 (2) shows

---

[2] Here, the atoms *sunstrk*, *migrn* and *allrg* denote that a person suffers from a sunstroke, migraine, or allergies, respectively.

[3] Future work will include the support of ASP extensions such as aggregates and cardinality atoms [1].



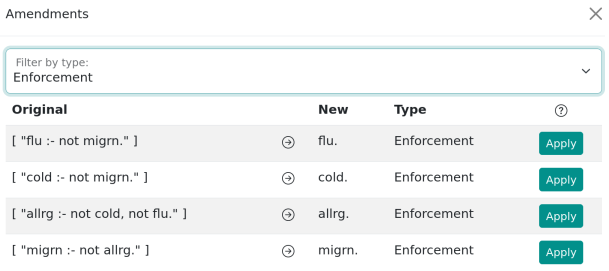
Graph for model:
< { srThrt, headch, plInSsn, lwEnrg, ftg, hTmp, sunstrk }, { fvr } >

Causes of incoherence

Analyze cause
Cause 1: ["not cold,not flu", "not allrg", "not migrn"]

Node flu	flu :- not migrn.
Node cold	cold :- not migrn.
Node allrg	allrg :- not cold, not flu.
Node migrn	migrn :- not allrg.

Figure 2: SCUP visualisation

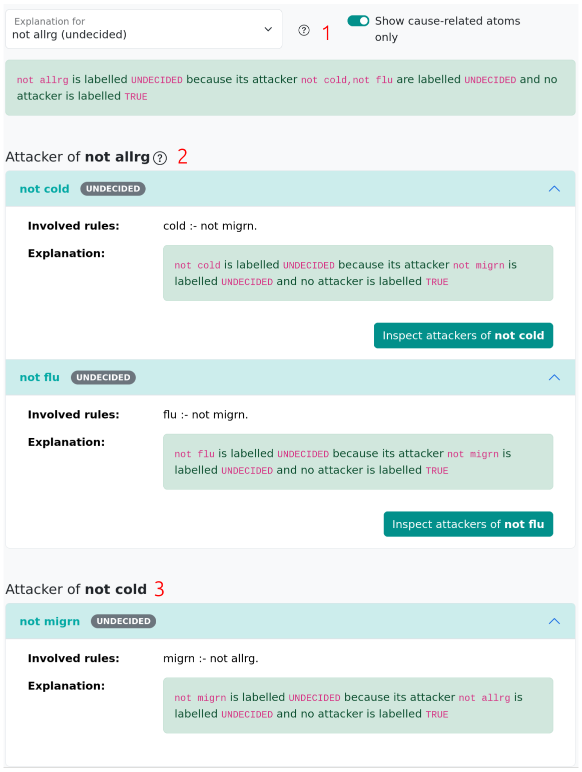


Amendments

Filter by type:
Enforcement

Original	New	Type
["flu :- not migrn."]	flu.	Enforcement
["cold :- not migrn."]	cold.	Enforcement
["allrg :- not cold, not flu."]	allrg.	Enforcement
["migrn :- not allrg."]	migrn.	Enforcement

Figure 3: Amendment selection



Explanation for
not allrg (undecided)

Show cause-related atoms only

`not allrg` is labelled `UNDECIDED` because its attacker `not cold,not flu` are labelled `UNDECIDED` and no attacker is labelled `TRUE`

Attacker of **not allrg**

not cold UNDECIDED

Involved rules: cold :- not migrn.

Explanation: `not cold` is labelled `UNDECIDED` because its attacker `not migrn` is labelled `UNDECIDED` and no attacker is labelled `TRUE`

not flu UNDECIDED

Involved rules: flu :- not migrn.

Explanation: `not flu` is labelled `UNDECIDED` because its attacker `not migrn` is labelled `UNDECIDED` and no attacker is labelled `TRUE`

Attacker of **not cold**

not migrn UNDECIDED

Involved rules: migrn :- not allrg.

Explanation: `not migrn` is labelled `UNDECIDED` because its attacker `not allrg` is labelled `UNDECIDED` and no attacker is labelled `TRUE`

Figure 4: Explanation component

the explanation why the assumption $\sim allrg$ is labelled *undecided* in the chosen model. If an explanation for an attacker is additionally available, the user can analyze these as well. In Figure 4 (3) the presented explanation is illustrated if the user decides to inspect the attackers of $\sim cold$.

In a next step, on the basis of the explanations for the labelling of $\mathrm{AF}(\mathscr{P}^i)$, the user can explore the amendments that are suitable for resolving for resolution of the chosen SCUP. The amendments are computed using the approach defined in [11]. The list of amendments can additionally be filtered by a strategy as even relatively small programs can have a vast amount of possible amendments (see Figure 3). For each amendment, its used strategy, the amended rule and amendment are shown. The chosen amendment is directly applied on the given input program $\mathscr{P}^i$ and the modified program $\mathscr{P}^{i+1}$ is solved. In our running example, applying the first suggested amendment that is shown in Figure 3 would replace r_7 in $\mathscr{P}^1$ by a fact *migrn.*, which results in a coherent program as this modification dissolves the unique SCUP in $\mathscr{P}^1$. If otherwise $\mathscr{P}^{i+1}$ is still incoherent, the remaining SCUPs can be resolved by starting a new iteration of this process again with the modified program $\mathscr{P}^{i+1}$. The result of this workflow is a coherent program $\mathscr{P}^n$ that represents professionally adequate knowledge.

The presented system was implemented using *Python 3.10* and *clingo 5.6* [6] with *asprin 3.1.2beta* [9].

References

[1] Gerhard Brewka & Thomas Eiter (1999): *Preferred Answer Sets for Extended Logic Programs*. Artif. Intell. 109(1-2), pp. 297–356, doi:10.1016/S0004-3702(99)00015-6.

[2] Martin Caminada (2017): *Argumentation semantics as formal discussion*. Journal of Applied Logics 4(8), pp. 2457–2492.

[3] Martin WA Caminada, Wolfgang Dvořák & Srdjan Vesic (2014): *Preferred semantics as socratic discussion*. Journal of Logic and Computation 26(4), pp. 1257–1292, doi:10.1093/logcom/exu005.

[4] Stefania Costantini (2006): *On the existence of stable models of non-stratified logic programs*. Theory Pract. Log. Program. 6(1-2), pp. 169–212, doi:10.1017/S1471068405002589.

[5] Stefania Costantini, Benedetto Intrigila & Alessandro Provetti (2003): *Coherence of updates in answer set programming*. In: In Proc. of the IJCAI-2003 Workshop on Nonmonotonic Reasoning, Action and Change, pp. 66–72.

[6] Martin Gebser, Roland Kaminski, Benjamin Kaufmann & Torsten Schaub (2019): *Multi-shot ASP solving with clingo*. Theory Pract. Log. Program. 19(1), pp. 27–82, doi:10.1017/S1471068418000054.

[7] Michael Gelfond & Vladimir Lifschitz (1991): *Classical Negation in Logic Programs and Disjunctive Databases*. New Gener. Comput. 9(3/4), pp. 365–386, doi:10.1007/BF03037169.

[8] Tuomo Lehtonen, Johannes Peter Wallner & Matti Järvisalo (2021): *Declarative Algorithms and Complexity Results for Assumption-Based Argumentation*. J. Artif. Intell. Res. 71, pp. 265–318, doi:10.1613/jair.1.12479.

[9] Javier Romero (2017): *asprin: Answer Set Programming with Preferences*. In Bernhard Mitschang, Norbert Ritter, Holger Schwarz, Meike Klettke, Andreas Thor, Oliver Kopp & Matthias Wieland, editors: Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Workshopband, LNI P-266, GI, pp. 159–162. Available at https://dl.gi.de/20.500.12116/910.

[10] Claudia Schulz & Francesca Toni (2018): *On the responsibility for undecisiveness in preferred and stable labellings in abstract argumentation*. Artif. Intell. 262, pp. 301–335, doi:10.1016/j.artint.2018.07.001.

[11] Andre Thevapalan, Jesse Heyninck & Gabriele Kern-Isberner (2021): *Establish Coherence in Logic Programs Modelling Expert Knowledge via Argumentation*. In Joaquín Arias, Fabio Aurelio D'Asaro, Abeer Dyoub, Gopal Gupta, Markus Hecher, Emily LeBlanc, Rafael Peñaloza, Elmer Salazar, Ari Saptawijaya, Felix Weitkämper & Jessica Zangari, editors: Proceedings of the International Conference on Logic Programming 2021 Workshops co-located with the 37th International Conference on Logic Programming (ICLP 2021), Porto, Portugal (virtual), September 20th-21st, 2021, CEUR Workshop Proceedings 2970, CEUR-WS.org. Available at https://ceur-ws.org/Vol-2970/causalpaper4.pdf.

Inferring Compensatory Kinase Networks in Yeast using Prolog

George A. Elder
Queen Mary University of London
School of Biological and Behavioural Sciences
QMUL [*]
London, United Kingdom
g.a.elder@qmul.ac.uk

Conrad Bessant
Queen Mary University of London
School of Biological and Behavioural Sciences
QMUL
London, United Kingdom
c.bessant@qmul.ac.uk

Signalling pathways are conserved across different species, therefore making yeast a model organism to study these via disruption of kinase activity. Yeast has 159 genes that encode protein kinases and phosphatases, and 136 of these have counterparts in humans. Therefore any insight in this model organism could potentially offer indications of mechanisms of action in the human kinome. This study utilises a Prolog-based approach, data from a yeast kinase deletions strains study and publicly available kinase-protein associations. Prolog, a programming language that is well-suited for symbolic reasoning is used to reason over the data and infer compensatory kinase networks. This approach is based on the idea that when a kinase is knocked out, other kinases may compensate for this loss of activity. Background knowledge on kinases targeting proteins is used to guide the analysis. This knowledge is used to infer the potential compensatory interactions between kinases based on the changes in phosphorylation observed in the phosphoproteomics data from the yeast study. The results demonstrate the effectiveness of the Prolog-based approach in analysing complex cell signalling mechanisms in yeast. The inferred compensatory kinase networks provide new insights into the regulation of cell signalling in yeast and may aid in the identification of potential therapeutic targets for modulating signalling pathways in yeast and other organisms.

1 Introduction

As with any organism, so with *Saccharomyces cerevisiae*, protein phosphorylation plays a crucial role in various signalling pathways that control cellular activities such as gene expression, protein synthesis, DNA replication, RNA processing, cell cycle regulation, metabolism, transport of vesicles and other organelles, response to environmental stress and nutrients, and cell differentiation. The specific actions of kinases and substrates are largely determined by protein interactions, which are effected via phosphorylation [10].

It is also known that many of these signalling pathways are conserved across different species, so yeast can be used as a model organism to study phosphorylation networks. Yeast has 159 genes that encode protein kinases and phosphatases, and 136 of these have counterparts in humans [8]. Therefore any insight in this model organism could potentially offer indications of mechanisms of action in the human kinome.

Ours is not the first application of symbolic AI in the study of yeast and its pathways more specifically. A major inspiration for applying techniques that fall under the symbolic framework came from the extensive work done by Prof R.King. Particularly, work of the type found in this study by A.Clare and R.King [2] showed that it is possible to build a knowledge base via the use of a declarative logic

---

[*] Research partially funded by Medical Research Council MRC

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 260–273, doi:10.4204/EPTCS.385.26

programming language. The seminal work on Robot Scientist Project also offered insight in how logic programming (in this case inductive) can be used to model the metabolism of yeast and how deletion in associated genes can affect their respective processes [6]. Further use of a Prolog based approach can be found in [12] where a Flux Balance Analysis model was combined with genome related information from the KEGG database to predict the growth patterns of yeast. More recently, network reconstruction as well as new structure hypotheses were made using ProbLog, with the aim of providing new insights to experimental biologists [4].

However, all of the above approaches have focused on the use of gene-based background knowledge and experimentally derived facts to reach their conclusions. These lack the granularity of a proteomics and even more so a phosphoproteomics based approach.

2 Methods

2.1 Toy Model development

Initially a Toy Model version of our logic program was developed, in order to test the feasibility and applicability of our proposed methodology. The perturbations modelled were gene deletions and the question thus became whether silencing (or deletion) of a given kinase has actually worked. Within the context of the study chosen (to be described in detail in subsequent section), specific gene deletions were confirmed by either proteomics or PCR assays. Therefore and due to the nature of gene deletions in yeast it was unnecessary to identify off targets or known targets of a given perturbation.

In Figure 1 the overall structure of our yeast Toy Model can be seen. Edges represent either perturbations (deletions in this case), observational *facts*, presence of a phosphosite on a kinase and kinase phosphosite associations. Nodes represent either phosphosites, perturbations, kinases or other proteins.

To determine whether a kinase has been successfully silenced, development of the logic program began with the first iteration of *rules*. When queried, it is satisfied in the case where a kinase **K** (1), for which a known phosphosite **S** target on a protein **P** is found (2). Lastly, (3) checks whether a corresponding mass spectrometry *fact* following deletion of **K** can be matched. If all these conditions are met then the interpreter responds with `true` thus confirming that the δgene perturbation encoding kinase **K** has been successful.

```
doesSilofKwork(deletionOF((kinase(K))) :-
    phosphorylates(kinase(K), phosphosite(S), protein(P)),
    perturbs(deletionOF((kinase(K))),phosphosite(S), occupancy(down),_).
```

In the next *rule*, developed in an iterative manner, a “uniqueness check” was added. This establishes whether a phosphosite **S** is uniquely targeted by a kinase **K**. This is achieved by establishing a *rule* and checking for its negation as part of this second iteration. In Prolog notation:

```
doesSilofKwork2(deletionOF((kinase(K))) :-
    phosphorylates(kinase(K), phosphosite(S), protein(P)),
    \+ sharedtarget(kinase(K), kinase(_K2), phosphosite(S)),
    perturbs(deletionOF((kinase(K))),phosphosite(S), occupancy(down),_).
```

The *rule* `sharedtarget(kinase(K), kinase(_K2), phosphosite(S))` includes two goals matching phosphorylation *facts* on shared phosphosites then checking that the two phosphorylating are not the same.

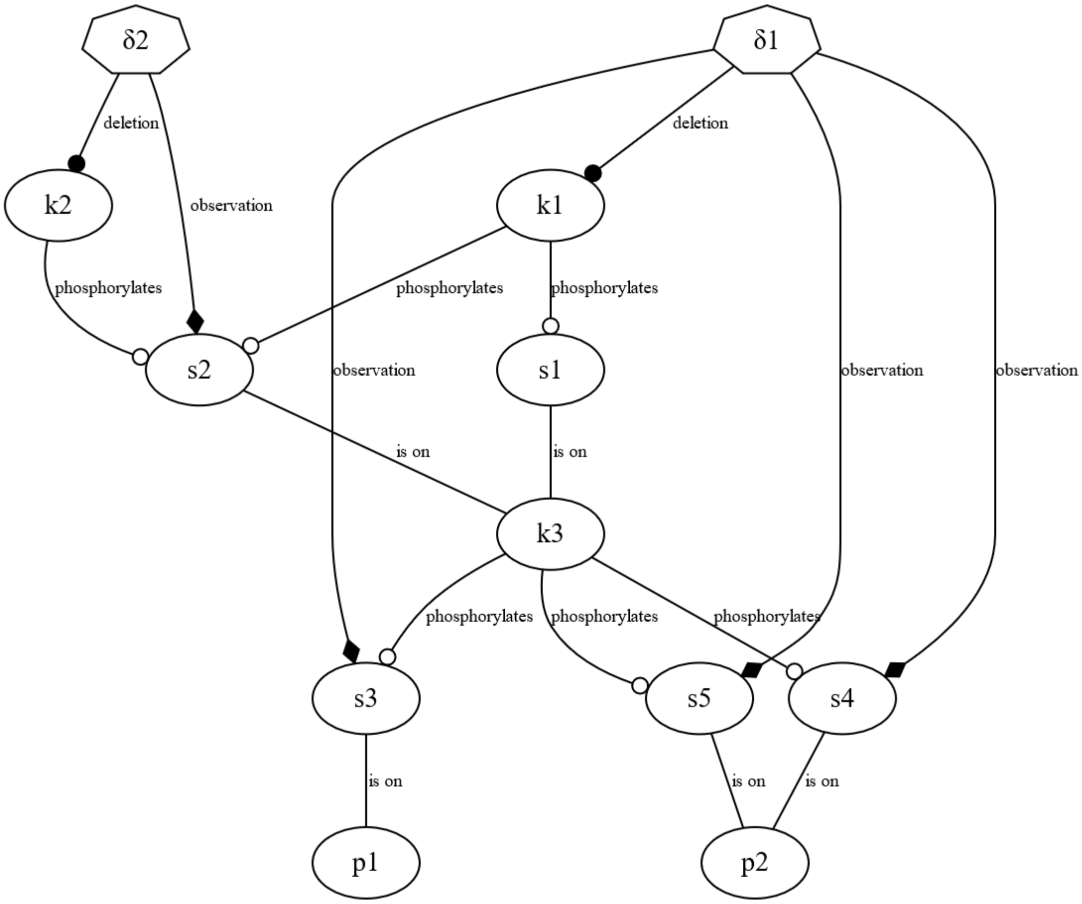


Figure 1: A Toy Model illustrates the minimal number of interactions needed to understand a biological scenario involving kinases (k), proteins (p), and their phosphosites (s) in response to an initial perturbation (δ). In our model, the perturbation refers to a gene encoding a specific kinase being deleted. Our background information in this implementation has levels of granularity, in this case the kinase targeting a phosphosite on a protein version is depicted.

```
sharedtarget(kinase(K), kinase(K2), phosphosite(S)) :-
    (phosphorylates(kinase(K), phosphosite(S), kinase(P)),
    phosphorylates(kinase(K2), phosphosite(S), kinase(P)),
    K1 \= K2).
```

In the toy model, the responses to these two iterations were as expected. However, there is still a concern that when applied to the larger dataset with more than five perturbation-affected phosphosites, the true responses will be based on only one phosphosite instance. Therefore the addition of a majority check was decided. The final version of the *rule* took the following form:

```
doesSilofKwork3(deletionOF((kinase(K))) :-
    phosphorylates(kinase(K), phosphosite(S), kinase(P),
    \+ sharedtarget(kinase(K), kinase(_K2), phosphosite(S)),
    findall((phosphosite(S),
    perturbs(deletionOF((kinase(K))),phosphosite(S), occupancy(up),_),
    perturbs(deletionOF((kinase(K))),phosphosite(S), occupancy(
       unaffected),_),
    List1)),
    findall((phosphosite(S),
    perturbs(deletionOF((kinase(K))),phosphosite(S), occupancy(down),_),
    List2)))),
    length(List1, Length1), length(List2, Length2),
    compare( '>', Length2, Length1).
```

This *rule* adds the majority check to the previous *rule* version. This creates two lists, one containing all known targets of the kinase that are either “up” or “unaffected” and another that contains all the ones described as “down”, then compares the length of the two. The aim of this is to establish whether the majority of the known targets were “down” rather than not, therefore confirming the silencing of the kinase.

2.2 Experimental *facts* and background knowledge sources

A recently published study that looked at the effect of kinase perturbation at the phosphoproteomic level was chosen as the source of our experimental data. In [7], a systems-level proteomic and phosphoproteomic analysis was carried out on 110 yeast single-kinase or phosphatase deletion strains under standard growth conditions. The 110 were split in 84 kinases and 26 phosphatases. They employed various methods, including traditional enrichment analysis, δgene-δgene correlation networks, and molecular covariance networks in order to analyse the functional relationships between these active proteins.

In total their chosen deletions, which will be referred to as perturbations in this section, cover 82% of all possible yeast and phosphatase deletion strains. Through their experimental and mass spectrometry workflow they were able to identify more than 4,600 and 13,000 proteins and phosphosites, respectively. Of interest is the finding that they were able to capture a large part of regulated phosphorylation events as well as 30% of those being newly captured. This was entirely attributed to their ability to normalise their data with protein abundance. This is also the data that we elected to include in our ’perturbs/n’ predicate format.

Key constituents of our background knowledge base are the associations between kinases and their targets both at the protein and phosphosite levels. For this, two main sources were chosen, namely the Yeast Kinase Interaction Database [11] (accessible at `http://www.moseslab.csb.utoronto.ca/KID/`) and the Yeast Kinome database [1] (accessible at `https://thebiogrid.org/project/2`).

The Yeast Kinase Interaction Database (KID) is a resource that contains data on phosphorylation events from various high- and low-throughput experiments. It includes a total of literature-curated low-throughput (6,225) and high-throughput (21,990) interactions, stemming from over 35,000 experiments. It includes 517 high-confidence (or 853 low-confidence) kinase-substrate pairs depending on the cutoffs used for the metric provided in the study. On the other hand the Yeast Kinome database contains 1,844 interactions observed by mass-spectrometry based analysis of protein complexes. It is updated monthly and has been subsumed as part of the BioGRID project which, in turn is a broad interaction database including more than 1.7m individual interactions [9].

Both these contain kinase (and phosphatase) target associations with the latter (Yeast Kinome db) also containing these at the phosphosite level. As will be shown below however, there was an issue of overlap between our experimental *facts* and background knowledge which limited the predictions that could be made.

2.2.1 Yeast perturbation observation *facts* and background knowledge sources

From the perturbation study chosen, a total of 685,358 *facts* were extracted containing the effect of the aforementioned 110 kinase and phosphatase deletions over 13,258 individual phosphosites. They were extracted from a .csv file containing the log2fold change recorded for each phosphosite between wild type and δgene strains of yeast (i.e. strains where selected genes have been deleted). For most phosphosites there were two replicates reported. Not all phosphosites had both replicates for all δgene combinations. Only phosphosites containing two replicates were considered and the mean value between the two was taken for those that did. The code determined whether the average of the log2 fold change values is less than or equal to -0.5, greater than or equal to 0.5, or between those two values. Depending on the value of the average, the code sets the value of a variable called “ change” to “down”, “up”, or “unaffected”, respectively. *Facts* took the following form:

```
perturbs('YCK2', 'ENO2_pT324', 'ENO2', unaffected, 0.0475).
```

The background knowledge base includes *facts* that describe the relationships between kinases, proteins and phosphosites at both the kinase-protein and kinase-phosphosite level. These were taken from the two databases described above and were transcribed into *facts* in the following format:

Protein level information: `knownKtarget('ARK1', 'PAN1').`
Phosphosite level information: `knowntarget('ATG1','ALY1_pS813').`

The first set of *facts* describing kinase - protein level information came from the Kinome Interaction Database and the Yeast KINOME database, while the kinase - phosphosite level information came from the KINOME/bioGRID database alone.

All scripts used during data and background knowledge curation and transcription as well as .pl Prolog files can be found at `https://github.com/Dudelder/Yeast_Prolog_Compensatory`). Total number of *facts* that make up knowledge base can be seen in Table 1.

2.2.2 Overlap between experimental observations and background knowledge

There is an overlap between the observational *facts* at the kinase-phosphosite level and the phosphosites in the background knowledge base. Specifically, out of the total number of individual phosphosites, only 667 are found in the kinase-phosphosite associations knowledge base (Figure 2 A), accounting for approximately 5% of the total.. At the protein level things improve slightly depending on the data source and cutoff used (for Kinase Interaction Database). In the Kinome database (see Figure 2), there is an overlap between the information obtained from Kinase Protein pairs and Kinase Protein Phosphosite level triples. However, there is a discrepancy in the numbers. The overlap consists of 72 deleted kinases when considering the Kinase Protein pairs, but only 58 deleted kinases when extracting the information from the Kinase Protein Phosphosite level triples. This difference may be due to errors in formatting or

Facts List	Source	Size (Number of facts)
Dgene-Dgene phosphosite phosphosite observations	Perturbation Study	685,358
Kinase Substrate Associations	Yeast Kinome Database / bioGRID	7589
Kinase Protein Associations	Yeast Kinome Database / bioGRID	1558
Kinase Protein Associations	Kinase Interaction Database (lenient)	853
Kinase Protein Associations	Kinase Interaction Database (Strict)	517
	Total =	695,875

Table 1: Table listing the title, sources and associated number of *facts* that make up the Background Knowledge section of the yeast logic model. The total reported is for the strict cut-off of the Kinome Interaction Database. For the lenient cut-off, the total is 696,211. Specific sources referenced [11], [7], [1].

the omission of a subset of data in one of the database files. Looking at the Kinase Interaction database as a source, the overlap depends on the cutoff used. Cutoffs of 6.73 and 4.72 correspond to a P value of less than 0.01 and 0.05, respectively, for strict and lenient lists of kinase-protein pairs as were suggested in the article. Accordingly the overlap between the deleted kinases and the two lists of Kinase and their target Protein pairs was 52 and 58 for the strict and lenient cutoffs, respectively. As will be demonstrated in the results section of this paper, the poor overlap was addressed by considering combinations of background knowledge sources.

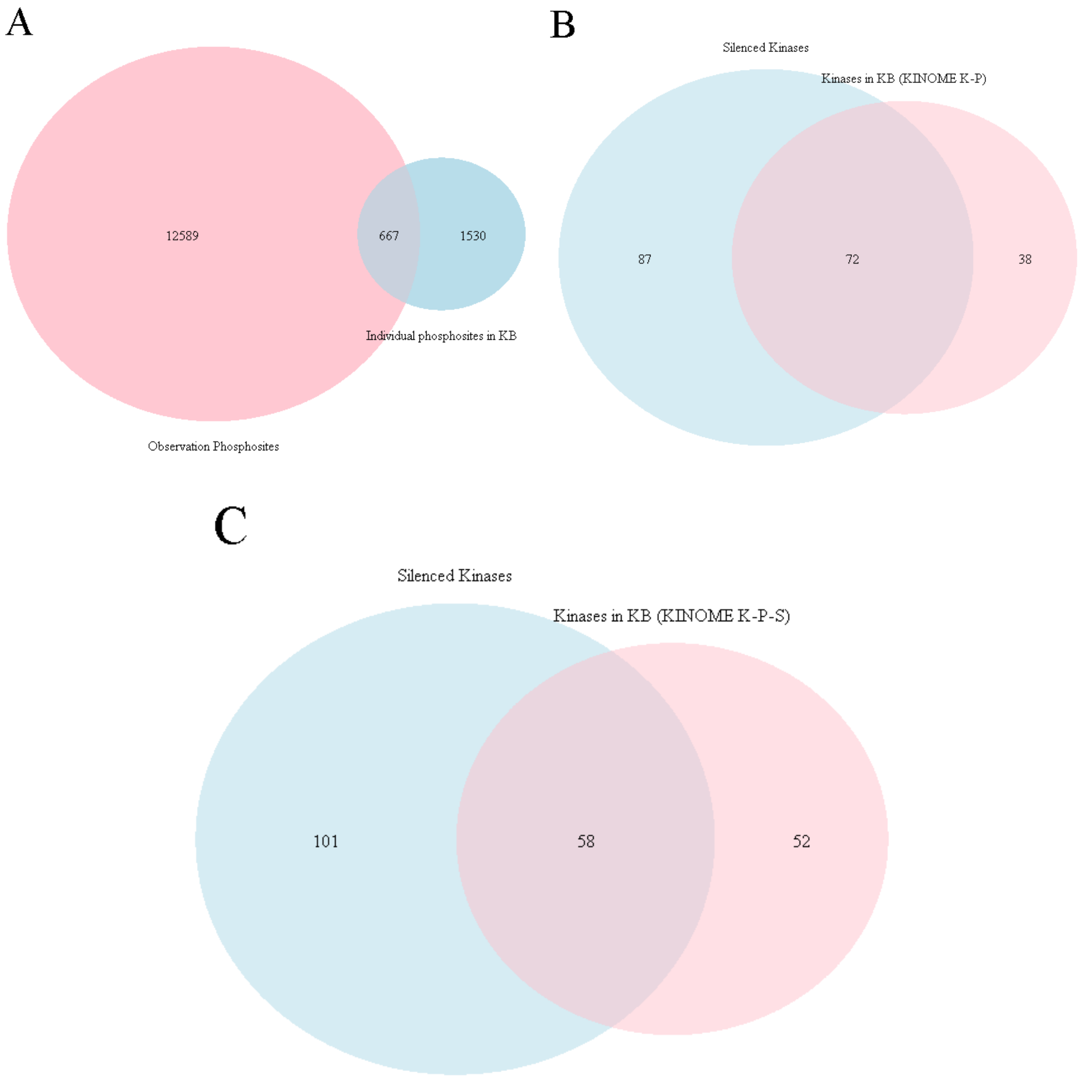


Figure 2: Venn Diagrams depicting the overlap between our experimental data and background knowledge sourced from Kinome Database / bioGRID [1]. In A) total number of phosphosites is 13,256. B: Overlap between kinases in KinomeDB/bioGRID and deleted kinases at the kinase-protein level of association. C: Overlap between kinases in KinomeDB/bioGRID and deleted kinases at the kinase-phosphosite level of association.

3 Results

3.1 Silence-of-K Rule iterations and Kinase Compensation Rule

Collecting the results was accomplished following a pipeline briefly described here. The Prolog interpreter was used to perform queries across all kinase deletions, and the resulting output was stored in lists. These lists were then parsed by Python scripts to perform additional visualisation and analysis in RStudio. This centered around the number of deletions that were predicted to have worked. Additionally, RStudio was used to create the kinase compensation networks, as described below in this section.

Using kinase-phosphosite level associations, the first iteration of the *rule* proved to exhibit the best recall, confirming 21 kinase gene deletions having taken place, thus confirming the ground truth established by PCR, as reported in the study. Successive iterations of the *rule*, 2nd (checking uniqueness) and 3rd (checking majority of known phosphosites) were less accurate, both confirming only 5 kinases as being silenced. Using kinase-protein level associations the results improved overall depending on the source of kinase protein pairs used. For the Kinome Database from bioGrid, 33, 7 and 3 kinase deletions were confirmed for *rule* iterations 1, 2 and 3, respectively. Kinase Interaction Database (KID) with a lenient cut-off yielded 26, 9 and 5, whereas with a strict cut-off the results were 21, 8 and 4. Results summarised in Table 2.

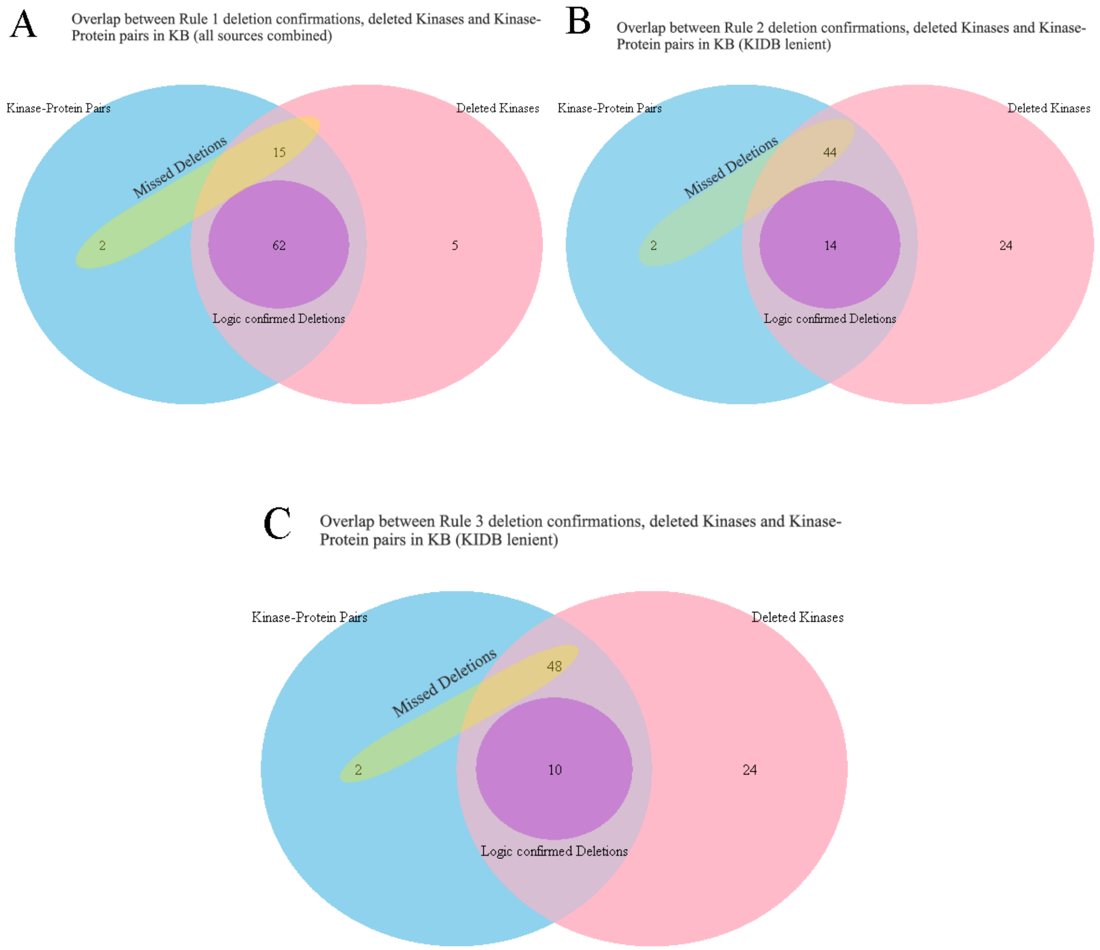
As the overlap with a ground truth (the deletion of a kinase encoding gene) was poor, it was imperative to look at ways of improving this. Firstly, combining the outputs of the two levels of background knowledge considered was the most straightforward approach. Starting with the Kinome Database as a Kinase-Protein association source, 54, 12 and 8 ground truths were captured with each successive iteration of the *rule*. From the Kinase Interaction Database, the lenient cut off gave 47, 14 and 10, and with the strict cut off 42, 13 and 9. Overall we noted a significant improvement which was mainly evident in the first iteration of the rule, picking up increasingly more of the ground truth. These can be considered True Positive results.

Rule Iteration & Background Knowledge Level (Source where appropriate)	True Positive	False Negative
First Iteration & Phosphosite level	21	63
Second Iteration & Phosphosite level	5	79
Third Iteration & Phosphosite level	5	79
First Iteration & Protein level (bioGrid)	33	51
Second Iteration & Protein level (bioGrid)	7	77
Third Iteration & Protein level (bioGrid)	3	81
First Iteration & Protein level (KID Strict)	21	63
Second Iteration & Protein level (KID Strict)	8	76
Third Iteration & Protein level (KID Stict)	4	80
First Iteration & Protein level (KID Lenient)	26	68
Second Iteration & Protein level (KID Lenient)	9	75
Third Iteration & Protein level (KID Lenient)	5	89

Table 2: Table listing the True Positive and False Negative counts for each of the rule iterations. The False Negatives refer to kinase gene deletions we were not able to pick up due to either poor background information overlap or *rule* iterations not performing as intended.

The next step was to combine all of the known background sources whilst removing duplicates to avoid clashes. This yielded a kinase-protein pair background knowledge base which covered (with at least one association) 79/82 of the deleted kinases in our dataset. This was the largest overall coverage and yielded the best result, confirming 62 individual deletions as seen in Figure 3. In contrast, iterations two and three yielded worse results, 13 and 8, respectively. From the above the best combinations between knowledge base sources (KID: strict/lenient and KINOME:bioGRID), level of background knowledge considered (kinase-phosphosite, kinase-protein or both) for each rule iteration were:

- Rule 1: All BK, both levels: 62 TP / 17 FN
- Rule 2: KID Lenient cut-off, both levels: 14 TP / 46 FN
- Rule 3: KID Lenient cut-off, both levels: 10 TP / 50 FN


A Overlap between Rule 1 deletion confirmations, deleted Kinases and Kinase-Protein pairs in KB (all sources combined)

B Overlap between Rule 2 deletion confirmations, deleted Kinases and Kinase-Protein pairs in KB (KIDB lenient)

C Overlap between Rule 3 deletion confirmations, deleted Kinases and Kinase-Protein pairs in KB (KIDB lenient)

Figure 3: Figure containing Venn Diagrams depicting the ground truth (true positives) and false negatives captured by the top performing *rule* iterations and background knowledge levels considered. In all three Venn diagrams (A,B and C) the colour scheme is as follows: Purple for True Positives, light yellow for False Negatives, light blue for background knowledge kinase protein target pairs and light pink for gene deletion kinases. A: *Rule* iteration 1 with background knowledge at the protein level and from all sources combined had the best recall with 62 True Positives identified out of a potential 78. B: Rule iteration 2, Kinase Interaction database lenient cut-off and phosphosite level information with 14 True Positives and 46 False Negatives out of 60 potential. C: The third iteration of the rule performed best when the background knowledge was sourced from KID with lenient cutoff at 10 True Positives, 50 False Negatives out of a potential 64. The difference in potential totals comes can be attributed to how many of the deleted kinases were present and therefore queriable in each instance of background knowledge source and protein or phosphosite level consideration.

3.1.1 Explaining the 17 missed deletions with a new rule

From the above it was decided to further focus on explaining the results obtained from *rule* iteration 1, with the results of both levels included and all background knowledge sources combined. However, as can be seen in the triple Venn diagram of Figure 3 A, there were 17 kinase deletions the *rule* iteration was not able to identify. As this iteration did not exclude shared targets between kinases a possible hypotheses is that other kinases are acting in a compensatory manner. For this the 'ykinasecompcheck/2' *rule* was developed.

In brief, it checks whether a kinase (K2) increases its activity when a different kinase (DelK1) is silenced, in order to compensate for the loss of K1 activity and maintain the overall integrity of the signalling pathway. Specifically the Prolog notation the *rule* took the following form:

```
1 ykinasecompcheck(DelK1, K2) :-
2     (perturbs(DelK1,_Pst,Tprot1,down,_) ;
3     perturbs(DelK1,_Pst,Tprot1,unaffected,_)),
4     sharedKinasetarget(DelK1,K2,Tprot1),
5     knowntarget(K2, Pst1, Tprot1),
6     (perturbs(DelK2,Pst1,Tprot1,up,_),
7     (DelK2 \= K2, DelK2 \= DelK1)).
```

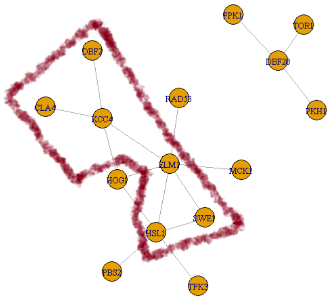
Specifically, the first goal 'perturbs/5' holds if the first argument (DelK1) perturbs the fourth argument (TProt1) in the direction specified by the fifth argument (down or unaffected). 'sharedKinasetarget/3' is a goal that is satisfied if the first argument (DelK1) and the second argument (K2) share the third argument (Tprot1) as a target. The information used for this, in order to maintain consistent background knowledge was the combination of all kinase protein background knowledge sources. 'knowntarget/3' is a goal that is satisfied if Tprot1 is a known target of the K2 at the phosphosite specified by the second argument (Pst1). Finally, (6) holds if the first argument (DelK2) perturbs the second argument (Pst1) in the direction specified by the fifth argument (up). The condition is true if DelK2 is different from both K2 and DelK1. The last goal ensures that kinase K2 indeed phosphorylates phosphosite Pst1 (in the condition it, itself is not deleted).

Using the above rule and the list of 17 missed deleted Kinases, for each, a further list of compensatory kinases was collated. These lists where then used to build compensatory kinase interaction networks in order to visualise the effect of kinase deletions on known kinase interaction networks. Predicted or compensatory pathways were compared with pathways reconstructed from literature [11]. Visualisations were carried out in Rstudio using the igraph package.

None of the false negatives (kinase deletions not present in the *rule* predictions) were present in the pathways associated with endocytosis, transcription response and mating. Cell cycle had the largest number of kinases compensating for ones from within our list. In Figure 4 Cell Cycle pathway is highlighted with a red border, the High-Osmorality glucose (HOG) pathway with a blue border and the meiosis with a fuchsia border. They are all connected canonically, showing the relative accuracy of the *rule* described above in picking up the shift in pathway activity to account for the deletion of a given kinase within the pathway. For the highlighted pathways the majority of their constituent kinases are present, deletions of which the first iteration of the *rule* did not pick up. This indirectly suggests that in response to certain key deletions these highlighted pathways are able to rewire themselves in an effort to counteract the perturbation.

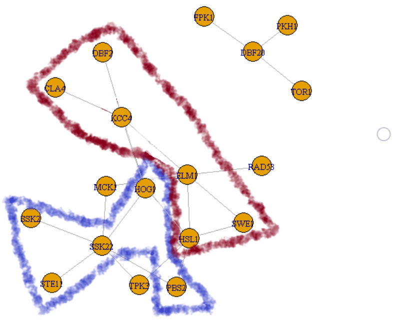
A

Predicted compensatory kinase network for Cell Cycle pathway



B

Predicted compensatory kinase network for HOG and Cell Cycle pathways



C

Predicted compensatory kinase network for HOG and Meiosis pathways

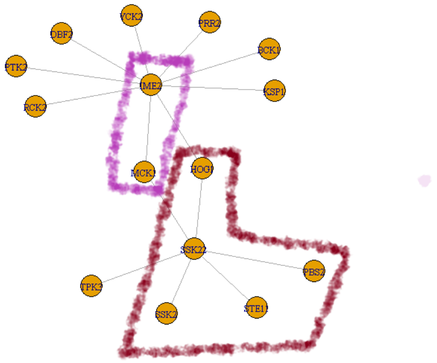


Figure 4: Predicted manner of kinase network rewiring following gene deletions. Kinases within pathways act in a compensatory manner. A) Cell Cycle pathway, predicted pathway rewiring. B) High-osmorality glucose and cell cycle pathway rewiring. C) High-osmorality and Meiosis pathway rewiring. Highlighted are the kinases belonging to the respective pathways that include False Negative i.e. missed gene deletions within our predictions.

4 Discussion

Despite *Saccharomyces cerevisiae* being extensively studied as a model system, there was a notable lack of coverage between kinase-phosphosite associations and the observational (phosphoproteomics) data obtained from the gene knockout study. Therefore implementing the methodology developed initially on the Toy Model, proved to be more challenging than anticipated.

The approach that yielded the most accurate results, in terms of recall, with the simplest iteration of the *rule*. The issue of poor overlap was also partially overcome by combining background knowledge sources and levels of granularity. Increasing complexity of *rules* did not yield better results.

Delving deeper into the False Negatives, i.e. kinases from verified deleted genes which we were unable to pick up, yielded interesting results. The interconnected nature of kinase interaction pathways was captured by the 'ykinasecompcheck/2' *rule*. Using the 17 kinases as a base we were able to identify and contextualise the majority of major large Yeast pathways, namely HOG and cell cycle as well as smaller one such as Meiosis. For the remaining pathways, their constituents were correctly captured as having been silenced. Combining the outputs of the individual silenced gene queries offers an insight into potential yeast signalling rewiring following strong perturbation such as silencing of a group of kinases.

The issue with overlap can be addressed by initiatives such as bioGRID and other datasets providing consistent information at both the protein and phosphosite level interactions. This problem is even greater and more present in the human kinome and proteome. However, combining resources, a crude approach at best, seemed to alleviate this, at least partially. A recently published study [5] made a significant contribution in assigning substrates to a large number of active Ser/Thr kinases (approximately 84%) in humans.

Next steps in applying this methodology would involve augmenting *facts* with probabilities, via the ProbLog2 [3] framework. Specifically, the immediate aim of this would be to capture the epistemic uncertainty in the kinase phosphosite associations as well as the mix of alleatoric and epistemic uncertainty that characterises the process of mass spectrometry based data acquisition and related post processing. The resulting output could further increase confidence in the predictions. Both of these require the mining of appropriate and well curated information in order to prove viable.

References

[1] Ashton Breitkreutz, Hyungwon Choi, Jeffrey R. Sharom, Lorrie Boucher, Victor Neduva, Brett Larsen, Zhen Yuan Lin, Bobby Joe Breitkreutz, Chris Stark, Guomin Liu, Jessica Ahn, Danielle Dewar-Darch, Teresa Reguly, Xiaojing Tang, Ricardo Almeida, Zhaohui Steve Qin, Tony Pawson, Anne Claude Gingras, Alexey I. Nesvizhskii & Mike Tyers (2010): *A global protein kinase and phosphatase interaction network in yeast*. *Science (New York, N.Y.)* 328(5981), doi:10.1126/SCIENCE.1176495. Available at `https://pubmed.ncbi.nlm.nih.gov/20489023/`.

[2] Amanda Clare & Ross D. King (2003): *Data Mining the Yeast Genome in a Lazy Functional Language*, pp. 19–36. doi:10.1007/3-540-36388-2_4. Available at `http://www.aber.ac.uk/compsci/Research/bio/dss/polyfarm/`.

[3] Anton Dries, Angelika Kimmig, Wannes Meert, Joris Renkens, Guy Van Den Broeck, Jonas Vlasselaer & Luc De Raedt (2015): *ProbLog2: Probabilistic logic programming*. doi:10.1007/978-3-319-23461-8_37. Available at `https://dtai.cs.kuleuven.be/problog`.

[4] Antonio Goncalves, Irene Ong, Jeffrey A. Lewis & Vitor Santos Costa (2014): *Towards using probabilities and logic to model regulatory networks*. *Proceedings - IEEE Symposium on Computer-Based Medical Systems*, pp. 239–242, doi:10.1109/CBMS.2014.9. Available at `https://ieeexplore.ieee.org/document/6881883`.

[5] Jared L. Johnson, Tomer M. Yaron, Emily M. Huntsman, Alexander Kerelsky, Junho Song, Amit Regev, Ting Yu Lin, Katarina Liberatore, Daniel M. Cizin, Benjamin M. Cohen, Neil Vasan, Yilun Ma, Konstantin Krismer, Jaylissa Torres Robles, Bert van de Kooij, Anne E. van Vlimmeren, Nicole Andrée-Busch, Norbert F. Käufer, Maxim V. Dorovkov, Alexey G. Ryazanov, Yuichiro Takagi, Edward R. Kastenhuber, Marcus D. Goncalves, Benjamin D. Hopkins, Olivier Elemento, Dylan J. Taatjes, Alexandre Maucuer, Akio Yamashita, Alexei Degterev, Mohamed Uduman, Jingyi Lu, Sean D. Landry, Bin Zhang, Ian Cossentino, Rune Linding, John Blenis, Peter V. Hornbeck, Benjamin E. Turk, Michael B. Yaffe & Lewis C. Cantley (2023): *An atlas of substrate specificities for the human serine/threonine kinome*. Nature 613(7945):4, doi:10.1038/s41586-022-05575-3. arXiv:2042.12345. Note.

[6] Ross D. King, Kenneth E. Whelan, Ffion M. Jones, Philip G.K. Reiser, Christopher H. Bryant, Stephen H. Muggleton, Douglas B. Kell & Stephen G. Oliver (2004): *Functional genomic hypothesis generation and experimentation by a robot scientist*. Nature 427(6971), pp. 247–252, doi:10.1038/nature02236. Available at `https://www.nature.com/articles/nature02236`.

[7] Jiaming Li, Joao A. Paulo, David P. Nusinow, Edward L. Huttlin & Steven P. Gygi (2019): *Investigation of Proteomic and Phosphoproteomic Responses to Signaling Network Perturbations Reveals Functional Pathway Organizations in Yeast*. Cell reports 29(7), p. 2092, doi:10.1016/J.CELREP.2019.10.034.

[8] G. Manning, D. B. Whyte, R. Martinez, T. Hunter & S. Sudarsanam (2002): *The protein kinase complement of the human genome*. Science (New York, N.Y.) 298(5600), pp. 1912–1934, doi:10.1126/SCIENCE.1075762. Available at `https://pubmed.ncbi.nlm.nih.gov/12471243/`.

[9] Rose Oughtred, Jennifer Rust, Christie Chang, Bobby Joe Breitkreutz, Chris Stark, Andrew Willems, Lorrie Boucher, Genie Leung, Nadine Kolas, Frederick Zhang, Sonam Dolma, Jasmin Coulombe-Huntington, Andrew Chatr-aryamontri, Kara Dolinski & Mike Tyers (2021): *The BioGRID database: A comprehensive biomedical resource of curated protein, genetic, and chemical interactions*. Protein science : a publication of the Protein Society 30(1), doi:10.1002/PRO.3978. Available at `https://pubmed.ncbi.nlm.nih.gov/33070389/`.

[10] Jason Ptacek, Geeta Devgan, Gregory Michaud, Heng Zhu, Xiaowei Zhu, Joseph Fasolo, Hong Guo, Ghil Jona, Ashton Breitkreutz, Richelle Sopko, Rhonda R. McCartney, Martin C. Schmidt, Najma Rachidi, Soo Jung Lee, Angie S. Mah, Lihao Meng, Michael J.R. Stark, David F. Stern, Claudio De Virgilio, Mike Tyers, Brenda Andrews, Mark Gerstein, Barry Schweitzer, Paul F. Predki & Michael Snyder (2005): *Global analysis of protein phosphorylation in yeast*. Nature 2005 438:7068 438(7068), doi:10.1038/nature04187.

[11] Sara Sharifpoor, Alex N. Nguyen Ba, Ji Young Young, Dewald van Dyk, Helena Friesen, Alison C. Douglas, Christoph F. Kurat, Yolanda T. Chong, Karen Founk, Alan M. Moses & Brenda J. Andrews (2011): *A quantitative literature-curated gold standard for kinase-substrate pairs*. Genome Biology 12(4), doi:10.1186/GB-2011-12-4-R39. Available at `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3218865/`.

[12] K. E. Whelan & R. D. King (2008): *Using a logical model to predict the growth of yeast*. BMC Bioinformatics 9(1), pp. 1–16, doi:10.1186/1471-2105-9-97/TABLES/12. Available at `https://link.springer.com/articles/10.1186/1471-2105-9-97https://link.springer.com/article/10.1186/1471-2105-9-97`.

Reliable Natural Language Understanding with Large Language Models and Answer Set Programming

Abhiramon Rajasekharan, Yankai Zeng, Parth Padalkar, Gopal Gupta
University of Texas at Dallas
Richardson, USA
{abhiramon.rajasekharan, yankai.zeng, parth.padalkar, gupta}@utdallas.edu

Humans understand language by extracting information (meaning) from sentences, combining it with existing commonsense knowledge, and then performing reasoning to draw conclusions. While large language models (LLMs) such as GPT-3 and ChatGPT are able to leverage patterns in the text to solve a variety of NLP tasks, they fall short in problems that require reasoning. They also cannot reliably explain the answers generated for a given question. In order to emulate humans better, we propose STAR, a framework that combines LLMs with Answer Set Programming (ASP). We show how LLMs can be used to effectively extract knowledge—represented as predicates—from language. Goal-directed ASP is then employed to reliably reason over this knowledge. We apply the STAR framework to three different NLU tasks requiring reasoning: qualitative reasoning, mathematical reasoning, and goal-directed conversation. Our experiments reveal that STAR is able to bridge the gap of reasoning in NLU tasks, leading to significant performance improvements, especially for smaller LLMs, i.e., LLMs with a smaller number of parameters. NLU applications developed using the STAR framework are also explainable: along with the predicates generated, a justification in the form of a proof tree can be produced for a given output.

1 Introduction

The long-term goal of natural language understanding (NLU) research is to build systems that are as good as humans in understanding language. This is a challenging task since there are multiple skills that humans employ to understand a typical sentence. First, a person needs to be proficient in the language to be able to interpret the sentence and understand its surface-level meaning. Second, they need to be able to interpret the meaning of the sentence in the current context, using the commonsense knowledge they already possess. This helps resolve ambiguities in the sentence and assess if any information is missing. Third, if required, they should be able to pose a question that would seek to fill in any information that is missing. Finally, once they attain a complete understanding of the sentence, they should be able to explain what they understood. We believe that all of these skills are also important for an NLU system that seeks to reliably answer questions or hold a conversation with a human.

In recent years, Large Language Models (LLMs) have been trained on massive amounts of text extracted from the internet. They have shown language proficiency to the extent that they are able to perform reading comprehension, translate languages, and generate text to complete stories, poems, or even code ([7, 9]). However, they can fall short when applied to problems that require complex reasoning. When tested on commonsense reasoning or mathematics word problems, LLMs such as GPT-3 have been shown to make simple reasoning errors ([12]). Though such errors may be mitigated with strategies such as chain-of-thought prompting ([28]), they continue to make mistakes that originate from calculation errors or missing reasoning steps in the solution, making it difficult to rely completely on such systems. While it is possible to prime LLMs to generate explanations for their answers, they sometimes generate

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 274–287, doi:10.4204/EPTCS.385.27

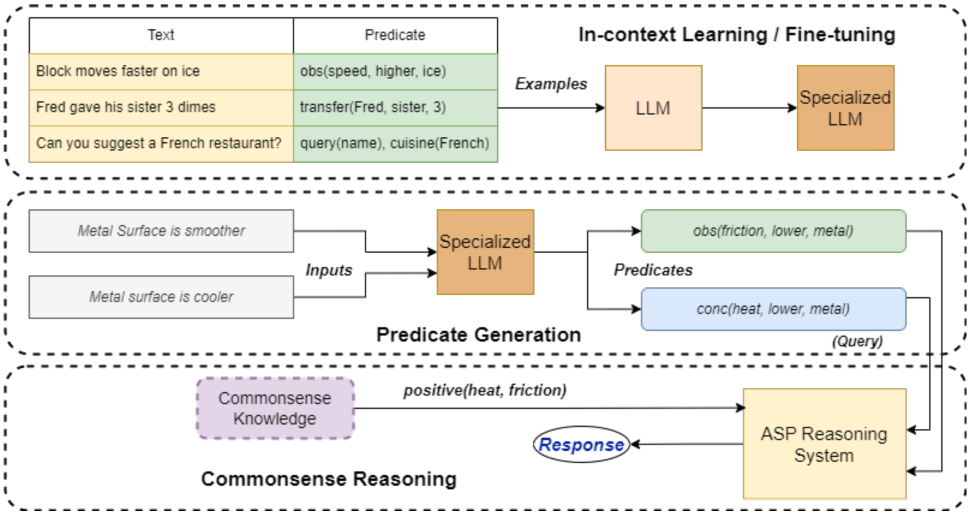


Figure 1: STAR framework Design

the right explanation along with a wrong answer and vice versa ([28]). This brings into question the dependability of such explanations. The lack of a clear separation of the reasoning process also makes it difficult to assess the models' state of knowledge and identify commonsense knowledge that needs to be integrated as necessary. These shortcomings point to the need for better NLU systems that use explicit reasoning.

With this motivation, we propose the STAR (Semantic-parsing Transformer and ASP Reasoner") framework that closely aligns with the way human beings understand language. STAR maps a sentence to the semantics it represents, augments it with commonsense knowledge related to the concepts involved—just as humans do—and then uses the combined knowledge to perform the required reasoning and draw conclusions (see Figure 1). The STAR framework relies on LLMs to perform semantic parsing (converting sentences to predicates that capture their semantics) and shifts the burden of reasoning to an answer set programming (ASP) system ([14, 6]). For our experiments, we use variants of GPT-3 ([7]) to generate predicates from the text. LLMs can be taught to do this either using fine-tuning or in-context learning using a small number of text-predicate pairs, resulting in a 'Specialized LLM'. Commonsense knowledge related to these predicates is coded in advance using ASP. Depending on the problem, a query is either pre-defined or can also be similarly generated from the problem using LLMs. The query is executed on the s(CASP) ([2]) goal-directed ASP system against the LLM-generated predicates and ASP-coded commonsense knowledge to generate a response.

In this paper, we use the STAR framework for three different NLU applications: (i) a system for solving qualitative reasoning problems, (ii) a system for solving math word problems, and (iii) a system representing a hotel concierge that holds a conversation with a human user who is looking for a restaurant recommendation. All three tasks require different types of reasoning. Qualitative reasoning and mathematical reasoning tasks require the system to perform a few steps of reasoning involving qualitative relationships and arithmetic operations, respectively. On the other hand, the conversation bot task requires the system to interact with the user to seek missing information, "understand" user requirements, and reason over it.

Our experiments involve two main variants of GPT-3; Davinci ($\sim$ 175B parameters) and Curie ($\sim$ 6.7B parameters). To measure the performance with STAR, we perform direct answer prediction using both models and compare them to the corresponding answers produced using our framework. The results show that STAR shows an increase in answer prediction accuracy and the difference is especially large for the smaller LLM, which might be weaker at reasoning. In both question-answering tasks, we are able to produce proof trees for the generated response, making them *explainable*. The knowledge predicates also help us understand the shortcomings and potential design improvements, which is not possible when the models are run using the LLMs alone. In the conversation bot task that requires in-depth reasoning, we observe that STAR provides much better control wrt seeking information from the user to understand their requirements. When used on its own for the purpose of restaurant recommendation, Davinci sometimes alters restaurant information based on user interaction. However, our approach always gives restaurant suggestions faithfully based on the database of restaurants available, making it more *reliable*. Since reasoning is performed using s(CASP) in our approach, we can also handle an arbitrarily large database of restaurants. This is not possible when LLMs are used end-to-end for this conversation bot, as there is a limit on the maximum prompt size. Thus, our approach can scale easily to larger restaurant databases.

2 Background

Large Language Models: Until recently, transformer-based deep learning models have been applied to NLP tasks by training and fine-tuning them on task-specific datasets ([8]). With the advent of Large Language Models, the paradigm changed to teaching a language model any arbitrary task using just a few demonstrations, called *in-context learning*. Brown et al. ([7]) introduced an LLM called GPT-3 containing approximately 175 billion parameters that have been trained using a massive corpus of filtered online text, on which the well-known ChatGPT is based ([23])). The model was able to perform competitively on several tasks such as question-answering, semantic parsing ([26]), and machine translation. However, such LLMs tend to make simple mistakes in tasks such as semantic (commonsense) and mathematical reasoning ([12, 28]).

In our work, we use GPT-3 for semantic parsing and leave the reasoning part to ASP. We theorize that given the vast pre-training they go through, LLMs can be used to automatically extract knowledge inherent in the text, just like humans do. Our experiments confirm that Davinci and Curie are able to extract such knowledge as predicates from sentences—with high accuracy—after learning from a few example demonstrations. Thus, our experiments show that LLMs are able to extract, what linguists call, the *deep structure* of a sentence, given a sentence's *surface structure*.

Answer Set Programming and the s(CASP) system: The s(CASP) system (developed by Arias et al.[2]) is an answer set programming ([6]) system that supports predicates, constraints over non-ground variables, uninterpreted functions, and, most importantly, a top-down, query-driven execution strategy. These features make it possible to return answers with non-ground variables (possibly including constraints among them) and compute partial models by returning only the fragment of a stable model that is necessary to support the answer to a given query. The s(CASP) system supports constructive negation based on a disequality constraint solver and unlike Prolog's negation as failure and ASP's default negation, `not p(X)` can return bindings for `X` on success, i.e., bindings for which the call `p(X)` would have failed. Additionally, s(CASP) system's interface with a constraint solver (over reals) allows for sound non-monotonic reasoning with constraints (useful for solving algebra problems in one of the NLU applications we discuss later).

Complex commonsense knowledge can be represented in ASP and the s(CASP) query-driven predicate ASP system can be used for querying it ([15, 29, 14]). Commonsense knowledge can be emulated using (i) default rules, (ii) integrity constraints, and (iii) multiple possible worlds ([14, 15]). Default rules are used for jumping to a conclusion in the absence of exceptions, e.g., a bird normally flies unless it's a penguin. Default rules with such exceptions represent an elaboration-tolerant way of representing knowledge ([14]).

```
flies(X) :- bird(X), not abnormal_bird(X).
abnormal_bird(X) :- penguin(X).
```

Integrity constraints allow us to express impossible situations and invariants. For example, a person cannot sit and stand at the same time.

```
false :- person(X), sit(X), stand(X).
```

Finally, multiple possible worlds allow us to construct alternative universes that may have some of the parts common but other parts inconsistent. For example, the cartoon world of children's books has a lot in common with the real world (e.g., birds can fly in both worlds), yet in the former birds can talk like humans but in the latter they cannot.

Default rules are used to model a bulk of our commonsense knowledge. Integrity constraints help in checking the consistency of the information extracted. Multiple possible worlds allow us to perform assumption-based reasoning (for example, knowing that "Alice loves Bob", we could assume that either Bob also loves Alice or he does not).

A large number of commonsense reasoning applications have already been developed using ASP and the s(CASP) system ([21, 25, 11, 29]). In the three applications reported in this paper, we have kept the commonsense reasoning component simple, as our main goal is to illustrate our framework for combining LLMs and ASP to develop NLU applications that are explainable and reliable. Because of the use of ASP, it is also possible to detect inconsistencies or biases in the text by reasoning over the predicates extracted. Justification for each response can also be given, as the s(CASP) system can generate justifications as proof trees ([1]).

3 Qualitative Reasoning

Qualitative reasoning tests a model's ability to reason about the properties of objects and events in the World. Tafjord et al. ([27]) introduced the QuaRel dataset in order to test question answering about qualitative relationships of a set of physical properties, which forms a perfect test-bed for our approach. Our experimental results show that the STAR framework significantly improves the model accuracy compared to the cases where the LLMs are applied directly to question answering.

3.1 The QuaRel Dataset

The QuaRel dataset consists of 2771 questions designed around 19 different properties such as 'friction', 'heat', 'speed', 'time', etc. In order to answer these questions, one must account for the correlation between these properties. Each question has a certain observation made about the two worlds where a property has a higher (or lower) value in one world compared to the other. Based on this observation, a (commonsense) inference needs to be drawn about other related properties described in the two worlds. This inference helps pick one of the two choices as the answer for the given question [27].

A question from the dataset is given in example 3.1. In this example, the two worlds are 'Carpet' and 'Floor'. The observation made is that the *distance* traveled by a toy car is more in world1 (floor). From this, the model needs to infer that the resistance or *friction* would be higher in world2 (carpet), which should lead to picking option A as the answer.

Example 3.1:

```
Question: Alan noticed that his toy car rolls further on a wood
floor than on a thick carpet. This suggests that:
(world1: wood floor, world2: thick carpet)
       (A) The carpet has more resistance (Solution)
       (B) The floor has more resistance
```

Along with each question, Tafjord et al. provide a logical form that captures the semantics of the question and we use it to extract the predicates needed for our method ([27]). For the above question (example 3.1), the logical form given is as follows:

$$qrel(distance, higher, world1) \rightarrow qrel(friction, higher, world2) \; ; \; qrel(friction, higher, world1) \tag{1}$$

The predicate $qrel(distance, higher, world1)$ refers to the observation that the *distance* is higher in world1, while $qrel(friction, higher, world2)$ and $qrel(friction, higher, world1)$ refer to the conclusions drawn in the two answer options, respectively.

3.2 Predicate Generation Step

We use GPT-3 to convert the Quarel dataset's natural language question (including the two answers) into appropriate predicates. We *fine-tune* the two GPT-3 model variants named Davinci and Curie ([7]) on the QuaRel dataset, instead of just using *in-context learning*[1]. Fine-tuning performs better since the models can learn from all the examples in the training set. Such data is available only for this task. Our input prompt consists of the question (including answer options), followed by the world descriptions. The world descriptions are included to enable the model to link the two worlds to the ones in the predicates (*obs* and *conc*) that are generated in the output. The prompt and completion formats for fine-tuning are given below:

Prompt format:

```
<Question-Answers>\n world1:<world1>\n world2:<world2>\n\n##\n\n
```

Completion format:

```
obs(<p>, <h/l>, <w1/w2>) → conc(<p>, <h/l>, <w1/w2>) ;
            conc(<p>, <h/l>, <w1/w2>) <EOS>
```

where p is the *property* involved, h/l is the relation which can be either *higher* or *lower* and w1/w2 is either *world1* or *world2*. After fine-tuning on the training set using the prompt and completion pairs, we use the prompt to generate the completion during testing. The `<EOS>` token helps cut off the generation when apt, avoiding completions that are either too long or too short. The extracted *obs* and *conc* predicates are then used by the logic program to determine the correct answer.

3.3 Commonsense Reasoning Step

The commonsense knowledge required to answer the questions is encoded in ASP as facts and rules. First, we ground the 19 properties using facts such as,

```
property(friction).     property(heat).     property(speed).
```

Next, we define the relationships between the properties, including their positive correlations (denoted as qplus), negative correlations (denoted as qminus) and symmetry,

```
qplus(friction, heat).           qminus(friction, speed).
qplus(speed, distance).          qminus(distance, loudness).
positive(X, Y) :- qplus(X, Y).   negative(X, Y) :- qminus(X, Y).
positive(X, Y) :- qplus(Y, X).   negative(X, Y) :- qminus(Y, X).
```

---

[1] Fine-tuning an LLM involves using additional training data to refine the LLM for the task at hand; in-context learning refers to giving some examples from the training data, along with the question posed, to the LLM as a part of its input.

In the QuaRel dataset, we are only dealing with two worlds. Hence, if a property P is higher in world1, it must be lower in world2 and vice versa. We capture this logic using the *opposite* predicates and the rules below:

```
opposite_w(world1,world2).        opposite_v(higher,lower).
opposite_w(world2,world1).        opposite_v(lower,higher).
conc(P, V, W) :- obs(P, Vr, Wr), property(P),
         opposite_w(W,Wr), opposite_v(V,Vr).
```

In order to capture the relationship between each pair of properties, we need to account for 4 different cases that may arise. If properties P and Pr are positively correlated, then (i) if P is higher in world W, Pr must also be higher in W, and (ii) if P is higher in world W, Pr must be lower in the other world Wr. Similarly, if P and Pr are negatively correlated, then (i) if P is higher in world W, Pr must be lower in W, and (ii) if P is higher in world W, Pr must be higher in the other world Wr. Note that the higher/lower relations may be swapped in all cases above. These 4 possible scenarios can be encoded in logic using the following rules:

```
conc(P,V,W) :- obs(Pr,V,W), property(P), property(Pr),
        positive(P,Pr).
conc(P,V,W) :- obs(Pr,Vr,Wr), property(P), property(Pr),
        opposite_w(W,Wr), opposite_v(V,Vr), positive(P,Pr).
conc(P,V,W) :- obs(Pr,Vr,W), property(P), property(Pr),
        opposite_v(V,Vr), negative(P,Pr).
conc(P,V,W) :- obs(Pr,V,Wr), property(P), property(Pr),
        opposite_w(W,Wr), negative(P,Pr).
```

Using this knowledge base, asserting a fact as an observation (*obs*) allows us to check for the correct conclusion (*conc*) that is entailed. For the example question in example 3.1, we can arrive at the answer by checking for entailment of the two possible conclusions as shown:

$assert(obs(distance, higher, world1)), conc(friction, higher, world2). \rightarrow\ True$

and

$assert(obs(distance, higher, world1)), conc(friction, higher, world1). \rightarrow\ False$

3.4 Results and Evaluation

We compare the results of our models to those reported by Tafjord et al. ([27]) in Table 1. Accuracy for four QuaRel datasets is considered ($QuaRel^F$ refers to the subset of the dataset which only focuses on friction-related questions). The first 8 rows show the accuracy of the baseline models proposed in the QuaRel paper. Curie-Direct and Davinci-Direct rows report the performance of Curie and Davinci models which directly predict the answer after fine-tuning on the QuaRel's training set. The Curie-STAR and Davinci-STAR rows show the performance for our approach, i.e., first generating the predicates and then reasoning using ASP and commonsense knowledge. The values in bold represent the highest accuracy values obtained for each dataset.

The results show a large improvement in the accuracy of the Curie model on all four QuaRel datasets. Table 1 shows that Davinci-STAR either matches or exceeds the performance of Davinci-Direct on three of the four QuaRel datasets. However, interestingly, we see that Davinci-Direct outperforms Davinci-STAR on the QuaRel-Dev dataset. Since our framework is explainable, we were able to analyze the cases where our approach makes a mistake. We found that the LLM sometimes generates properties that are not in the domain for some predicates (such as 'smoke' instead of 'heat' since the question mentions smoke). We hypothesize that this is because similar examples were not seen during training. QuaRel has a larger number of questions based on friction, which we believe led to our framework performing better on *QuaRel*^{F} datasets (which contain solely friction-based questions). Similarly, adding more examples for other properties might help our framework bridge the gap for other properties. Clearly, there is a stark difference between Curie and Davinci when used with our framework. We infer from this that while Davinci has some ability to reason, Curie lacks the reasoning skill required for the task and our approach helps bridge this reasoning gap.

Table 1: Comparison of accuracy of models on the QuaRel Dataset (Qualitative Reasoning)

No.	Model	QuaRel Dev	QuaRel Test	*QuaRel*^{F} Dev	*QuaRel*^{F} Test
1.	Random	50.0	50.0	50.0	50.0
2.	Human	96.4	-	95.0	-
3.	IR	50.7	48.6	50.7	48.9
4.	PMI	49.3	50.5	50.7	52.5
5.	Rule-Based	-	-	55.0	57.7
6.	BiLSTM	55.8	53.1	59.3	54.3
7.	QUASP	62.1	56.1	69.2	61.7
8.	QUASP+	68.9	68.7	79.6	74.5
9.	Curie-Direct	67.6	63.5	45.7	52.7
10.	Curie-STAR (ours)	86.2	85.2	87.9	85.9
11.	Davinci-Direct	**93.1**	**90.5**	90.0	91.3
12.	Davinci-STAR (ours)	90.6	**90.5**	**90.6**	**93.5**

4 Solving Word Problems in Algebra

Solving word problems in algebra requires extracting information from the question (interpreting its language) and performing mathematical reasoning to come up with an answer. Hence, it forms a good experiment to test our framework. We choose a specific type of addition and subtraction problems from the dataset used by Koncel-Kedziorski et al. ([18]). We define the predicates `has/4`, `transfer/5` and `total/4` as shown below to encode the knowledge in the problems:

```
has(entity, quantity, time_stamp, k/q).
transfer(entity1, entity2, quantity, time_stamp, k/q).
total(entity, quantity, time_stamp, k/q).
```

The predicate `has/4` defines that an `entity` has a certain `quantity` of some objects, at a particular `time_stamp`. The `transfer/5` predicate defines that an `entity1` has transferred a certain `quantity` of objects, to `entity2` at a particular `time_stamp`. Finally, the `total/4` predicate defines that an `entity` has a total amount of some objects equal to the `quantity`, at a particular `time_stamp`. The last term in each predicate is either the current knowledge (denoted as `k`) or a placeholder for the query (denoted `q`). We design these based on what information a human might glean from the problem in order to solve it.

The computation of the answer is done by simple s(CASP) rules. The rules are not shown due to lack of space and can be thought of as commonsense knowledge required to solve simple Algebra word problems given the `has/4`, `transfer/5`, and `total/4` predicates. An example problem and corresponding predicates generated to represent the knowledge are shown below.

Ex 1: *Joan found 70 seashells on the beach. Joan gave Sam some of her seashells. Joan has 27 seashells left. How many seashells did Joan give to Sam?*

```
has(joan,70,0,k).  transfer(joan,sam,X,1,q).  has(joan,27,2,k).
```

Following the STAR approach, we convert the knowledge in the chosen algebraic problem to the predicates defined above using an LLM. The predicates thus obtained (including the query) along with the rules then constitute the logic program. The query predicate is then executed against the program to solve the word problem.

4.1 Experiments and Results

Our dataset contains 91 problems drawn from a collection of word problems provided by Koncel-Kedziorski et al. ([18]). Since we hand-craft the rules for the domain, we select a set of problems that have a similar logic that we can encode. We use text-davinci-003 which is the most capable GPT-3 model for in-context learning. We did not use the text-curie-001 model as done in the qualitative reasoning experiment because the model requires fine-tuning on a larger set of questions to be effective. We provide a context containing a few problems with their corresponding predicates to the GPT-3 models and then use each problem as a prompt along with the context for the model to generate the facts and the query predicate(s) corresponding to the new problem. We then use the commonsense rules we defined along with the generated predicates (facts) as the logic program and query the program using the query predicate. We then compare the answer generated by the logic program with the actual, human-computed answer for each problem. As a baseline, we use the GPT-3 model for direct answer prediction. Here, with the algebra problems as the context, we provide the correct answer as the expected completion.

For our experiments, we initially started with a smaller context of 12 problems and examined the mistakes the LLM was making in generating the predicates. Since our approach is explainable (unlike the direct answer prediction approach), we were able to analyze the mistakes and added more problems to the context that might fix them. Repeating this process a few times, we end up with 24 problems as the final context for the GPT-3 model. Results of our experiments are shown in table 2.

Table 2: Performance comparison between the baseline model and our approach

Model	Accuracy
text-davinci-003-Direct	**1.00**
text-davinci-003-STAR	**1.00**

Both text-davinci-003-Direct and text-davinci-003-STAR result in 100% accuracy on the test set of 67 problems. Our experiments show that algebraic word problems that require simple reasoning can be solved easily by large LLMs. While our STAR approach achieves the same accuracy, it can also generate justification, making it explainable. Below we show a justification tree generated by s(CASP) ([1]) for the problem shown in Example 1 above, where the computed answer is 43.

```
JUSTIFICATION_TREE:
transfer(joan,sam,43,1,q) :-
    has(joan,70,0,k),
    has(sam,27,2,k),
    43 #= 70-27.
global_constraint.
```

A major weakness of LLMs is that they perform poorly on mathematical tasks. Since the computation in our approach is done by the reasoning process externally (as seen above), it circumvents any potential calculation mistakes the LLMs might make. The set of problems we selected ended up being simple enough for the baseline model to also learn. We believe that the performance difference will be evident with more complex algebraic problems, similar to our experiments on qualitative reasoning. We plan to explore this in our future work.

5 Concierge Bot: A Goal-Directed Conversational Bot

In this section, we describe the conversational bot that we have developed using our STAR framework. We first leverage LLMs to extract implicit and explicit knowledge predicates from a user's dialogue and then reason over the extracted predicates using s(CASP) to satisfy the user's request. To illustrate the applicability of our method to conversational AI, we narrow the domain of the bot to a very specific problem: restaurant recommendations. This ensures that commonsense knowledge that needs to be modeled is within a limited domain that can be expanded later as necessary. We call this bot the concierge bot. Since the conversational bot is an interactive application, the commonsense reasoning involved is more complex than the other two tasks. Here, the reasoner also has to keep track of the state of the conversation. In order to make a restaurant recommendation, the bot needs to ask the user for information (preferred cuisine, price range, etc.) and recommend a restaurant once all information is procured. At any stage, the bot can explain why it's asking for a given piece of information as well as the reason for its recommendation. The benefits of the STAR framework are more apparent in this application. The bot can ask user questions to acquire more information when missing or to verify if any errors are made in predicate generation. We show an example transcript of an interaction session below of the concierge bot with a human user. The English responses are generated using GPT-3 again after learning from predicate-text demonstrations (essentially, using the reverse of the text-to-predicate generation process).

```
Bot:   Hi, what can I assist you with?
Human: Can you help me find a place for food with curry?
       I don't want a pricey one.
Bot:   Are you looking for a place with a specific customer rating?
Human: No, I don't mind the rating.
Bot:   What kind of restaurant would you like to visit?
Human: A normal restaurant.
Bot:   Would you like somewhere family-friendly for tonight?
Human: No. Just for myself.
Bot:   Unfortunately, we cannot provide the results to your request.
Human: How about one with a high price?
       But it should be then at least above average quality.
Bot:   Are you looking for a Thai restaurant?
       The Rice Boat may be suitable but it can be pricey and
       customer feedback has been mixed. Additionally, it may
       not have the best atmosphere for families.
Human: Yes, that's what I need! Tell me where it is.
Bot:   The Rice Boat, located on 901 Bangkok Road, has an average
       customer rating and offers Thai cuisine at a premium cost.
       Unfortunately, it is not suitable for children.
Human: Great! Thank you for the service!
Bot:   It's no problem, I'm happy to assist.
```

5.1 Predicate Generation using LLMs

We adapt examples from the E2E dataset released by Novikova et al. ([22]) for our use case. We employ in-context learning where we give the LLM examples of pairs containing a natural language sentence and the corresponding restaurant predicates. We use the following six predicates from the dataset: **restaurant-name**, **eatType** (as establishment type), **food** (as cuisine), **priceRange**, **customerRating**, **familyFriendly**.

We use the STAR framework with in-context learning where we provide the GPT-3 model with 11 selected examples from the dataset, which covers all the predicates along with their possible arguments. This ensures that the LLM is aware of every possible predicate as well as every possible argument value these predicates can take. To assess the viability of LLMs for the predicate generation task, we tested the model using the first 500 examples in the E2E training set and obtained an accuracy of 89.33%. The accuracy metric we use is designed to account for the generation of correct predicates as well as arguments. The high predicate generation accuracy supports the feasibility of using our STAR framework for the concierge bot. Our framework can similarly be applied, to build any robust domain-specific conversational bots such as a front desk office receptionist or an airline reservation assistant.

5.2 Concierge Bot System Construction

To make GPT-3 better understand the meaning of each predicate, we first change the predicate names in E2E as follows: **restaurnt-name**, **typeToEat**, **cuisine**, **priceRange**, **customerRating**, **familyFriendly**. We also add two predicates **address** and **phoneNumber** to record the location and contact information for the user's query. An external predicate **prefer** is also added to capture the user's preference (such as curry, spicy, etc.) The information asked by the user is expressed by the value "query". We specialized GPT-3 with about a dozen example sentences along with the corresponding predicate(s). Below we show some examples of the sentences and the predicates generated after this specialization.

```
Sentence: Fitzbillies coffee shop provides a kid-friendly venue for
    Chinese food at an average price point in the riverside area.
    It is highly rated by customers.
Predicates: restaurant-name(Fitzbillies), typeToEat(coffee shop),
            cuisine(Chinese), priceRange(moderate),
            customerRating(high), familyFriendly(yes)

Sentence: Can you find a place for food at a low price? Both English
    and French cuisine is fine for me.
Predicates: restaurant-name(query), cuisine([Engish, French]),
            priceRange(cheap)
```

Commonsense knowledge involved in making a restaurant recommendation is coded using s(CASP). The interactive bot will take in the user's response and convert it to predicates using GPT-3. The predicates become part of the state. At this stage, we check for user preference. For example, if the user wants curry, Indian and Thai cuisine would be automatically added to the state through appropriate rules. The bot then examines the state to assess if all the information needed is present so that it can make a recommendation and if not, it will generate a question to ask the user for that information. This logic, shown in Figure 2, can be thought of as a state machine and has been referred to as a conversational knowledge template (CKT) by Basu et al. ([5]). The concierge bot determines which predicates are missing in its state to make a recommendation. One of the missing predicates is then selected and a query is created using it. Note that we use GPT-3 again to generate natural-sounding text from the predicate(s)

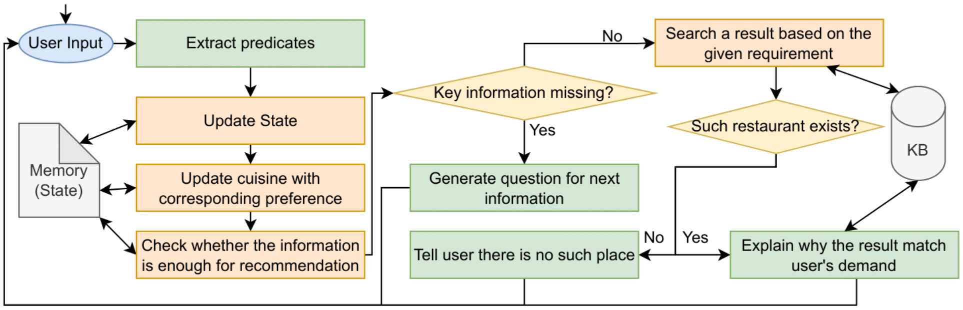


Figure 2: The framework of the reasoning system in Concierge Bot. The green boxes indicate the steps done by LLMs and the orange ones indicate the steps done by s(CASP).

corresponding to the response that our bot computes. The users can also change their preferences during the conversation and our bot can handle that. Take the conversation mentioned above as an example. When the user asks “Can you help me find a place for food with curry? I don’t want a pricey one.”, following predicates are generated by the GPT-3 text-davinci-003 model: *“restaurant-name(query), prefer(curry), priceRange([cheap, moderate])”*. The predicates are then added to the memory of the bot, where the log of the user requirements of the current conversation is stored. Note that the predicates *“prefer(curry)”* and *“cuisine([indian,thai])”* are also added to the state using the specific commonsense rule we wrote for this purpose (not shown due to lack of space). After the state gets updated, the bot checks to see whether any other key information is missing using another commonsense rule (not shown also due to lack of space). If no information is missing, the bot comes to the recommendation step where it searches for the appropriate restaurant in its database, consistent with the user’s input, and outputs it as a recommendation.

Since we use GPT-3 to generate predicates from user utterances, the user has complete freedom in terms of what they say. GPT-3 is able to extract the appropriate predicates from the corresponding deep form of the sentence. GPT-3 is also used to generate the natural language response of the bot. As a result, the interaction appears quite natural, as the example above shows.

5.3 Experiments

To explore GPT-3’s ability to generate goal-directed conversation without the reasoning system that STAR contains, we tested the conversation directly on GPT-3 with the sample conversations as input. We showed GPT-3 three full paragraphs of conversations and asked GPT-3 to generate a similar conversation according to the user input. The database of restaurants used by our concierge bot is also included in the input to the GPT-3 model. An example interactive conversation directly with GPT-3 is as follows:

```
Bot:   Hello, what can I do for you today?
Human: I’d like to have a family meal with my children.
Bot:   What type of cuisine would you like?
Human: Mexican food is suitable for me.
Bot:   What is your desired price range for the place?
Human: At a higher price, please.
Bot:   I recommend The Wrestlers, a coffee shop serving Mexican
       cuisine at a moderate price range. It is family-friendly
       and has an average customer rating. The address is 470 Main Rd.
```

The responses given by GPT-3 in the above conversation are correct except for the price range. In the given database, the restaurant recommended only serves cheap food. Hence, GPT-3 modified the information to align with the user's request. GPT-3 also follows the given examples and asks about the cuisine and price, but does not request other information like our STAR framework does. This is because these questions are not motivated by missing information, unlike in our approach. This example shows that although GPT-3 used on its own as a conversational bot is able to generate natural-sounding sentences fluently, it is unreliable and does not understand the knowledge given. Bots developed using our framework do not face such problems because they employ explicit commonsense reasoning. The methodology we use to build the concierge bot is explained in more detail in the paper Zeng et. al. [32].

6 Related Work

A recent line of research on improving the reasoning capabilities of LLMs focus on prompt engineering. Wei et al. ([28]) show that generating a chain of thought before the answer leads to a significant improvement in performance in a variety of reasoning tasks. However, in some cases, a wrong reasoning chain can lead to the right answer or vice versa. Zelikman et. al. ([31]) extend this by generating rationales using a self-taught approach. While the above approaches focus primarily on machine learning, our approach instead relies on s(CASP) to perform reasoning explicitly. This explicit reasoning is not only more reliable but is also explainable. Our approach falls into the line of Neuro-symbolic research that does heavy reasoning and light learning as categorized in the survey by Hamilton et al. ([16]). Typically such works try to integrate neural and symbolic components ([30, 24]). In contrast, we separate both components by using LLMs for predicate extraction and the s(CASP) system for reasoning.

This paper follows our earlier research where we advocate a combination of machine learning and commonsense reasoning to carry out intelligent tasks in a human-like manner ([4, 3, 19]). In the SQuARE question-answering system ([3]), knowledge was extracted using the Stanford CoreNLP Parser ([20]) and then mapped to templates from VerbNet. This method was only usable for simple sentences such as for the bAbI dataset. Along similar lines, Tafjord et al. ([27]) report models that convert problems in English to logical forms, which are then processed using a custom interpreter. The semantic parsers discussed in their paper are variations of LSTMs that generate CFG-like grammar rules which create the logic form. The main bottleneck of these two approaches was in the parsers used. In our work, we use LLMs that can extract predicates from arbitrary sentences which makes our approach applicable to more complex problems and translates to better performance. The closest works to ours are along the lines of Chen et al. ([10]) and Gao et al. ([13]) who use LLMs to generate program steps which are then executed in a programming language such as Python. Here the program steps still need to be generated by LLMs completely. Instead, our work delegates the entire reasoning task to the s(CASP) system. Since s(CASP) is designed for complex reasoning, our research can be extended more easily to complex text-based reasoning problems. Similar to our conversation bot, Inclezan et al. ([17]) use an ASP-based action language to reason in a restaurant setting, but it is not an interactive system like our work.

7 Conclusions and Future Work

In this paper, we described the STAR framework that combines LLMs and ASP for NLU tasks. We show that our system is reliable and explainable using three different reasoning tasks. For the qualitative reasoning task, STAR outperforms purely LLM-based approaches and advances the state-of-the-art wrt performance on most datasets in QuaRel. The performance difference is more significant for Curie, indicating that it helps bridge the reasoning gap in smaller LLMs. In all three tasks, STAR can explain its reasoning process by producing a justification tree. In the LLM-only approach for developing a concierge bot, we noticed that the LLM mixes up information collected during the conversation and leads to incorrect suggestions, while our STAR-based approach stays faithful to the information given in the restaurant database. Our approach also allows for holding long, interactive, and meaningful conversations.

The potential applications of our STAR framework are very broad. It can help in any NLU application that requires reasoning about knowledge in text or utterances. Some examples are, automatically extracting formal software requirements from textual specifications, building conversational agents for other domains, and reliable machine translation. We believe that performance improvement using STAR will be more pronounced for problems that require complex reasoning. We also plan to develop a general commonsense knowledge base that applications developed using the framework can employ.

Acknowledgment: Authors acknowledge support from NSF grants IIS 1910131, IIP 1916206, and a DARPA ARCOS award. We are grateful to Farhad Shakerin, Kinjal Basu, Joaquin Arias, Elmer Salazar, and members of the UT Dallas ALPS Lab for their discussions.

References

[1] Joaquín Arias, Manuel Carro, Zhuo Chen & Gopal Gupta (2020): *Justifications for Goal-Directed Constraint Answer Set Programming*. In: *Proceedings 36th ICLP (Tech. Comm.)*, *EPTCS* 325, pp. 59–72, doi:10.4204/EPTCS.325.12.

[2] Joaquín Arias, Manuel Carro, Elmer Salazar, Kyle Marple & Gopal Gupta (2018): *Constraint Answer Set Programming without Grounding*. *TPLP* 18(3-4), pp. 337–354, doi:10.1017/S1471068418000285.

[3] K. Basu, S. C. Varanasi, F. Shakerin, J. Arias & G. Gupta (2021): *Knowledge-driven Natural Language Understanding of English Text and its Applications*. In: *Proc. AAAI*, pp. 12554–12563, doi:10.1609/aaai.v35i14.17488.

[4] Kinjal Basu, Farhad Shakerin & Gopal Gupta (2020): *AQuA: ASP-Based Visual Question Answering*. In: *Proc. PADL*, Springer-Verlag, Berlin, Heidelberg, p. 57–72, doi:10.1007/978-3-030-39197-3_4.

[5] Kinjal Basu, Huaduo Wang, Nancy Dominguez, Xiangci Li, Fang Li, Sarat Chandra Varanasi & Gopal Gupta (2021): *CASPR: a commonsense reasoning-based conversational Socialbot*, doi:10.48550/arXiv.2110.05387.

[6] Gerhard Brewka, Thomas Eiter & Miroslaw Truszczynski (2011): *Answer Set Programming at a Glance*, doi:10.1145/2043174.2043195.

[7] Tom Brown, Benjamin Mann & Others (2020): *Language Models are Few-Shot Learners*. In: *NeurIPS*, 33, Curran Associates, Inc., pp. 1877–1901, doi:10.5555/3495724.3495883.

[8] Silvia Casola, Ivano Lauriola & Alberto Lavelli (2022): *Pre-trained transformers: an empirical comparison*, doi:10.1016/j.mlwa.2022.100334. Available at https://www.sciencedirect.com/science/article/pii/S2666827022000445.

[9] Mark Chen, Jerry Tworek & Others (2021): *Evaluating Large Language Models Trained on Code*, doi:10.48550/arXiv.2107.03374. Available at https://arxiv.org/abs/2107.03374.

[10] Wenhu Chen, Xueguang Ma, Xinyi Wang & William W. Cohen (2022): *Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks*, doi:10.48550/arXiv.2211.12588.

[11] Zhuo Chen, Kyle Marple & Others (2016): *A Physician Advisory System for Chronic Heart Failure management based on knowledge patterns*. *Theory Pract. Log. Program.* 16(5-6), pp. 604–618, doi:10.1017/S1471068416000429.

[12] Luciano Floridi & Massimo Chiriatti (2020): *GPT-3: Its Nature, Scope, Limits, and Consequences*, doi:10.1007/s11023-020-09548-1.

[13] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan & Graham Neubig (2023): *PAL: Program-aided Language Models*, doi:10.48550/arXiv.2211.10435.

[14] M. Gelfond & Y. Kahl (2014): *Knowledge representation, reasoning, and the design of intelligent agents: Answer Set Programming approach.* Cambridge Univ. Press, doi:10.1017/CBO9781139342124.

[15] Gopal Gupta (July 7, 2022): *Automating Common Sense Reasoning with ASP and s(CASP).* Technical Report, https://utdallas.edu/~gupta/csr-scasp.pdf.

[16] Kyle Hamilton, Aparna Nayak, Bojan Bož ić & Luca Longo (2022): *Is neuro-symbolic AI meeting its promises in natural language processing? A structured review.* Semantic Web, pp. 1–42, doi:10.3233/sw-223228.

[17] Daniela Inclezan (2019): *RestKB: A Library of Commonsense Knowledge about Dining at a Restaurant*, doi:10.4204/EPTCS.306.19.

[18] Rik Koncel-Kedziorski, Hannaneh Hajishirzi, Ashish Sabharwal, Oren Etzioni & Siena Ang (2015): *Parsing Algebraic Word Problems into Equations*, doi:10.1162/tacl_a_00160.

[19] Suraj Kothawade, Vinaya Khandelwal & Others (2021): *AUTO-DISCERN: Autonomous Driving Using Common Sense Reasoning*, doi:10.48550/arXiv.2110.13606.

[20] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard & David McClosky (2014): *The Stanford CoreNLP NLP Toolkit.* In: ACL System Demonstrations, pp. 55–60, doi:10.3115/v1/P14-5010.

[21] Jason Morris (2023): *Blawx: User-friendly Goal-Directed Answer Set Programming for Rules as Code.*

[22] Jekaterina Novikova, Ondřej Dušek & Verena Rieser (2017): *The E2E Dataset: New Challenges For End-to-End Generation.* In: Proc. SIGdial Meeting on Discourse and Dialogue, Association for Computational Linguistics, pp. 201–206, doi:10.18653/v1/W17-5525.

[23] OpenAI (2022): *Optimizing Language Models for Dialog.* https://openai.com/blog/chatgpt/.

[24] Ryan Riegel, Alexander Gray & Others (2020): *Logical Neural Networks*, doi:10.48550/arXiv.2006.13155.

[25] Galileo sartor, Jacinto Davila & Others (2022): *Integration of Logical English and s(CASP).*

[26] Richard Shin & Benjamin Van Durme (2022): *Few-Shot Semantic Parsing with Language Models Trained on Code.* In: Proc. NACL 2022: Human Language Technologies, ACL, Seattle, United States, pp. 5417–5425, doi:10.18653/v1/2022.naacl-main.396.

[27] Oyvind Tafjord, Peter Clark, Matt Gardner, Wen-tau Yih & Ashish Sabharwal (2019): *QUAREL: A Dataset and Models for Answering Questions about Qualitative Relationships*, doi:10.1609/aaai.v33i01.33017063.

[28] Jason Wei, Xuezhi Wang & Others (2022): *Chain of Thought Prompting Elicits Reasoning in Large Language Models*, doi:10.48550/arXiv.2201.11903.

[29] Zesheng Xu, Joaquín Arias & Others (2023): *Jury-Trial Story Construction and Analysis Using Goal-Directed Answer Set Programming.* In: Proc. PADL, LNCS 13880, Springer, pp. 261–278, doi:10.1007/978-3-031-24841-2_17.

[30] Zhun Yang, Adam Ishay & Joohyung Lee (2020): *NeurASP: Embracing Neural Networks into Answer Set Programming.* In Christian Bessiere, editor: Proc. IJCAI-20, International Joint Conferences on Artificial Intelligence Organization, pp. 1755–1762, doi:10.24963/ijcai.2020/243. Main track.

[31] Eric Zelikman, Yuhuai Wu, Jesse Mu & Noah Goodman (2022): *STaR: Bootstrapping Reasoning With Reasoning*, doi:10.48550/arXiv.2203.14465.

[32] Yankai Zeng, Abhiramon Rajasekharan, Parth Padalkar, Kinjal Basu, Joaquín Arias & Gopal Gupta (2023): *Automated Interactive Domain-Specific Conversational Agents that Understand Human Dialogs.*

Leasing the Cloud-Edge Continuum, à la Carte[*]

Stefano Forti
stefano.forti@unipi.it
University of Pisa
Pisa, Italy

Ahmad Ibrahim
a.ibrahim@bham.ac.uk
University of Birmingham
Birmingham, United Kingdom

Antonio Brogi
antonio.brogi@unipi.it
University of Pisa
Pisa, Italy

Next-gen computing paradigms foresee deploying applications to virtualised resources along a continuum of Cloud-Edge nodes. Much literature has focused on how to place applications onto such resources so as to meet their requirements. To lease resources to application operators, infrastructure providers need to identify a portion of their Cloud-Edge assets to meet application operators' requirements. This article proposes a novel declarative resource selection strategy, prototyped in Prolog, to determine a suitable infrastructure portion that satisfies all requirements. The proposal is showcased over a lifelike scenario.

1 Introduction

Recent utility computing paradigms (e.g. Fog, Edge, Mist, Cloud-IoT computing) foresee deploying applications to virtualised resources along a continuum of Cloud-Edge computing, storage and networking resources. Much literature (surveyed, for instance, by [13] and [17]) has focussed on how to place (multi-service) applications onto such resources so as to meet their stringent functional and non-functional requirements (e.g. in terms of latency, bandwidth, security, deployment location, Quality of Service (QoS), energy sources, costs). Application placement has been proven an NP-hard problem ([1]), featuring worst-case exp-time complexity of $O(N^S)$ for placing S application services onto an infrastructure of N nodes. Most of the proposed solutions (*i*) take the perspective of application operators that need to deploy their application services, and (*ii*) determine eligible placements by trying to match each service with any node in the infrastructure.

We tackle here a different, yet complementary, problem by (*a*) *taking the perspective of infrastructure providers* leasing Infrastructure-as-a-Service (IaaS) resources in the Cloud-Edge continuum to paying application operators that need to deploy their services, and by (*b*) *identifying a portion of the available Cloud-Edge infrastructure* that meets the requirements set by application operators and maximizes the expected profit that the infrastructure provider will get from leasing such infrastructure portion.

Said otherwise, we consider a setting in which:

(1) Application operators request to lease a Cloud-Edge continuum *portion*, featuring specific (hardware, software, security, network) *resources* and suitably guaranteeing *performance metrics* (e.g. sustainability, availability),

(2) Infrastructure providers identify a portion that meets all the requirements of applications operators and that maximises their profit, by exploiting an automated (virtual) *resource selection* mechanism,

---

[*] Research partly funded by projects: *Energy-aware management of software applications in Cloud-IoT ecosystems* (RIC2021PON_A18), funded with ESF REACT-EU resources by the *Italian Ministry of University and Research* through the *PON Ricerca e Innovazione 2014–20* and *hOlistic Sustainable Management of distributed softWARE systems (OSMWARE)*, UNIPI PRA_2022_64, funded by the University of Pisa, Italy.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 288–299, doi:10.4204/EPTCS.385.28

(3) Application operators can seamlessly deploy applications onto the leased infrastructure portion by employing any *appplication placement strategy* of their choice.

It is worth noting that such a change of perspective enables taming the exp-time complexity of the application placement problem by considering only a small portion of $n < N$ nodes in the available infrastructure. For instance, our experiments in [5] and [7] show how placement execution times grow exponentially at increasing sizes of the considered infrastructure and, therefore, how reducing decision-making to smaller portions of such an infrastructure brings considerable speed-ups, viz. from $2\times$ up to a $1000\times$ in realistic use cases.

This article presents a declarative strategy to automate the aforementioned step (2) by solving the following *resource selection* problem:

> *Let N be a set of heterogeneous nodes of a Cloud-Edge infrastructure managed by an infrastructure provider. Let $p : N \rightarrow \mathbb{R}$ be the function that defines the profit of the infrastructure provider for leasing a node. Let R be a request of resources of an application operator. A solution to the considered resource selection problem is a portion $C = \{n_1, n_2, \ldots n_m\} \subseteq N$ that guarantees all requirements in R while maximising the infrastructure provider profit. Formally, given N, p, and R, we aim at determining C such that*
>
> $$\underset{\substack{C \subseteq N \\ C \models R}}{\operatorname{argmax}} \sum_{n \in N} p(n).$$

We present our novel open-source prototype[1] fogCutter, entirely written in Prolog. fogCutter allows infrastructure providers to describe the capabilities of their nodes and application operators to input their requests with a declarative syntax – implying better readability, maintainability and extensibility. fogCutter will benefit both infrastructure providers and application operators. Infrastructure providers will identify suitable infrastructure portions to lease to maximise their profit. Application operators will reduce decision-making times for managing their distributed software by considering an infrastructure portion built *à la carte*, specially based on their deployment needs.

The rest of this article is organised as follows. We describe a motivating scenario (Sect. 2) and illustrate the fogCutter Prolog prototype (Sect. 3), which we then use over the motivating scenario (Sect. 4). Last, after discussing some related work (Sect. 5), we conclude pointing out to directions for future work (Sect. 6).

2 Motivating Scenario

Consider the Cloud-Edge continuum of Fig. 1, provided by a single infrastructure operator across different geographical regions (viz. US, EU, China). The infrastructure is made of 20 nodes: 10 access points (APs) `ap1`–`ap10`, 7 more edge gateways `n1`–`n7` and 3 Cloud datacentres `c1`–`c3`. Assume the application operator connects to the network through `ap3`.

Each node in the available Cloud-Edge infrastructure of Fig. 1 is characterised by its geographical location (represented as a flag), security capabilities (represented as icons), share of renewable energy it exploits to run (indicated by the colour of the node), and availability (reported in the bottom right side of each node). Finally, each node features a different amount of hardware and is associated to a profit value established by the infrastructure provider. Hardware and profit for each node are listed in Table 1.

---

[1] Available at `https://github.com/di-unipi-socc/fogCutter`

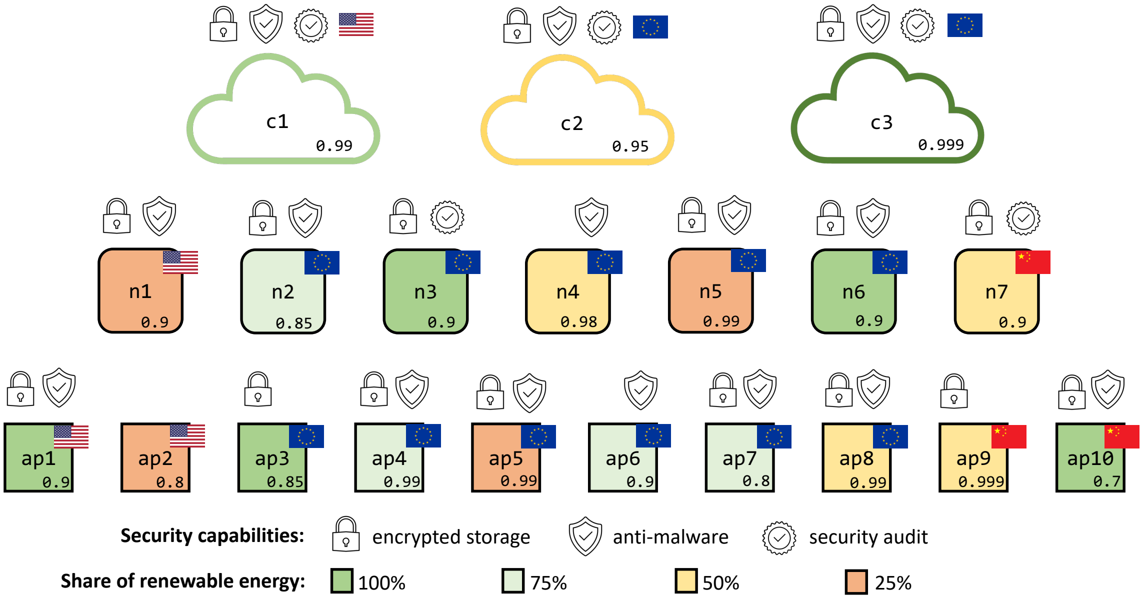


Figure 1: Available Cloud-Edge infrastructure.

As an example, the Cloud node `c2` is located in the EU, features encrypted storage, anti-malware and security audit capabilities, is powered by 50% of renewable energy, has an availability of 95%, features 16 free hardware units, and – when leased – brings a profit of 7 €/hour to the infrastructure provider.

Node Id	Hardware Units	Profit
ap1	2	2.5
ap2	4	1
ap3	3	1.75
ap4	4	3
ap5	2	2.5
ap6	2	1.5
ap7	4	3
ap8	1	2.25
ap9	3	1.75
ap10	8	4

Node Id	Hardware Units	Profit
n1	8	4
n2	12	5
n3	4	3
n4	12	4
n5	2	2.5
n6	8	4
n7	8	4
c1	24	9
c2	16	7
c3	12	6

Table 1: Node hardware and profit.

As they are deployed at different geographical locations and interconnected with different (wired and wireless) technologies, end-to-end latency and bandwidth between nodes vary depending on the type of node and its location. For instance, on average, APs in the same location reach out to each other with latency of 10 ms and average symmetric bandwidth of 100 Mbps; those at different locations incur in a latency of 150 ms with a bandwidth of 25 Mbps. Besides, APs reach out edge nodes in their location with a latency of 60 ms, an upload bandwidth of 30 Mbps and a download bandwidth of 200 Mbps (viz. 200/30 Mbps). Last, APs connect to Cloud nodes at their location experiencing 130 ms latency and a 90/15 Mbps connection; those at a different location are reached out to with a latency of 200 ms and a

25/10 bandwidth. Data about the QoS of all other end-to-end links can be found online[2].

An application operator aims at deploying a *Smart Traffic* application onto a portion of the available infrastructure. Smart traffic management is a classical use case for Cloud-Edge applications ([2, 15]), having to collect and process data from various sensors (i.e. live video feed from traffic cameras, speed radar, air quality monitor sensor) and services (weather, date-time, location) to detect traffic congestion, issue speeding fines, dynamically adjust the price of tollgates and speed limits, control traffic signal duration, and send alerts and notifications.

Assume that the application operator accesses the network from node `ap3`. To deploy her software artefacts, the application operator estimates she will need at least 20 free hardware units, reachable within 250 ms from node `ap3` with a 10 Mbps bandwidth. Not to incur in too high management costs (i.e. time to decide where to place/migrate her application services at deployment and runtime, respectively), she would like her Cloud-Edge portion to include at most 4 nodes (`ap3` included).

Receiving the above request, the infrastructure provider might wonder the following:

> **Q1.** *Is there a portion of the available Cloud-Edge infrastructure that features at least 20 hardware units, and is made of at most four nodes with end-to-end connections featuring at least 10 Mbps bandwidth and at most 250 ms latency?*

Before she can deploy her *Smart Traffic* application, the application operator realises she also would need to meet GDPR constraints and deploy the software only to resources in the EU, provided with an antimalware and encrypted storage capabilities. She consequently updates her request to the infrastructure provider, who must answer the following question:

> **Q2.** *Is there a portion of the available Cloud-Edge infrastructure that satisfies all requirements at* **Q1***, is fully located in the EU, and features antimalware and encrypted storage on all nodes?*

Lastly, the application operator also determines that she wants the resources onto which to deploy her application to feature an overall availability of 85% and to be likely to be powered by renewable energy at least 30% of the time. Therefore, considering this new request, the infrastructure provider will look for an answer to the following question:

> **Q3.** *Is there a portion of the available Cloud-Edge infrastructure that satisfies all requirements at* **Q1** *and* **Q2** *, and features 85% availability and 30% sustainability score?*

3 Our Methodology

In this section, we first illustrate how FOGCUTTER enables describing infrastructure capabilities and user requests with a simple and extensible model (Sect. 3.1). Then, we illustrate the logic program that, by processing such a model, determines solutions to the considered problem (Sect. 3.2).

3.1 Infrastructure and request model

Nodes are declared with a valid `N` (e.g. their IP address, a symbolic name within the DNS) by means of facts like

```
node(N).
```

---

[2] Available at: `https://github.com/di-unipi-socc/fogCutter/blob/main/scenarios/smarttraffic.pl`

Each node is associated to the profit that its lease brings to the infrastructure provider, as in

```
profit(N, Profit).
```

After declaring a node, it is possible to declare its capabilities[3] through facts like

```
nodeCap(N, CapabilityType, FeaturedCapabilities).
```

that specifies the `CapabilityType` and the currently `FeaturedCapabilities` for such a type at node `N`. For instance, node `c1` of the motivating scenario of Sect. 2 is declared as in:

```
node(c1).
nodeCap(c1, hardware, 24).
nodeCap(c1, location, us).
nodeCap(c1, security, [antimalware, encryptedStorage, audit]).
nodeCap(c1, availability, 0.99).
nodeCap(c1, sustainability, 0.75).
```

Similarly, the capabilities of end-to-end links between nodes `N` and `M` can be declared as facts like

```
linkCap(N, M, Capability, FeaturedCapabilities).
```

As an example, the end-to-end link connecting nodes `ap1` and `n1` in the previous example can be expressed through the following facts:

```
linkCap(ap1,n1,latency,60).     linkCap(ap1,n1,bandwidth,30).
linkCap(n1,ap1,latency,60).     linkCap(n1,ap1,bandwidth,200).
```

We distinguish two kinds of capability types: *local* and *global*. Local capabilities are checked on a single resource, either a node or a link. Such capability types are declared through facts like

```
capType(Capability, Resource, ComparisonOperator).
```

that specify the `Resource` for which such capability can be declared (viz. `node` s or `link` s), a `ComparisonOperator` to check whether the `FeaturedCapabilities` at a node satisfy a request on such `Capability`. For instance, fogCutter defines the following four local capabilities

```
capType(security,  node, supset).   capType(location,  node, member).
capType(latency,   link, smaller).  capType(bandwidth, link, greater).
```

that enable checking[4] the set of security capabilities featured by a node against a set of requested capabilities, the location of a node against a list of permitted locations, and the latency and bandwidth featured by communication links against an upper and a lower bound, respectively.

Global capabilities are instead checked against a portion of the available infrastructure and also need to be aggregated. They can be declared through facts like

```
capType(CapabilityType, Resource, ComparisonOperator, AggregationOperator).
```

---

[3] Node capabilities can be obtained by tools targeting Cloud-Edge infrastructure monitoring, e.g. [6].

[4] For the sake of conciseness, comparison operators are not listed here. The interested reader can find them in the online project repository.

that also specify a suitable `AggregationOperator` to compute them over a selected portion of the whole infrastructure. Again, FOGCUTTER features the following default capabilities

```
capType(hardware,       node,   smaller,    sum).
capType(availability,   node,   smaller,    product).
capType(sustainability, node,   smaller,    product).
```

that correspond to the total amount of hardware of an infrastructure portion, and to the overall availability[5] and sustainability[6] featured by such a portion. The model above is designed to be easily extensible by taking into account further node/link capabilities of arbitrary (local and global) types.

FOGCUTTER receives requests for Cloud-Edge portions through facts like

```
request(RequestId, SourceNode, MaxNodes, Requirements).
```

where `RequestId` is a unique identifier for the request, `SourceNode` is the identifier of the user node from which the request is issued, `MaxNodes` is the maximum number of nodes the user wants in her portion, and requirements is a list of pairs `(CapabilityType, R)` that specifies which global/local capability is needed and the target amount or threshold `R` that is requested by the user. For instance, the fact

```
request(req42, ap3, 3, [(hardware,20),
                        (latency,250),(bandwidth,10),
                        (security,[antimalware, encryptedStorage]), (location,[eu]),
                        (availability, 0.85), (sustainability, 0.3)]).
```

requires an infrastructure portion of at most 3 nodes in addition to `ap3`, with at least 20 hardware units, reachable through end-to-end link featuring at most 250 ms latency and at least 10 Mbps bandwidth, featuring antimalware and encrypted storage capabilities, located in the EU, and with at least 85% availability and 30% sustainability.

3.2 Resource Selection: a Logic Programming Strategy

Predicate `portion/3` inputs a `RequestId`, determines a `Portion` of the available Cloud-Edge continuum that satisfies all the request's requirements and associates the `Portion` with an estimate of the `Profit` that the infrastructure provider will get from leasing such `Portion`.

```
portion(RequestId, (Portion, Profit)) :-
    request(RequestId, N, MaxNodes, Reqs),
    splitRequirements(Reqs, NodeReqs, LinkReqs, GlobalReqs),
    portion(N, MaxNodes, NodeReqs, LinkReqs, GlobalReqs, [N], Portion),
    portionProfit(Portion, Profit).
```

Predicate `portion/3` first fetches the data (`N`, `MaxNodes`, `Reqs`) of the input `RequestId`. Then predicate `splitRequirements/4` simply partitions the list of request's requirements `Reqs` into three lists, depending on whether they refer to local capabilities of nodes (`NodeReqs`) or of links (`LinkReqs`), or on whether they are global requirements on the overall infrastructure portion (`GlobalReqs`).

Predicate `portion/7` determines a `Portion` of the infrastructure that contains node `N` and at most other `MaxNodes` nodes and satisfies all the `NodeReqs`, `LinkReqs` and `GlobalReqs` requirements.

---

[5] We define *availability* as the probability that a system is operational at a given time. It is obtained over a portion of an infrastructure by multiplying the availability of the single nodes in the portion.

[6] We define *sustainability* as the probability that a system is fully powered by renewable energy. It is obtained over a portion of an infrastructure by multiplying the sustainability of the single nodes in the portion.

```
portion(N, MaxNodes, NodeReqs, LinkReqs, GlobalReqs, I, NewI) :-
    MaxNodes > 0,
    \+ satisfiesGlobalReqs(GlobalReqs, I),
    node(M), \+ member(M,[N|I]),
    satisfiesNodeReqs(M,NodeReqs),
    satisfiesLinkReqs(M,N,LinkReqs),
    NewMaxNodes is MaxNodes-1,
    portion(N, NewMaxNodes, NodeReqs, LinkReqs, GlobalReqs, [M|I], NewI).
portion(_, _, _, _, GlobalReqs, I, I) :-
    satisfiesGlobalReqs(GlobalReqs, I).
```

Until all the global requirements `GlobalReqs` are satisfied, predicate `portion/7` recursively adds to the infrastructure portion a new node `M` that satisfies all the `NodeReqs` and `LinkReqs` local requirements.

Predicate `satisfiesGlobalReqs/2` verifies whether the (non empty) infrastructure portion determined so far satisfies the list `GlobalReqs` of global requirements.

```
satisfiesGlobalReqs(Rs, [I|Is]) :-  satisfies(Rs, [I|Is]).

satisfies([], _).
satisfies([(P,V,Op,Ag)|Rs], I) :-
    findall(Vm, (member(M,I),nodeCap(M,P,Vm)), Vs),
    aggregatedListValue(Vs,Ag,AV), compareValue(V,Op,AV),
    satisfies(Rs, I).

aggregatedListValue([V|Vs],Ag,AV):-
    aggregatedListValue(Vs,Ag,AV2), aggregatedValue(V,Ag,AV2,AV).
```

Predicate `satisfies/2` verifies that each global requirement `(P,V,Op,Ag)` is satisfied by the infrastructure portion `I` determined so far. After fetching into a list `Vs` all the values of property `P` in the nodes `M` in `I`, `satisfies/2` aggregates all the values in `Vs` according to the indicated aggregator operator `Ag` and compares the obtained aggregated value `AV` with the value `P` of the considered global requirement according to the indicated comparison operator `Op`. For instance, the aggregator operators `sum` and `product` and the comparison operators `smaller` and `greater` are defined as follows:

```
aggregatedListValue([],sum,0).
aggregatedListValue([],product,1).

aggregatedValue(X,sum,Y,Z) :- Z is X+Y.
aggregatedValue(X,product,Y,Z) :- Z is X*Y.

compareValue(X,smaller,Y) :- X < Y.
compareValue(X,greater,Y) :- X > Y.
```

Predicates `satisfiesNodeReqs/2` and `satisfiesLinkReqs/2` verify that each local node and link requirement is satisfied if the new node `M` is added to the infrastructure portion `I` determined so far.

```
satisfiesNodeReqs(M,[(P,V,Op)|Rs]) :-
    nodeCap(M,P,Vm), compareValue(Vm,Op,V),
    satisfiesNodeReqs(M,Rs).
satisfiesNodeReqs(_,[]).
```

```
satisfiesLinkReqs(M,N,[(P,V,Op)|Rs]) :-
    linkCap(M,N,P,Vmn), compareValue(Vmn,Op,V),
    satisfiesLinkReqs(M,N,Rs).
satisfiesLinkReqs(_,_,[]).
```

Predicate `portionProfit/2` associates an infrastructure `Portion` with an estimate of the `Profit` that the infrastructure provider will get from leasing such `Portion`. The estimate of the overall `Profit` can be simply obtained by summing the profit associated with each node in the determined `Portion`.

```
portionProfit([N|Ns], Profit) :-
    profit(N, N_Profit),
    portionProfit(Ns, Ns_Profit),
    Profit is N_Profit + Ns_Profit.
portionProfit([], 0).
```

Finally, predicate `fogCutter/2` inputs a `RequestId` and determines the `Portion` of the available Cloud-Edge continuum that satisfies all the request's requirements and that corresponds to the highest estimated `Profit` for the infrastructure provider.

```
fogCutter(RequestId, (Portion,Profit)) :-
    setof(X, portion(RequestId, X), CandidatePortions),
    bestPortion(CandidatePortions, (Portion,Profit)).

bestPortion(CandidatePortions, (Portion,Profit)) :-
    member((Portion,Profit), CandidatePortions),
    \+ (member((_,HigherProfit), CandidatePortions), Profit < HigherProfit).
```

4 FOGCUTTER in Action

In this section, we exploit the FOGCUTTER prototype to answer questions **Q1**, **Q2** and **Q3** of Sect. 2.

We start by retaking question **Q1**:

> **Q1.** *Is there a portion of the available Cloud-Edge infrastructure that features at least 20 hardware units, and is made of at most four nodes with end-to-end connections featuring at least 10 Mbps bandwidth and at most 250 ms latency?*

The request of the application operator is denoted by a fact like:

```
request(req42, ap3, 3, [(hardware,20), (latency,250), (bandwidth,10)]).
```

The infrastructure provider can suitably query predicate `fogCutter/2`, obtaining the following as a result:

```
?- fogCutter(req42,P).
P = ([ap3, ap8, c1, c2], 20.0) ;
false.
```

Therefore, *there exist only one best candidate portion to lease to the application operator, associated to a profit of 20 €/h, and spanning node* `ap3`, `ap8`, `c1` *and* `c2`. *Note that there exist other 412 alternative solutions*[7] *that satisfy all the requirements of the application operator but feature a lower profit.*

---

[7] We determined the number of alternative solutions by querying predicate `setof(X,portion(RequestId, X),CandidatePortions)` and counting the found results.

We now consider question **Q2**:

> **Q2.** *Is there a portion of the available Cloud-Edge infrastructure that satisfies all requirements at* **Q1***, is fully located in the EU, and features antimalware and encrypted storage on all nodes?*

In this case, the request of the application operator is extended with security and location requirements, and can be expressed through the fact:

```
request(req42, ap3, 3, [(hardware,20), (latency,250), (bandwidth,10),
                        (security,[antimalware, encryptedStorage]),
                        (location,[eu])]).
```

Querying again predicate `fogCutter/2`, the infrastructure provider obtains:

```
?- fogCutter(req42,P).
P = ([ap3, ap4, c2, n2], 16.75) ;
P = ([ap3, ap7, c2, n2], 16.75) ;
false.
```

Therefore, *this request can be satisfied by two different candidate portions of the available infrastructure that bring a maximum profit of 16.75 €/h, and span nodes* `ap3`*,* `ap4`*,* `c2` *and* `n2`*, or* `ap3`*,* `ap7`*,* `c2` *and* `n2`*, respectively. Also in this case, other 35 alternative portions exist that do not maximise the profit of the infrastructure provider.*

Last, we focus on answering question **Q3**:

> **Q3.** *Is there a portion of the available Cloud-Edge infrastructure that satisfies all requirements at* **Q1** *and* **Q2** *, and features 85% availability and 30% sustainability score?*

Here, the request of the application operator becomes:

```
request(req42, ap3, 3, [(hardware,20), (latency,250),(bandwidth,10),
                        (security,[antimalware, encryptedStorage]),
                        (location,[eu]),
                        (availability, 0.85), (sustainability, 0.3)]).
```

The query to `fogCutter/2` from the infrastructure provider returns:

```
P = ([ap3, ap4, c2], 11.75).
```

Hence, *the request can be satisfied by providing the application operator with the portion made from nodes* `ap3`*,* `ap4` *and* `c2`*, with an associated profit of 11.75 €/h. For this last question, such a portion also represents the only eligible solution to the considered resource selection problem instance.*

Table 2 recapitulates on the performance of the experiments in our motivating scenario, showing the number of Prolog inferences and the CPU time needed[8] to answer the queries. It also lists the number of optimal solutions (i.e. those that maximise profit and satisfy all requirements set by the application operator) and the number of eligible solutions (i.e. those that satisfy all requirements set by the application operator). It is worth noting that FOGCUTTER determines an optimal solution almost instantaneously for all three questions.

---

[8] Prolog inferences and CPU time are obtained by relying on SWI-Prolog's `time/1` predicate. Experiments are obtained by running FOGCUTTER in SWI-Prolog v. 8.4.3 ([19]) on a machine equipped with Apple Silicon M1 and 16GB of RAM.

Question	Prolog Inferences	CPU Time	Optimal Solution(s)	Eligible Solution(s)
Q1	1,712,813	0.186 s	1	413
Q2	208,776	0.033 s	2	37
Q3	343,393	0.048 s	1	1

Table 2: Performance of FOGCUTTER over the motivating scenario.

More in general, given the maximum number `MaxNodes` of nodes requested for a portion, the worst-case time complexity for running FOGCUTTER is $O(N^{\mathtt{MaxNodes}})$. As mentioned before, determining valid placements of S services onto N nodes also features an exp-time complexity of $O(N^S)$. Setting, however, `MaxNodes` $< S$ permits to quickly identify a portion of the infrastructure where to deploy applications and can dramatically reduce decision-making times related to placements. Indeed, as shown for instance in [5] and [7], considering smaller problem instances can reduce execution times from minutes to milliseconds, with speed-ups from $2\times$ up to a $1000\times$ in realistic use cases. Also, note that portions of the Cloud-Edge continuum can be pre-computed by the infrastructure provider over pre-defined request templates, and be immediately ready to be returned to application operators, when needed.

5 Related Work

Resource allocation is an important challenge in the Cloud-edge continuum. Lahmar and Boukadi [9] conducted a comprehensive survey of resource allocation challenges that should be considered while deploying an application over a Cloud-Edge continuum. A similar survey by Shakarami et al. [16] classified existing resource allocation algorithms into five categories i.e., heuristic, framework, model, machine learning and game theory based, discussed their pros and cons and common performance metrics that are relevant for the application operators. Faticanti et al. [3] tackle a different problem which is to maximise the profit of the infrastructure provider by maximising the number of Fog applications in a batch that can be deployed over the Cloud-Edge continuum. To accomplish that their algorithm partition the application's components into two groups i.e., Cloud-only and Fog-only. Using throughput as QoS metrics, their algorithm tried to recommend appropriate regions for these components.

The role of the broker in the Cloud is also well investigated. Mei et al. [11] discuss the role and necessity of brokers in a Cloud-Edge continuum. According to them, the presence of a Cloud broker can lead to better pricing for the end user and at the same time, high utilisation of providers' resources. Cloud brokers can also earn profit in these transactions. Wang et al. [18] discussed the idea that a Cloud broker can maximise resource utilisation by differentiating between latency sensitive tasks (such as Fog applications) versus non-latency sensitive applications. They highlighted different strategies to deal with fairly pricing the two categories of applications for the broker that could make the offer attractive for a wide variety of users and providers.

The concept of IoT brokers was described in [12, 14]. Sciancalepore et al. [14] discussed a strategy that is aimed towards IoT brokers that can service different customers using IoT slices. In their approach, resources can be increased for a given slice as per demand in real-time. A subscription based model for IoT providers on the Cloud-Edge continuum was described by Niyato et al. [12]. In their scheme, IoT owners and IoT data users have to subscribe to a service that enables data exchange data based on an agreed pricing model. They also described different strategies for IoT owners to bundle their offerings and make them attractive to the Cloud-Edge continuum users.

Slicing in a network is also thoroughly investigated although it is a slightly different problem for a different setting (i.e., telecom) [20]. IoT slicing is another related topic that deals with creating slices (partitions) of IoT resources in a Cloud-Edge continuum to satisfy different types of user requirements. Fernandez et al. [4] proposed a concept of an IoT orchestration service that could satisfy a customer request, especially on a resource constraint platform using IoT slices. Similarly, Hwang et al. [8] described a central service to manage incoming requests (tasks), partition the infrastructure into different slices of the IoT sensors and distribute the tasks to different IoT slices to answer the query. Experimental results conducted by Luthra et al. [10] show that different operator placement algorithms are needed to suit the dynamic environment and conflicting QoS demands.

Our approach is different from the existing approaches as we try to solve a *resource selection* problem marginally addressed before. Our approach allows a declarative and extensible definition of parameters which can be adopted as per application operator and infrastructure operator requirements.

6 Concluding Remarks

The problem of selecting an infrastructure portion matching the requirements set by an application operator is challenging in the Cloud-Edge continuum due to the large number of assets with heterogeneous capabilities. The problem is important both for application operators, who need to suitably deploy their applications, and for infrastructure providers, who wish to maximize their profit in leasing infrastructure resources to application operators.

In this article, we have presented a declarative resource selection strategy for the Cloud-Edge continuum, implemented in the open source prototype FOGCUTTER. Using a motivating example, we have shown how FOGCUTTER can be successfully exploited to reply to different requests of customers (viz. application operators). The main advantages of the declarative solution implemented by FOGCUTTER are its readability and ease of extensibility to include new types of requirements. Indeed, it is straightforward to include further local (e.g. availability of IoT devices or specialised hardware, link jitter) and global requirements (e.g. QoS-assurance, nodes' trust) by simply programming new `capType` predicates and the associated comparison and aggregation operators. Directions for future work include:

- returning partial solutions (when not all requirements can be met), considering partial allocations of Cloud-Edge resources, and allowing customers to declare hard and soft constraints in requests,
- performing a thorough scalability assessment at varying sizes of the input Cloud-Edge infrastructure and number of considered requirements for the portion to be determined, also extending FOGCUTTER with heuristic strategies, and
- assessing the tool with real data (possibly against alternative solutions, e.g. constraint-based) in simulated, laboratory or testbed settings, also in pipeline with placement strategies to better estimate the obtainable speed-ups.

References

[1] Antonio Brogi & Stefano Forti (2017): *QoS-aware deployment of IoT applications through the fog*. *IEEE internet of Things Journal* 4(5), pp. 1185–1192, doi:10.1109/JIOT.2017.2701408.

[2] JM Chabas, Chandra Gnanasambandam, Sanchi Gupte & Mitra Mahdavian (2018): *New demand, new markets: What edge computing means for hardware companies*. Technical Report, McKinsey & Company.

[3] Francescomaria Faticanti, Francesco De Pellegrini, Domenico Siracusa, Daniele Santoro & Silvio Cretti (2020): *Throughput-aware partitioning and placement of applications in fog computing*. *IEEE Transactions on Network and Service Management* 17(4), pp. 2436–2450, doi:10.1109/TNSM.2020.3023011.

[4] Juan-Manuel Fernandez, Ivan Vidal & Francisco Valera (2019): *Enabling the orchestration of IoT slices through edge and cloud microservice platforms*. Sensors 19(13), p. 2980, doi:10.3390/s19132980.

[5] Stefano Forti, Giuseppe Bisicchia & Antonio Brogi (2022): *Declarative continuous reasoning in the cloud-IoT continuum*. Journal of Logic and Computation 32(2), pp. 206–232, doi:10.1093/logcom/exab083.

[6] Marco Gaglianese, Stefano Forti, Federica Paganelli & Antonio Brogi (2023): *Assessing and enhancing a Cloud-IoT monitoring service over federated testbeds*. Future Generation Computer Systems, doi:10.1016/j.future.2023.04.026.

[7] Juan Luis Herrera, Javier Berrocal, Stefano Forti, Antonio Brogi & Juan M Murillo (2023): *Continuous QoS-aware adaptation of Cloud-IoT application placements*. Computing, pp. 1–23, doi:10.1007/s00607-023-01153-1.

[8] Jaeyoung Hwang, Lionel Nkenyereye, Nakmyoung Sung, Jaeho Kim & Jaeseung Song (2021): *IoT service slicing and task offloading for edge computing*. IEEE Internet of Things Journal 8(14), pp. 11526–11547, doi:10.1109/JIOT.2021.3052498.

[9] Imen Ben Lahmar & Khouloud Boukadi (2020): *Resource allocation in fog computing: A systematic mapping study*. In: 2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC), IEEE, pp. 86–93, doi:10.1109/FMEC49853.2020.9144705.

[10] Manisha Luthra, Boris Koldehofe, Niels Danger, Pascal Weisenberger, Guido Salvaneschi & Ioannis Stavrakakis (2021): *TCEP: Transitions in operator placement to adapt to dynamic network environments*. Journal of Computer and System Sciences 122, pp. 94–125, doi:10.1016/j.jcss.2021.05.003.

[11] Jing Mei, Kenli Li, Zhao Tong, Qiang Li & Keqin Li (2018): *Profit maximization for cloud brokers in cloud computing*. IEEE Transactions on Parallel and Distributed Systems 30(1), pp. 190–203, doi:10.1109/TPDS.2018.2851246.

[12] Dusit Niyato, Dinh Thai Hoang, Nguyen Cong Luong, Ping Wang, Dong In Kim & Zhu Han (2016): *Smart data pricing models for the internet of things: a bundling strategy approach*. IEEE Network 30(2), pp. 18–25, doi:10.1109/MNET.2016.7437020.

[13] Farah Ait Salaht, Frédéric Desprez & Adrien Lebre (2020): *An overview of service placement problem in fog and edge computing*. ACM Computing Surveys (CSUR) 53(3), pp. 1–35, doi:10.1145/3391196.

[14] Vincenzo Sciancalepore, Flavio Cirillo & Xavier Costa-Perez (2017): *Slice as a service (SlaaS) optimal IoT slice resources orchestration*. In: GLOBECOM 2017-2017 IEEE Global Communications Conference, IEEE, pp. 1–7, doi:10.1109/GLOCOM.2017.8254529.

[15] Suman Sehra & Varsha Ramamurthy (2022): *Paving the Way Forward Intelligent Road Infrastructure*. Technical Report, Intel. Available at `https://www.intel.com/content/www/us/en/transportation/resources/road-infrastructure-ebook.html`.

[16] Ali Shakarami, Hamid Shakarami, Mostafa Ghobaei-Arani, Elaheh Nikougoftar & Mohammad Faraji-Mehmandar (2022): *Resource provisioning in edge/fog computing: A comprehensive and systematic review*. Journal of Systems Architecture 122, p. 102362, doi:10.1016/j.sysarc.2021.102362.

[17] Sven Smolka & Zoltán Ádám Mann (2022): *Evaluation of fog application placement algorithms: a survey*. Computing 104(6), pp. 1397–1423, doi:10.1007/s00607-021-01031-8.

[18] Xinhou Wang, Song Wu, Kezhi Wang, Sheng Di, Hai Jin, Kun Yang & Shumao Ou (2017): *Maximizing the profit of cloud broker with priority aware pricing*. In: 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS), IEEE, pp. 511–518, doi:10.1109/ICPADS.2017.00073.

[19] Jan Wielemaker, Tom Schrijvers, Markus Triska & Torbjörn Lager (2012): *SWI-Prolog*. Theory and Practice of Logic Programming 12(1-2), pp. 67–96, doi:10.1017/S1471068411000494.

[20] Shalitha Wijethilaka & Madhusanka Liyanage (2021): *Survey on network slicing for Internet of Things realization in 5G networks*. IEEE Communications Surveys & Tutorials 23(2), pp. 957–994, doi:10.1109/COMST.2021.3067807.

Assessing Drivers' Situation Awareness in Semi-Autonomous Vehicles

ASP based Characterisations of Driving Dynamics for Modelling Scene Interpretation and Projection

Jakob Suchan Jan-Patrick Osterloh
German Aerospace Center (DLR)
Institute for Systems Engineering for Future Mobility, Oldenburg, Germany
jakob.suchan@dlr.de, jan-patrick.osterloh@dlr.de

Semi-autonomous driving, as it is already available today and will eventually become even more accessible, implies the need for driver and automation system to reliably work together in order to ensure safe driving. A particular challenge in this endeavour are situations in which the vehicle's automation is no longer able to drive and is thus requesting the human to take over. In these situations the driver has to quickly build awareness for the traffic situation to be able to take over control and safely drive the car. Within this context we present a software and hardware framework to asses how aware the driver is about the situation and to provide human-centred assistance to help in building situation awareness. The framework is developed as a modular system within the Robot Operating System (ROS) with modules for sensing the environment and the driver state, modelling the driver's situation awareness, and for guiding the driver's attention using specialized Human Machine Interfaces (HMIs).

A particular focus of this paper is on an Answer Set Programming (ASP) based approach for modelling and reasoning about the driver's interpretation and projection of the scene. This is based on scene data, as well as eye-tracking data reflecting the scene elements observed by the driver. We present the overall application and discuss the role of semantic reasoning and modelling cognitive functions based on logic programming in such applications. Furthermore we present the ASP approach for interpretation and projection of the driver's situation awareness and its integration within the overall system in the context of a real-world use-case in simulated as well as in real driving.

1 Introduction

With the rise of automated and semi-automated driving, a range of new challenges have moved into sight of developers. In particular semi-autonomous vehicles pose important questions when it comes to the interplay between driver and vehicle, since for these systems it is critical that the cooperation of the human and the automation system is seamless. To ensure this, the automation system needs to be designed with the needs of the human in mind and ideally have the functionality to asses the mental state of the driver and the driver's interaction with the system. Such functionality is especially important in situations when the driver and the automation have to function together, in order to safely operate the vehicle. A particular challenge in this context, and of importance with respect to this paper, is the question of how to enable the driver to quickly take over control from the automation system in situations, in which the automation cannot ensure safe driving, e.g., in unforeseen situations, or in situations outside of the Operational Design Domain (ODD) of the automation.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 300–313, doi:10.4204/EPTCS.385.29

The SituWare project presented in this application paper aims at developing a system to provide the driver with assistance in such situations, by modelling the driver's awareness of the situation and guide their attention to critical elements of the situation possibly missed by the driver. The focus of this paper is on highlighting the application of declarative logic programming methods as a means to model the driver's mental processing of the situation and demonstrate its integration within a large-scale modular assistive system. The work presented in this paper is driven by considerations in the fields of human-centred design, cognition, and formal semantic reasoning.

Human-centred design for (semi-)autonomous vehicles. With the above challenges in mind human factors play a central role in the development of novel mobility systems and autonomous vehicles [12] and development of novel systems is often accompanied and driven by empirical research investigating how humans interact with these systems, either for external communication with vulnerable road users, e.g. [18, 26], and for internal communication e.g. in handover scenarios [7, 8, 16] or for acceptance and user experience [20].

Cognitive abilities for driving assistance systems. Research in the area of cognitive systems and cognitive modelling is concerned with developing methods and tools that reflect cognitive abilities. Most directly connected to the topic of this paper and the development of SituSYS is the research on Situation Awareness and its modelling in computational systems [3]. In the context of driving assistance systems research on Situation Awareness has mostly been concerned with modelling e.g. by [11, 17, 19].

Semantic reasoning about dynamic situations. A key factor in developing such capabilities is the ability to abstractly represent the driving environment and the traffic dynamics within it, to interpret and reason about them on a semantic level. In this context logic programming and Answer Set Programming (ASP) [4, 5, 6] in particular has evolved as a powerful tool for semantic reasoning about dynamic scenes. For instance, ASP has been used in the area of stream reasoning [2] as a general tool for reasoning about large scale dynamic data, e.g, in the driving domain [13], for decision-making [9], or for recognition and reasoning about events [24]. Furthermore, [23] has developed a general method for visual abduction based on ASP, and applied it in the area of online semantic sense-making with commonsense knowledge in the context of safety-critical situations in driving [21, 22].

Within this paper we build on these works to develop an ASP-based approach for assessing the driver's situation awareness by characterising driving dynamics for modelling and reasoning about the driver's interpretation and projection of the scene dynamics.

2 SituWare: An Online System for Assessing Situational Awareness

The general focus of the SituWare project is f build and evaluate the technological basis for developing a usable software and hardware system (*SituSYS*), targeted at the reliable detection, interpretation, and consideration of the driver's situational awareness. In this section we discuss the relevance of situation awareness in (semi-)autonomous driving and provide an overview of the full system and its components.

2.1 Situation Awareness in (Semi-)Autonomous Driving

Today, in the automotive domain and within the framework of the SAE J3016 automation level system, it is generally assumed that the driver observes the environment and is thus immediately available as a fallback level (SAE level 2 - partial automation), can react to a request to intervene (SAE level 3 -

conditional automation) or can take over the vehicle control outside the operational area of the automation (SAE level 4 - high automation). Previous studies on handover scenarios, especially in the context of conditional automation, show that a considerable amount of time of at least 7 to 10 seconds must be reserved for the driver to safely take over the driving task. This time is needed by the driver(s) to gain an accurate situation awareness so that a safe handover is accomplished. Current approaches for handover scenarios from the system to the driver consider the current driver's state only peripherally and very roughly. However, the potential of such recognition and interpretation is very large. Adaptive interaction sequences can be used to optimally return the driver to the driving task. In addition, a more detailed picture of the driver's state also allows for the adaptation of the driving behavior of the automation system, so that larger time reserves can be kept available for a highly distracted driver, while time reserves can be reduced for an attentive driver without reducing safety. One of the most prominent models for Situation Awareness is the one developed by Mica Endsley [3]. The idea behind this model is that Situation Awareness is built in three Stages, namely the *Perception*, *Interpretation*, and *Projection* stage, also known as Level 1 to Level 3 Situation Awareness (**L1** - **L3**):

L1. At Level 1, the data and elements of the environment that build the current situation are perceived. The main factor that influences Level 1 Situation Awareness is the focus of attention that is apparent in the situation, which in turn is guided by the goals and objectives of the user, as well as their expectations. Previous Experience and Training can influence the attention process and thus the ability of the user to perceive the environment correctly. Drivers need to perceive the cars around their own car, signs along the road, the road curvature and condition, and lots of other things.

L2. Level 2 Situation Awareness builds upon Level 1 through interpretation and comprehension of the perceived elements. At Level 2, a holistic picture of the environment including the significance of objects and events is formed. This interpretation is influenced by the goals of the user, previous experience as well as the workload and stress. In the automotive example, drivers need to understand that proximity of other cars might indicate a risk, that there are speed limits to keep, or the importance of the other signs on the road.

L3. At Level 3, the user builds a projection of the current situation into the future, based on the comprehension built at Level 2. Level 3 Situation Awareness is mainly influenced by the previous experience and training of the user, since this builds his knowledge on how the entities in the environment may act in the near future. For drivers, this could mean to predict the future speed and positions of other cars as well as possible actions the other drivers might take (like changing lanes, or breaking) as well as upcoming speed limits and the risk associated with the road conditions ahead.

2.2 A Modular System implemented in ROS

Objective of the SituWare project is to build *SituSYS*, a system that predicts the situation awareness of a driver, calculates possible deviations from an optimal situation awareness and finally uses specialized interaction techniques to improve the driver's situation awareness. Figure 1 shows a conceptual overview of the architecture of *SituSYS*: *SituSYS* consists of three parts, the sensor layer *SituSENSORS* which measures the driver and environment, the model layer *SituMODEL* which calculates the situation awareness, and *SituHMI* for the interaction. Starting at the sensor-layer *SituSENSORS*, multiple *Vehicle Sensors* are used to sense the environment and vehicle state, i.e. surrounding objects, signs, current vehicle speed and automation state. In addition to the vehicle sensors, an eye-tracker is used to detect the gaze of the driver. The calculated gaze vector is then used in the *Perception*-Model within *SituMODEL* to predict which elements in the environment and in the car has been looked at, i.e. other cars, street signs, or

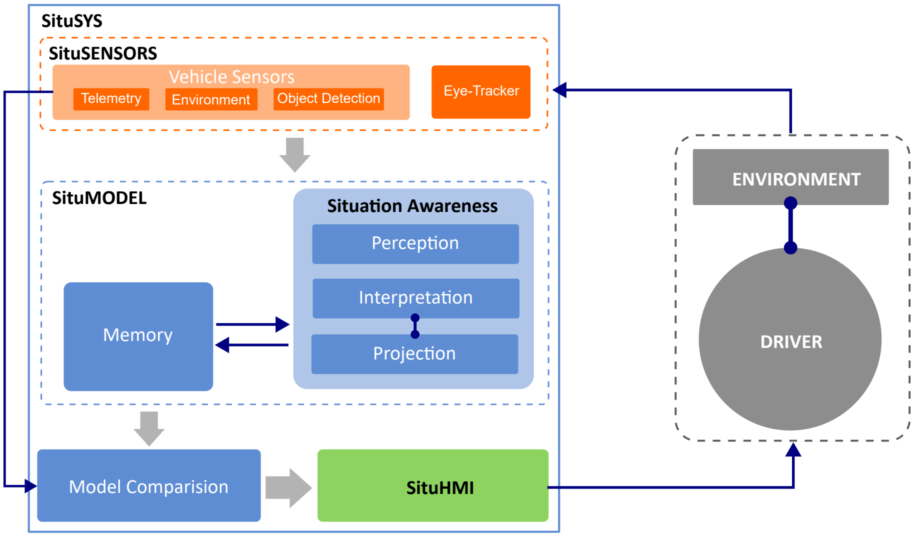


Figure 1: **SituSYS**: Conceptual Overview

any cockpit elements. Based on this information, the objects that have been perceived are then written into the *Memory*-Model. The *Memory*-Model implements retrieval and forgetting processes, and can be accessed by the other *SituMODEL* components. The *Interpretation*- and *Projection*-Models are for the calculation of the situation interpretation and the projection of the future state (A detailed description can be found in Section 3.2). The outputs of the *Perception*-, *Interpretation*-, and *Projection*-Models are then used by the *Model Comparison* to calculate the overall situation awareness (Level 1, Level 2 and Level 3 in Endsley's Model), by comparing the content in the memory, that has been produced by the components of the *SituMODEL*, with the data gathered by the vehicle sensors. Although the vehicle sensors have an error probability associated with them, we assume in this case that this is the *ground truth* for *SituSYS*. The *Model Comparison* then weights the differences between the memory and the ground truth and calculates a list of diverging elements sorted by priority. This list is then used by *SituHMI* to direct the focus of attention to the most important element. In SituWare we tested different interaction methods for that purpose [1].

To organize and coordinate the implementation of the different components of *SituSYS*, the *ROS* framework has been used. Each component of *SituSYS* has been implemented as separate node, allowing a modularization of the system. *ROS messages* describing the data exchanged between the components are used to provide a standardised interface between the different modules. The use of the ROS framework also facilitates the use of different sources for the input, i.e. we connected *SituSYS* to two different driving simulators as well as to a real car, by implementing a dedicated node that collects the data from the simulator or car and then sends them in the defined messages to *SituSYS*, and receives the messages from *SituHMI* to implement the selected interaction method in the vehicle/simulator.

2.3 Technical Setup and Data

The SituWare project integrates SituSYS within two different simulators and one real semi-autonomous vehicle and provides the hardware basis for sensing the ego vehicle and the environment together with mobile tracking of the driver's gaze. Figure 2 shows the simulated scenario within one of the two simulators used within the SituWare project.

a) Simulated scene of construction site after takeover

b) Driving Simulator

Figure 2: **Simulation Environment:** The simulated scene and the simulator setup.

▶ **Scene Data** The vehicle sensors implemented in the specific platform are used to sense the state of the ego vehicle and to sense the environment including the vehicles within it. This information is published as *ROS messages* and used by the modules of *SituSYS*. The messages contain data on the ego vehicle, the automation system, and other vehicles in the scene as listed in table 1.

▶ **Eye-Tracking Data** The driver's perception is sensed using mobile eye-tracking that provides gaze coordinates and fixations within an ego view of the scene, as captured by the camera of the eye tracker. These coordinates are translated to the 3-dimensional space of the scene, which allows mapping of gaze data to scene elements. Within *SituSYS* the gaze data is provided as a 3D vector. The modular design of the system abstracts from the actual eye-tracking system, however, in the case study presented in this paper the PupilLabs mobile eye-tracker is used, which is equipped with a eye camera capturing the gaze with up to $120Hz$ and providing an accuracy of 0.60°.

3 Assessing the Driver's Situation Awareness

Within *SituSYS* situation awareness is modelled by the perception module and the interpretation & projection module. The perception module implements 1st level Situation Awareness using scene data from the *SituSENSORS* together with eye-tracking data to calculate fixation probabilities for each scene object.

Perception

Level 1 Situation Awareness – *perception of the scene*[1] – is estimated based on the gaze data of the driver which is recorded by *SituSENSORS*. From this information a probability is computed for each scene object giving the likelihood that the driver has fixated the object together with a fixation time providing the duration of the fixation on the object. These measures are then published within the *ROS message* as an input for the interpretation & projection module, as well as the model comparison.

2nd and 3rd level Situation Awareness are implemented together in the interpretation & projection module. These consist of:

- Level 2 Situation Awareness – *interpretation of the scene* – models the awareness of the driver regarding scene elements in the current situation;

---

[1] Technical details on the used method for calculating the probabilities are out of the scope of this paper. For the examples of this paper we consider the perception module to be a black-box system providing the information if an object was perceived by the driver.

Attribute	Type	Description
	Ego Vehicle	
ID	*string*	The *ID* of the ego vehicle.
type	*string*	The *type* of the ego vehicle.
position	*{float, float, float}*	The *position* of the ego vehicle as 3D vector.
orientation	*{float, float, float}*	The *orientation* of the ego vehicle as 3D vector.
velocities	*{float, float, float}*	The *velocity* of the ego vehicle as 3D vector.
indicator_left	*bool*	Truth value whether the *left indicator* is active.
indicator_right	*bool*	Truth value whether the *right indicator* is active.
acceleration	*float*	The *acceleration* of the ego vehicle.
current_speed_limit	*int*	The *current speed limit* holding for the ego vehicle.
current_lane	*float*	The *current lane* the ego vehicle is on.
	Ego Automation	
takeover_request	*bool*	Truth value whether the *takeover request* is active.
time_until_odd_boundary	*float*	*Time* until the driver has to take over.
criticality_level	*int*	*Criticality level* of the takeover request.
takeover_reason	*string*	*Reason* for the takeover request.
ego_automation_state	*bool*	Truth value whether the *automation* is active.
	Other Vehicles	
ID	*string*	The *ID* of the traffic vehicle.
type	*string*	The *type* of the traffic vehicle.
position	*{float, float, float}*	The *position* of the traffic vehicle as 3D vector.
orientation	*{float, float, float}*	The *orientation* of the traffic vehicle as 3D vector.
velocities	*{float, float, float}*	The *velocity* of the traffic vehicle as 3D vector.
acceleration	*{float, float, float}*	The *acceleration* of the traffic vehicle as 3D vector.
dimension	*{float, float, float}*	The *dimension* of the traffic vehicle as 3D vector.
lane	*int*	The *lane* the traffic vehicle is on.
fixation_probability	*float*	The *probability* that the driver fixated the vehicle.
fixation_time	*int*	The *duration* of the driver's fixation on the vehicle.

Table 1: **Scene Data:** Relevant data-points from *SituSYS*.

- Level 3 Situation Awareness – *projection of scene dynamics* – models the expectations of the driver how the scene will evolve, i.e., the scene dynamics with respect to the task of the driver.

The implementation of these levels is based on semantic characterisations of the dynamics of the driving domain, which are declaratively defined within answer set programming (ASP) [4, 5, 6]. In particular, the interpretation & projection module of *SituSYS* consist of an online Python process maintaining a representation of the scene and uses an integrated ASP solver to generate interpretation and projection models based on the scene data from the *SituSENSORS* and the perception data from the perception module. In the following we provide the formal characterisation of the driving domain, and describe the interpretation and projection process in detail.

3.1 The Driving Domain

The domain is characterised by $\Sigma{<}\mathcal{O}, \mathcal{E}, \mathcal{R}, \mathcal{T}, \Phi, \Theta{>}$, which is used to formalise the driver's representation of the scene dynamics. In particular, the driver's belief state is represented on the one hand by the static and dynamic properties of the scene and the elements within it given by the domain objects, the basic entities representing these objects, and the spatial and temporal aspects of the scene $< \mathcal{O}, \mathcal{E}, \mathcal{R}, \mathcal{T} >$. On the other hand it is represented by the high-level event dynamics of perceived events and possible future events given by $< \Phi, \Theta >$.

Events	**Description**
$change\_lane(Entity, Lane_1, Lane_2, Location)$	An Entity changing the lane from $Lane_1$ to $Lane_2$ to a specific $Location$.
audio_signal_start	The audio signal indicating that the driver has to take over started.
audio_signal_end	The audio signal indicating that the driver has to take over ended.
takeover_manual	The driver takes over control of the vehicle.
takeover_automation	The automation takes over control of the vehicle.

Fluents	**Description**
curr_task(Task)	The current task, the driver has to perform.
automation	If the driving automation of the vehicle is on or not.
audio_signal	If the audio signal indicating that the driver has to take over control of the vehicle is on or not.
on_lane(Entity, Lane)	The lane a scene entity is driving on.

Table 2: **Event Dynamics:** Exemplary events and fluents applicable in take-over situations of the use-case described in section 3.2.

▶ **Domain Objects ($\mathcal{O}$) and Spatial Entities ($\mathcal{E}$).** The scene consists of different scene elements, in particular we are considering the ego vehicle and other vehicles in the scene, as well as the lanes on the road and gaps between vehicles.

Vehicles. We distinguish between the ego vehicle and other vehicles in the scene, constituting the traffic. We use the following objects for representing vehicles:

- The *ego vehicle*: $\mathcal{O}_{ego} = ego$;
- other vehicles in the traffic: $\mathcal{O}_{trf} = \{trf_1, ..., trf_n\}$.

These elements are geometrically represented as spatial entities $\mathcal{E} = \{\varepsilon_1, ..., \varepsilon_n\}$ within the 3-dimensional scene space. Additionally they are assigned dynamic and static attributes as obtained from the egos driving system (for the attributes of the ego vehicle) and the ego vehicle sensors (for estimated attributes of other vehicles) as detailed in section 2.3.

Lanes. These are based on the OpenDRIVE standard [25] , in which lanes are numbered with positive and negative numbers and 0 represents the middle of the road. Lanes are adjacent when the lane *ids* are consecutive.

- Lanes: $\mathcal{O}_{lanes} = \{lane_1, ..., lane_n\} \cup \{lane_{-1}, ..., lane_{-n}\}$.

We define the adjacency of lanes $lane_i, lane_j \in \mathcal{O}_{lanes}$ on the road using the predicate $adjacent(lane_i, lane_j)$.

Gaps. Of particular interest for the task at hand are gaps between vehicles in the scene. Therefore we introduce objects representing these gaps.

- Gaps: $\mathcal{O}_{gaps} = \{gap_{trf_i,trf_j}, ..., gap_{trf_p,trf_q}\}$.

Gaps are declaratively defined based on the vehicles in the scene using the predicate $gap(trf_i, trf_j)$, where $trf_i, trf_j \in \mathcal{O}_{trf}$. Additionally we define the predicate $gap\_size(gap_{trf_i,trf_j}, size)$, where $gap_{trf_i,trf_j} \in \mathcal{O}_{gaps}$ and *size* is a number representing the distance between trf_i and trf_j.

▶ **Spatial Configuration ($\mathcal{R}$).** The spatial arrangement of the entities in the scene is represented from an egocentric perspective in the context of the road, i.e., representing whether vehicles are ahead or behind the ego vehicle, on the same lane, or on the lane to the left or right of the ego vehicle, and how many other vehicles are between a vehicle and the ego vehicle. To this end we define the following relations in $\mathcal{R}$ holding between the ego vehicle and the other scene elements.

- *Relative longitudinal direction*: $R_{rel\_long} = \{ahead, behind, overlapping\}$, representing the direction of an object on the longitudinal axis based on the road layout.
- *Relative lane*: $R_{rel\_lane} = \{same, left, right\}$, representing the lane an object is on, relative to the lane the ego vehicle is on.
- *Relative ordering*: $R_{rel\_order} = n+1$, where n is the number of other vehicles between the vehicle and the ego vehicle, representing the position of an object with respect to the ego vehicle and the other vehicles on the road.

▶ **Time ($\mathcal{T}$).** We represent time using time points $\mathcal{T} = \{t_1, ..., t_n\}$. These are used to denote that dynamic object properties and spatial relations between basic entities representing scene objects hold at a certain time, as well as to describe temporal aspects of event dynamics.

▶ **Driving Event Dynamics ($<\Phi$, $\Theta>$).** We use the event calculus notation to define the event dynamics in the driving domain. Towards this we define **fluents** $\Phi = \{\phi_1, ..., \phi_n\}$ and **events** $\Theta = \{\theta_1, ..., \theta_n\}$ to characterise dynamic properties of the scene objects and high-level events (e.g., Table 2). We use the axioms $occurs\_at(\theta, t)$ denoting that an event occurred at time t and $holds\_at(\phi, v, t)$ denoting that v holds for a fluent ϕ at time t.

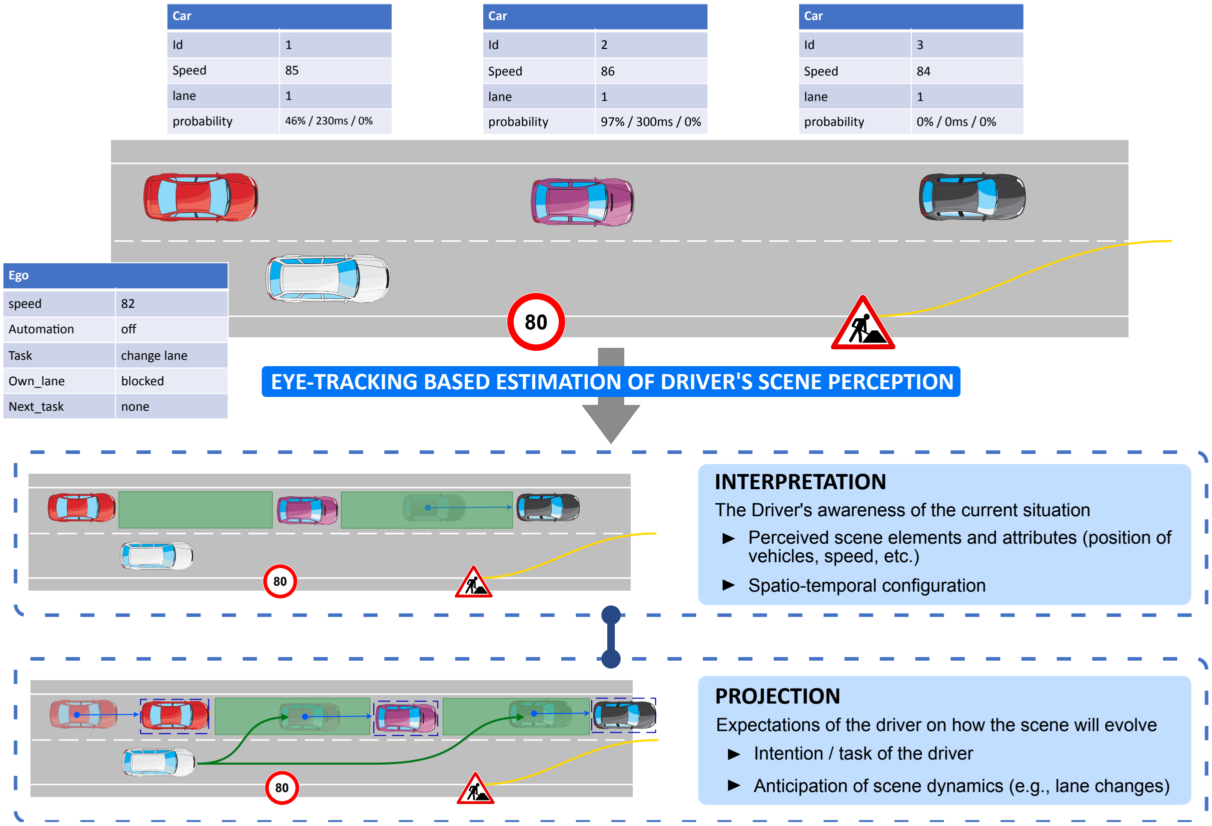
3.2 Reasoning about Situation Awareness Level 2 & 3: Interpretation and Projection

The interpretation and projection module is based on declarative characterisations of driving dynamics (as defined in section 3.1) implemented in ASP, using Event Calculus [10] for reasoning about events in the scene. In particular, we are building on the Event Calculus (EC) as formalised in [14, 15]. The module is implemented as a hybrid system, in which an online Python process maintains a representation of the mental belief state of the driver and uses an integrated ASP solver for generating the interpretation and the projection model. For this, the module uses the scene data $\mathcal{S}$ containing the ego vehicle data and the environmental data including the gaze data of the driver, together with the characterisations of the driving domain in Σ.

Application: The Case of Take-Over Situations in a Construction Environment

As a use case we have applied *SituSYS* in the context of a case study conducted in the driving simulator. The case study implemented a situation in which a driver had to take over the control of a highly automated vehicle at a section of a highway, where the current lane ended because of a construction site and the vehicle had to change lanes. The driver got notified about the upcoming take-over and had a certain time window to get familiar with the situation and take over control to manually perform the lane change.

As a test case the overall system is applied in the context of the above case study, in which the driver has to take over control from the automation to safely drive the vehicle. Fig. 3 shows an exemplary situation in the context of the case study and depicts the interpretation and the projection step.



Car	
Id	1
Speed	85
lane	1
probability	46% / 230ms / 0%

Car	
Id	2
Speed	86
lane	1
probability	97% / 300ms / 0%

Car	
Id	3
Speed	84
lane	1
probability	0% / 0ms / 0%

Ego	
speed	82
Automation	off
Task	change lane
Own_lane	blocked
Next_task	none

INTERPRETATION

The Driver's awareness of the current situation

- Perceived scene elements and attributes (position of vehicles, speed, etc.)
- Spatio-temporal configuration

PROJECTION

Expectations of the driver on how the scene will evolve

- Intention / task of the driver
- Anticipation of scene dynamics (e.g., lane changes)

Figure 3: **Application Example:** Interpretation & Projection

Computational Steps for Situation Awareness Level 2 & 3. The overall process for interpretation and projection consists of the following steps (S1-S3, also refer to Alg. 1) performed at each time point:

S1. Update Scene Elements and Predict Current State. Update the driver's mental belief state ($\mathcal{MBS}$) with the scene elements for which the fixation probability is above the fixation threshold, and predict the current position of all scene elements that are part of the driver's mental belief state, using a Kalman Filter base motion model, assuming constant velocity.

S2. Generate the Problem Specification. Generate an ASP problem statement, containing the scene elements that are part of the driver's mental belief state, and the characterisations of the driving domain in Σ.

S3. Integrated ASP Solving. Generate the interpretation model ($\mathcal{IM}$) and the projection model ($\mathcal{PM}$) by solving the generated problem specification using the Clingo solver integrated within the Python process.

Within this process the interpretation and the projection are implemented as follows:

▶ **Interpretation.** The interpretation level of situation awareness is modelled as an extrapolated representation of the scene, consisting of the perceived scene elements, together with the arrangement of these elements and the events happening in the scene. When the probability that the driver has fixated a particular object exceeds a certain threshold we register it within the scene representation and use a Kalman Filter based motion model assuming constant velocity to maintain a representation of the object

```
Algorithm 1: Interpret_and_Project($\mathcal{S}$, $\Sigma$)
Data: Scene data ($\mathcal{S}$), and the characterisation of the driving domain $\Sigma$
Result: Interpretation Model ($\mathcal{IM}$), Projection Model ($\mathcal{PM}$)
$\mathcal{MBS} \leftarrow \varnothing$
for $t \in T$ do
    for $object \in \mathcal{MBS}$ do
        $position_{object} \leftarrow kalman\_predict(object)$
    for $object \in \mathcal{S} \wedge not\ object \in \mathcal{MBS}$ do
        if $fixation\_probability_{object} > fixation\_threshold$ then
            $\mathcal{MBS} \leftarrow \mathcal{MBS} \cup object$
    for $object \in \mathcal{S} \wedge object \in \mathcal{MBS}$ do
        if $fixation\_probability_{object} > fixation\_threshold$ then
            $position_{object} \leftarrow kalman\_update(object)$
    $< \mathcal{IM}, \mathcal{PM} > \leftarrow ASP\_solve(\mathcal{MBS}, \Sigma)$
return $< \mathcal{IM}, \mathcal{PM} >$
```

while the driver is not fixating on it. When the driver is fixating the object again the estimated position is updated with the sensed one. In this way we estimate the driver's mental belief state about the movement of scene elements.

This estimated mental representation of the scene is then used together with the characterisations of the driving domain in Σ to generate the interpretation model ($\mathcal{IM}$). To this end we declaratively model scene artefacts, spatial configuration, and events.

Scene Artefacts. These are elements of the scene that are indirectly obtained from the sensed objects. For instance, gaps between vehicles the driver is aware of, are declaratively defined using the following rule, stating that there is a gap between two entities if they are on the same lane and there is no other entity between these two entities.

```
gap(entity(ID1), entity(ID2)) :-
    entity(ID1), entity(ID2), ID1 != ID2,
    same_lane(entity(ID1), entity(ID2)),
    in_front_of(entity(ID1), entity(ID2)),
    not between(_, entity(ID1), entity(ID2)).
```

Spatial Configuration. The relational spatial structure holding between the scene elements the driver is aware of, is represented using the spatial relations defined in $\mathcal{R}$.

Events. We use the event calculus to detect driving events based on the driver's mental belief state of the scene. Towards this we define fluents representing dynamic scene properties, e.g, the fluent $on\_lane(entity(ID), lane(Lane))$ denotes that an entity is on a particular lane, or the fluent $curr\_task(Task)$ denotes the current task of the driver.

```
fluent(on_lane(entity(ID), lane(Lane))) :- entity(lane(Lane)), entity(ID).
fluent(curr_task(Task)) :- task(Task).
```

Additionally, we define events changing these scene properties. For instance the following definition of the event $takeover\_manual$ states that the event occurs if the fluent $automation$ is $true$ and the sensed state of the automation is $false$. Further, it states that the event initiates the fluent $curr\_task(change_{l}ane)$, and terminates the fluent $automation$ and $curr\_task(build\_sit\_aware)$.

```
event(takeover_manual).
initiates(takeover_manual, curr_task(change_lane), T) :- timepoint(T).
terminates(takeover_manual, automation, T) :- timepoint(T).
terminates(takeover_manual, curr_task(build_sit_aware), T) :- timepoint(T).
```

```
occurs_at((takeover_manual, T) :- holds_at(automation, T), ego_automation_state(false), timepoint(T).
```

Generating the Driver's Scene Interpretation. Solving this Answer Set Program with the scene elements the driver is aware of results in a model of the scene based on the drivers subjective perception, which constitutes the interpretation model ($\mathcal{IM}$).

In particular this model includes the following elements:

Driving events the driver is aware of.

```
occurs_at(audio_signal_start,119208). ... occurs_at(takeover_manual,135258).
```

Fluents representing the driver's belief state about the properties of the scene.

```
holds_at(curr_task(lane_change),136275). ...
holds_at(on_lane(entity(trf(1)),lane(1)),136275). holds_at(on_lane(entity(trf(2)),lane(1)),136275). ...
```

Gaps between the vehicles the driver is aware of.

```
gap(entity(trf(1)),entity(trf(2))). gap(entity(trf(2)),entity(trf(3))). ...
```

The qualitative spatial configuration of the vehicles the driver is aware of.

```
rel_longitudal_direction(ahead, entity(3)). rel_lane(left, entity(3)). rel_order(0, entity(3)). ...
```

This generated interpretation model ($\mathcal{IM}$) serves as input to the model comparison module to compare the drivers interpretation of the scene to the scene sensed by the SituSENSORS. Additionally these serve as abstractions needed for the projection step.

▶ **Projection.** Within SituWare the projection step is used to explore possible future states by generating the set of events, which are possible in the current situation, and which are consistent with the task of the driver. For instance in our use-case the driver has to perform a lane change after taking over control from the automation system. To model this we define the lane change event within our domain characterisation and use the event calculus to generate possible lane change events the driver could perform for each time point. This set of possible events constitute the projection model ($\mathcal{PM}$) and can be used to identify scene elements the driver needs to be aware of in order to safely perform the given task.

Event Anticipation. To anticipate scene events we define the events relevant for the driving task within the event calculus. For instance, the lane change event is defined based on the changes the event causes to the fluents and the constraints defining in which situations the event is possible.

```
event(change_lane(entity(ego(ID)), lane(Lane1), lane(Lane2), Location)) :-
    entity(ego(ID)), entity(lane(Lane1)), entity(lane(Lane2)),
    gap(entity(ID1), entity(ID2)).
```

```
initiates(change_lane(entity(ego(ID)), lane(Lane1), lane(Lane2), Location),
        on_lane(entity(ego(ID)), lane(Lane2)), T) :-
                entity(ego(ID)), entity(lane(Lane1)), entity(lane(Lane2)), timepoint(T).
terminates(change_lane(entity(ego(ID)), lane(Lane1), lane(Lane2), Location),
        on_lane(entity(ego(ID)), lane(Lane2)), T) :-
                entity(ego(ID)), entity(lane(Lane1)), entity(lane(Lane2)), timepoint(T).
```

```
poss(change_lane(entity(ego(ID)), lane(Curr_Lane), lane(Lane), Location)) :-
    holds_at(on_lane(entity(ego(ID)), lane(Curr_Lane)), T), curr_time(T),
    adjacent(lane(Lane), lane(Curr_Lane)), free(Location),
    on_lane(gap(entity(ID1), entity(ID2)), lane(Lane)).
```

In particular, a lane change is possible, when the lanes are adjacent to each other and there is a free location on the target lane. This free location may be either a gap between two vehicles, a free space behind or in front of a vehicle, or a completely empty lane. For the projection of the scene we generate all events that initiate the goal of a particular task and that are possible in the current situation.

```
anticipate(Event) :-
    initiates(Event, Goal, T), holds_at(curr_task(Task), T), goal(curr_task(Task), Goal), curr_time(T).
```

```
:- anticipate(Event), not poss(Event).
```

For the example scene this results in two possible lane changes.

```
anticipate(change_lane(ego(1),lane(2),lane(1),gap(entity(trf(1)),entity(trf(2))))).
anticipate(change_lane(ego(1),lane(2),lane(1),gap(entity(trf(2)),entity(trf(3))))).
```

4 Results and Future Directions

We presented an online and real-time capable approach for modelling drivers' situation awareness in the context of takeover situations as they happen in semi-autonomous driving. The approach is based on declarative characterisations of driving dynamics and is implemented as a ROS module within Python and Answer Set Programming (ASP) as a part of *SituSYS*, a modular system for measuring the drivers' situation awareness and guiding their attention based on the results. We highlighted the application of the presented approach as part of the *SituSYS* framework, which has been integrated in simulated as well as real-world driving. And we presented a case study concerned with situation awareness in take-over situations.

▶ **Application Results.** Application of the interpretation and projection model in the context of the case study has shown that the model is capable of maintaining a representation of the scene and provide the necessary information for comparing the estimated scene representation of the driver with the sensed information, and to provide individual guidance using the specialized attention guidance devices implemented in the *SituHMI*. The module works in real-time (approx. 30 Hz) with up to 20 vehicles plus the ego vehicle in the scene. However, to fully assess the effectiveness of the system it is required to perform a long term empirical study with the complete integrated system and measure the effect of *SituSYS* on the driving performance in take-over situations. To conduct such a study is not in the scope of this paper and subject of ongoing and future research.

▶ **Future Directions.** The presented *SituSYS* framework constitutes an ideal basis for applying logic based reasoning within simulated and real-world driving tasks. In this context future developments are aiming for more elaborate approaches for projecting scene dynamics, including the generation of possible and expected trajectories. Aside from this, an ongoing effort is the long term empirical assessment of the performance of *SituSYS* with respect to the cognitive adequacy of the generated mental models within simulated as well as real-world driving.

Acknowledgments

We acknowledge partial funding by the *Federal Ministry for Economic Affairs and Climate Action* (BMWK) as part of the *SituWare* project (reference no. 19A19011C and 19A19011F).

References

[1] Mark Colley, Lukas Gruler, Marcel Woide & Enrico Rukzio (2021): *Investigating the Design of Information Presentation in Take-Over Requests in Automated Vehicles*. In: *Proceedings of the 23rd International Conference on Mobile Human-Computer Interaction*, MobileHCI '21, Association for Computing Machinery, New York, NY, USA, doi:10.1145/3447526.3472025.

[2] Thomas Eiter & Rafael Kiesel (2020): *Weighted LARS for Quantitative Stream Reasoning*. In *ECAI 2020 - 24th European Conference on Artificial Intelligence, Santiago de Compostela, Spain, Frontiers in Artificial Intelligence and Applications* 325, IOS Press, doi:10.3233/FAIA200160.

[3] Mica R. Endsley (1995): *Toward a Theory of Situation Awareness in Dynamic Systems*. *Human Factors* 37(1), pp. 32–64, doi:10.1518/001872095779049543.

[4] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub & Philipp Wanko (2016): *Theory Solving Made Easy with Clingo 5*. In Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016), *OpenAccess Series in Informatics (OASIcs)* 52, SchlossDagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 2:1–2:15, doi:10.4230/OASIcs.ICLP.2016.2.

[5] Martin Gebser, Roland Kaminski, Benjamin Kaufmann & Torsten Schaub (2012): *Answer Set Solving in Practice*. Morgan & Claypool Publishers, doi:10.1007/978-3-031-01561-8.

[6] Martin Gebser, Roland Kaminski, Benjamin Kaufmann & Torsten Schaub (2014): *Clingo = ASP + Control: Preliminary Report*. *CoRR* abs/1405.3694, doi:10.48550/arXiv.1405.3694.

[7] Remo MA van der Heiden, Shamsi T Iqbal & Christian P Janssen (2017): *Priming drivers before handover in semi-autonomous cars*. In: *Proceedings of the 2017 CHI conference on human factors in computing systems*, ACM, New York, NY, USA, pp. 392–404, doi:10.1145/3025453.3025507.

[8] Lei Hsiung, Yung-Ju Chang, Wei-Ko Li, Tsung-Yi Ho & Shan-Hung Wu (2022): *A Lab-Based Investigation of Reaction Time and Reading Performance Using Different In-Vehicle Reading Interfaces during Self-Driving*. In: *Proceedings of the 14th* , AutomotiveUI '22, ACM, New York, NY, USA, p. 96 – 107, doi:10.1145/3543174.3545254.

[9] Suraj Kothawade, Vinaya Khandelwal, Kinjal Basu, Huaduo Wang & Gopal Gupta (2021): *AUTO-DISCERN: Autonomous Driving Using Common Sense Reasoning*. In: *Proceedings of the International Conference on Logic Programming 2021 Workshops, Porto, Portugal (virtual)* , *CEUR Workshop Proceedings* 2970, CEUR-WS.org, doi:10.48550/arXiv.2110.13606.

[10] Robert Kowalski & Marek Sergot (1989): *A Logic-Based Calculus of Events*, pp. 23–51. Springer-Verlag, Berlin, Heidelberg, doi:10.1007/978-3-642-83397-7_2.

[11] Josef F. Krems & Martin R. K. Baumann (2009): *Driving and Situation Awareness: A Cognitive Model of Memory-Update Processes*. In Masaaki Kurosu, editor: *Human Centered Design*, Springer Berlin Heidelberg, pp. 986–994, doi:10.1007/978-3-642-02806-9_113.

[12] M. Kyriakidis, J. C. F. de Winter, N. Stanton, T. Bellet, B. van Arem, K. Brookhuis, M. H. Martens, K. Bengler, J. Andersson, N. Merat, N. Reed, M. Flament, M. Hagenzieker & R. Happee (2019): *A human factors perspective on automated driving*. *Theoretical Issues in Ergonomics Science* 20(3), pp. 223–249, doi:10.1080/1463922X.2017.1293187.

[13] Danh Le-Phuoc, Thomas Eiter & Anh Le-Tuan (2021): *A Scalable Reasoning and Learning Approach for Neural-Symbolic Stream Fusion*. *Proceedings of the AAAI Conference on Artificial Intelligence* 35(6), pp. 4996–5005, doi:10.1609/aaai.v35i6.16633.

[14] Jiefei Ma, Rob Miller, Leora Morgenstern & Theodore Patkos (2014): *An Epistemic Event Calculus for ASP-based Reasoning About Knowledge of the Past, Present and Future*. In: *LPAR: 19th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, *EPiC Series in Computing* 26, EasyChair, pp. 75–87, doi:10.29007/zswj.

[15] Rob Miller, Leora Morgenstern & Theodore Patkos (2013): *Reasoning About Knowledge and Action in an Epistemic Event Calculus*. In: *COMMONSENSE 2013*.

[16] Phillip L. Morgan, Chris Alford, Craig Williams, Graham Parkhurst & Tony Pipe (2018): *Manual Takeover and Handover of a Simulated Fully Autonomous Vehicle Within Urban and Extra-Urban Settings*. In *Advances in Human Aspects of Transportation*, Springer International Publishing, Cham, pp. 760–771, doi:10.1007/978-3-319-60441-1_73.

[17] Jan-Patrick Osterloh, Jochem W. Rieger & Andreas Lüdtke (2017): *Modelling Workload of a Virtual Driver*. In: *Proceedings of the 15th International Conference on Cognitive Modeling*.

[18] Amir Rasouli & John K. Tsotsos (2020): *Autonomous Vehicles That Interact With Pedestrians: A Survey of Theory and Practice*. *IEEE Transactions on Intelligent Transportation Systems* 21(3), pp. 900–918, doi:10.1109/TITS.2019.2901817.

[19] Umair Rehman, Shi Cao & Carolyn MacGregor (2019): *Using an Integrated Cognitive Architecture to Model the Effect of Environmental Complexity on Drivers' Situation Awareness*. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 63(1), pp. 812–816, doi:10.1177/1071181319631313.

[20] Christina Rödel, Susanne Stadler, Alexander Meschtscherjakov & Manfred Tscheligi (2014): *Towards Autonomous Cars: The Effect of Autonomy Levels on Acceptance and User Experience*. In: *Proceedings of the 6th* , AutomotiveUI '14, ACM, New York, NY, USA, p. 1 – 8, doi:10.1145/2667317.2667330.

[21] Jakob Suchan, Mehul Bhatt & Srikrishna Varadarajan (2019): *Out of Sight But Not Out of Mind: An Answer Set Programming Based Online Abduction Framework for Visual Sensemaking in Autonomous Driving*. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China*, ijcai.org, pp. 1879–1885, doi:10.24963/ijcai.2019/260.

[22] Jakob Suchan, Mehul Bhatt & Srikrishna Varadarajan (2021): *Commonsense visual sensemaking for autonomous driving - On generalised neurosymbolic online abduction integrating vision and semantics*. *Artificial Intelligence* 299, p. 103522, doi:10.1016/j.artint.2021.103522.

[23] Jakob Suchan, Mehul Bhatt, Przemyslaw Andrzej Walega & Carl Schultz (2018): *Visual Explanation by High-Level Abduction: On Answer-Set Programming Driven Reasoning About Moving Objects*. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence, (AAAI-18), New Orleans, Louisiana, USA*, AAAI Press, doi:10.1609/aaai.v32i1.11569.

[24] Efthimis Tsilionis, Nikolaos Koutroumanis, Panagiotis Nikitopoulos, Christos Doulkeridis & Alexander Artikis (2019): *Online Event Recognition from Moving Vehicles: Application Paper*. *Theory and Practice of Logic Programming* 19(5-6), doi:10.1017/S147106841900022X.

[25] ASAM e. V. (2021): *OpenDRIVE Format Specification*. Available at `https://www.asam.net/standards/detail/opendrive/`.

[26] Raphael Zimmermann & Reto Wettach (2017): *First step into visceral interaction with autonomous vehicles*. In: *Proceedings of the 9th International Conference on Automotive User Interfaces and Interactive Vehicular Applications*, pp. 58–64, doi:10.1145/3122986.3122988.

On the Development of PASTA: Inference in Probabilistic Answer Set Programming under the Credal Semantics

Damiano Azzolini
University of Ferrara
Ferrara, Italy
damiano.azzolini@unife.it

PASTA is a novel framework to perform inference, both exact and approximate, in Probabilistic Answer Set Programming under the Credal Semantics and to solve other related tasks, such as abduction, decision theory, maximum a posteriori inference, and parameter learning. Moreover, it also allows the expression of statistical statements. This paper shows the overall structure of the PASTA tool as well as some of the main options.

1 Description

Answer Set Programming (ASP) is a logic-based language useful in describing complex combinatorial domains. Most of ASP research focuses on deterministic programs. There are some semantics that extend ASP and allow uncertainty on data, mainly LP^{MLN} [15], P-log [7], and the credal semantics [9]. The PASTA tool considers the last.

A probabilistic answer set program under the credal semantics is an answer set program extended with probabilistic facts [10], i.e., constructs of the form $\Pi :: a.$ where a is an atom and $\Pi \in [0,1]$ is its probability value. The intuitive meaning is that a is true with probability Π. We assume that probabilistic facts cannot appear as head of rules. The following is an example of a program:

```
0.4::a.
0.7::b.
qr:- a.
qr ; nqr:- b.
```

A selection of a truth value for every probabilistic fact defines a *world* w, i.e., an answer set program, whose probability can be computed as $P(w) = \prod_{f_i \text{ selected}} \Pi_i \cdot \prod_{f_i \text{ not selected}} (1 - \Pi_i)$. The previously shown program has $2^2 = 4$ worlds. The probability of the world, for example, where both a and b are true is $0.4 \cdot 0.7 = 0.28$. Since every world is an answer set program [14], every world has zero or more answer sets. The credal semantics requires that every world has at least one answer set. The probability of a *query* (conjunction of ground atoms) $P(q)$ is defined by a range, with a lower and upper probability, i.e., $P(q) = [\underline{P}(q), \overline{P}(q)]$. A world contributes to the upper probability if it has at least one answer set where the query is true and it contributes to both the lower and upper probability if the query is true in all the answer sets. That is, the lower probability is the sum of the probabilities of the worlds where the query is true in all the answer sets while the upper probability is the sum of the probabilities of the worlds where the query is true in at least one answer set. In formula

$$\underline{P}(q) = \sum_{w_i \mid \forall m \in AS(w_i),\ m \models q} P(w_i),\ \overline{P}(q) = \sum_{w_i \mid \exists m \in AS(w_i),\ m \models q} P(w_i). \quad (1)$$

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 314–316, doi:10.4204/EPTCS.385.30

Consider the query qr in the previous program. The world where both a and b are false has one empty answer set, so it does not contribute to the probability. The world where a is false and b is true has two answer sets, $\{\{b\ qr\},\{b\ nqr\}\}$. The query is true in only one of the two, so we have a contribution only to the upper probability of $0.6 \cdot 0.7 = 0.42$. The world where a is true and b is false has one answer set, $\{\{a\ qr\}\}$. The query is true in all the answer sets, so we have a contribution of $0.4 \cdot 0.3 = 0.12$ to both the lower and upper probability. Lastly, the world where both a and b are true has one answer set, $\{\{a\ b\ qr\}\}$. The query is true in all the answer sets, so we have a contribution of $0.4 \cdot 0.7 = 0.28$ to both the lower and upper probability. Overall, $P(q) = [0.12+0.28, 0.42+0.12+0.28] = [0.4, 0.82]$.

Currently, the PASTA tool performs exact inference [2], whose algorithm is based on projected answer set enumeration [13], approximate inference via sampling [3], abduction [1] (i.e., computing the minimal set, in terms of set inclusion, of abducible facts such that the probability of the query is maximized), MAP/MPE [5] (finding an assignment to a subset of probabilistic facts that maximizes the sum of the probabilities of the identified worlds while evidence is satisfied), decision theory [6] (finding a subset of decision atoms that maximizes a reward), and parameter learning [4] (learning the probability of the probabilistic facts such that the likelihood of the examples is maximized). It allows to express statistical statements of the form [2] $(C|A)[\Pi_l, \Pi_u]$ meaning that the fraction of As that are also Cs is between Π_l and Π_u. Moreover, it adopts normalization [11] to perform inference in probabilistic answer set programs where some worlds have no answer sets. That is, the lower and upper probability bounds are divided by the sum of the probabilities of the worlds with no answer sets.

The system is written in Python, is open source, and is available on GitHub[1]. It can be installed via the Python package installer `pip` and provides a command line interface with several options to select the task to solve. Moreover, a web interface[2] exposes some of the most relevant options in a more intuitive way. It is also possible to use the system as a Python library. PASTA leverages the clingo ASP solver [12] to compute answer sets. The system is structured in multiple modules: the `pasta` module parses the command line arguments and selects the subsequent functions and methods to call. The `generator` module parses the program into an ASP version that can be interpreted by the clingo solver. The interface with clingo is managed by the `asp_interface` module and the resulting answer sets are analyzed with the `models_handler` module which also computes the various probabilities. There are some reserved keywords: `abducible`, that denotes abducible facts in the context of abduction (for example, `abducible a` states that the atom a is an abducible); `map`, that denotes facts to be considered for the MAP/MPE task (for example, `map 0.4::a` states that a is a probabilistic fact whose probability is 0.4 and that can be selected to be included in the solution set for the MAP/MPE task); and `decision` and `utility`, that mark decision facts and utility attributes [8] for the decision theory task (for example, `decision a` states that a is a decision atom and `utility(f,2.3)` states that the reward obtained when f is true is 2.3).

Acknowledgements

This research was partly supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No. 952215.

---

[1] `https://github.com/damianoazzolini/pasta`

[2] `https://github.com/damianoazzolini/pasta_website`

References

[1] Damiano Azzolini, Elena Bellodi, Stefano Ferilli, Fabrizio Riguzzi & Riccardo Zese (2022): *Abduction with probabilistic logic programming under the distribution semantics*. International Journal of Approximate Reasoning 142, pp. 41–63, doi:10.1016/j.ijar.2021.11.003.

[2] Damiano Azzolini, Elena Bellodi & Fabrizio Riguzzi (2022): *Statistical Statements in Probabilistic Logic Programming*. In Georg Gottlob, Daniela Inclezan & Marco Maratea, editors: *Logic Programming and Nonmonotonic Reasoning*, Springer International Publishing, Cham, pp. 43–55, doi:10.1007/978-3-031-15707-3_4.

[3] Damiano Azzolini, Elena Bellodi & Fabrizio Riguzzi (2023): *Approximate Inference in Probabilistic Answer Set Programming for Statistical Probabilities*. In Agostino Dovier, Angelo Montanari & Andrea Orlandini, editors: *AIxIA 2022 – Advances in Artificial Intelligence*, Springer International Publishing, Cham, pp. 33–46, doi:10.1007/978-3-031-27181-6_3.

[4] Damiano Azzolini, Elena Bellodi & Fabrizio Riguzzi (2023): *Learning the Parameters of Probabilistic Answer Set Programs*.

[5] Damiano Azzolini, Elena Bellodi & Fabrizio Riguzzi (2023): *MAP Inference in Probabilistic Answer Set Programs*. In Agostino Dovier, Angelo Montanari & Andrea Orlandini, editors: *AIxIA 2022 – Advances in Artificial Intelligence*, Springer International Publishing, Cham, pp. 413–426, doi:10.1007/978-3-031-27181-6_29.

[6] Damiano Azzolini, Elena Bellodi & Fabrizio Riguzzi (2023): *Towards a Representation of Decision Theory Problems with Probabilistic Answer Set Programs*.

[7] Chitta Baral, Michael Gelfond & Nelson Rushton (2009): *Probabilistic reasoning with answer sets*. Theory and Practice of Logic Programming 9(1), pp. 57–144, doi:10.1017/S1471068408003645.

[8] Guy Van den Broeck, Ingo Thon, Martijn van Otterlo & Luc De Raedt (2010): *DTProbLog: A Decision-Theoretic Probabilistic Prolog*. In Maria Fox & David Poole, editors: *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI Press, pp. 1217–1222.

[9] Fábio Gagliardi Cozman & Denis Deratani Mauá (2020): *The joy of Probabilistic Answer Set Programming: Semantics, complexity, expressivity, inference*. International Journal of Approximate Reasoning 125, pp. 218–239, doi:10.1016/j.ijar.2020.07.004.

[10] Luc De Raedt, Angelika Kimmig & Hannu Toivonen (2007): *ProbLog: A Probabilistic Prolog and Its Application in Link Discovery*. In Manuela M. Veloso, editor: *20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, 7, AAAI Press, pp. 2462–2467.

[11] Daan Fierens, Guy Van den Broeck, Maurice Bruynooghe & Luc De Raedt (2012): *Constraints for probabilistic logic programming*. In Daniel M. Roy, Vikash K. Mansinghka & Noah D. Goodman, editors: *Proceedings of the NIPS Probabilistic Programming Workshop*.

[12] Martin Gebser, Roland Kaminski, Benjamin Kaufmann & Torsten Schaub (2019): *Multi-shot ASP solving with clingo*. Theory and Practice of Logic Programming 19(1), pp. 27–82, doi:10.1017/S1471068418000054.

[13] Martin Gebser, Benjamin Kaufmann & Torsten Schaub (2009): *Solution Enumeration for Projected Boolean Search Problems*. In Willem-Jan van Hoeve & John Hooker, editors: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Springer Berlin Heidelberg, pp. 71–86, doi:10.1007/978-3-642-01929-6_7.

[14] Michael Gelfond & Vladimir Lifschitz (1988): *The stable model semantics for logic programming*. In: *5th International Conference and Symposium on Logic Programming (ICLP/SLP 1988)*, 88, MIT Press, pp. 1070–1080.

[15] Joohyung Lee & Yi Wang (2016): *Weighted Rules under the Stable Model Semantics*. In Chitta Baral, James P. Delgrande & Frank Wolter, editors: *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016*, AAAI Press, pp. 145–154.

Logical English Demonstration

Robert Kowalski,
Department of Computing, Imperial College, London, UK
r.kowalski@imperial.ac.uk

Jacinto Dávila
Universidad de Los Andes, Mérida, Venezuela
jacinto@ula.ve

Logical English (LE) is a natural language syntax for pure Prolog and other logic programming languages, such as ASP and s(CASP). Its main applications until now have been to explore the representation of a wide range of legal texts, helping to identify ambiguities, explore alternative representations of the same text, and compare the logical consequences of the alternatives. The texts include portions of loan agreements, accountancy law, Italian citizenship, EU law on criminal rights, International Swaps and Derivative contracts, and insurance contracts. The current implementation in SWI Prolog can be accessed at `https://logicalenglish.logicalcontracts.com`.

1 Introduction to Logical English

The basic form of LE is simply syntactic sugar for pure Prolog[2], with predicates written in infix form and declared by means of templates, as in:

```
*a borrower* defaults on *a date*
```

where asterisks delimit the argument places of the predicate. Variables are signalled by the use of one of the determiners “a”, “an” or “the”. An indefinite determiner, “a” or “an”, introduces the first occurrence of a variable in a sentence. All later occurrences of the same variable in the same sentence are prefixed by the definite determiner “the”.

LE has only minimal linguistic knowledge of English. Its knowledge of English vocabulary is restricted to the determiners; the logical connectives “if”, “and”, “or” and “it is not the case that”; the logical pattern “for all cases in which…it is the case that…”; and the logical keyword “that”. The keyword “that” identifies the use of a meta-predicate, for representing such “propositional attitudes as prohibition, obligation, belief, desire, fear, notification, etc. Indentation, rather than parentheses, is used to indicate the relative strength of binding of the logical connectives. LE has virtually no knowledge of English grammar. In particular, it does not distinguish between singular and plural nouns and verbs, and it does not know about the relationship between the different tenses of verbs. Despite these restrictions, and because it has the same expressiveness as pure Prolog, it can be used to represent a broad range of knowledge, as shown by its application to the representation of legal texts [3, 4, 5, 6, 7]. Here is an example based on the loan agreement in [1]. The SWISH implementation of the example can be found at `https://logicalenglish.logicalcontracts.com/p/new_loan_with_cure.pl`.

Ordinary English: The Borrower will be in default under this agreement upon the failure of the Borrower to fulfil any obligation of this agreement, provided the failure shall remain uncured within a period of two days after notice is given to the Borrower by the Lender of the failure (such an uncured event an “Event of Default”).

Logical English:

```
the borrower defaults on a date D2
if the borrower has the obligation that an eventuality
```

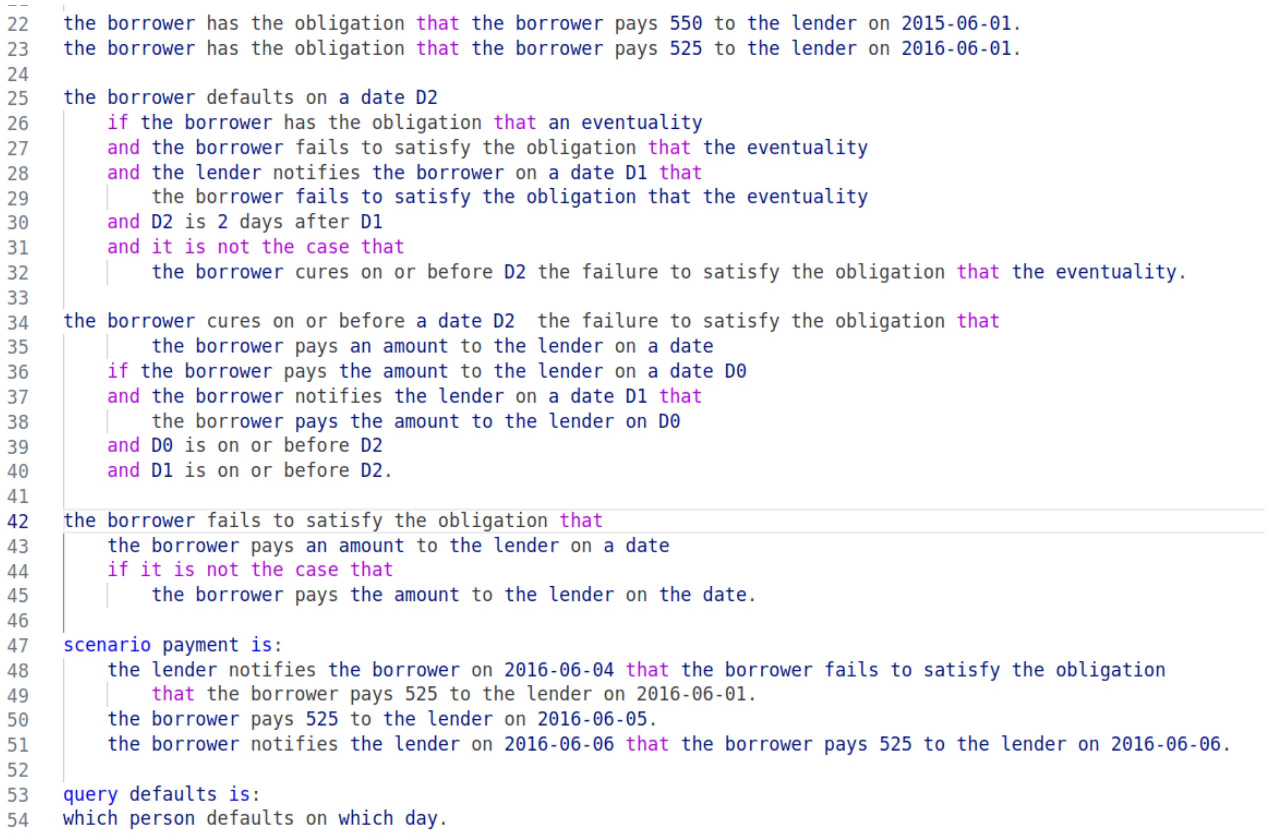
S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D’Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 317–320, doi:10.4204/EPTCS.385.31

```
and the borrower fails to satisfy the obligation that the eventuality
and the lender notifies the borrower on a date D1 that
    the borrower fails to satisfy the obligation that the eventuality
and D2 is 2 days after D1
and it is not the case that
the borrower cures on or before D2 the failure to satisfy the obligation that
    the eventuality.
```

Prolog:

```
defaults_on(the_borrower, A) :-
    has_the_obligation_that(the_borrower, B),
    fails_to_satisfy_the_obligation_that(the_borrower, B),
    notifies_on_that(the_lender,  the_borrower, C,
        fails_to_satisfy_the_obligation_that(the_borrower, B)),
    is_days_after(A, 2, C),
    not cures_on_or_before_the_failure_to_satisfy_the_obligation_that(the_borrower, A, B).
```

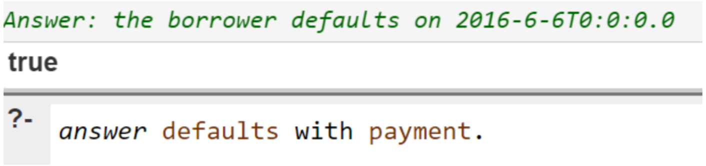
Notice that the original English suggests that an Event of Default occurs retroactively on the date D0 of the failure to fulfil an obligation. However, elsewhere the loan agreement states that "upon the occurrence of an Event of Default all outstanding payments under this agreement will become immediately due and payable". To avoid inconsistency, the LE version represents the intended interpretation as being that the default occurs on the date D2, two days after the notification of the failure on date D1.



```
22  the borrower has the obligation that the borrower pays 550 to the lender on 2015-06-01.
23  the borrower has the obligation that the borrower pays 525 to the lender on 2016-06-01.
24
25  the borrower defaults on a date D2
26      if the borrower has the obligation that an eventuality
27      and the borrower fails to satisfy the obligation that the eventuality
28      and the lender notifies the borrower on a date D1 that
29          the borrower fails to satisfy the obligation that the eventuality
30      and D2 is 2 days after D1
31      and it is not the case that
32          the borrower cures on or before D2 the failure to satisfy the obligation that the eventuality.
33
34  the borrower cures on or before a date D2  the failure to satisfy the obligation that
35          the borrower pays an amount to the lender on a date
36      if the borrower pays the amount to the lender on a date D0
37      and the borrower notifies the lender on a date D1 that
38          the borrower pays the amount to the lender on D0
39      and D0 is on or before D2
40      and D1 is on or before D2.
41
42  the borrower fails to satisfy the obligation that
43      the borrower pays an amount to the lender on a date
44      if it is not the case that
45          the borrower pays the amount to the lender on the date.
46
47  scenario payment is:
48      the lender notifies the borrower on 2016-06-04 that the borrower fails to satisfy the obligation
49          that the borrower pays 525 to the lender on 2016-06-01.
50      the borrower pays 525 to the lender on 2016-06-05.
51      the borrower notifies the lender on 2016-06-06 that the borrower pays 525 to the lender on 2016-06-06.
52
53  query defaults is:
54  which person defaults on which day.
```

Figure 1: Rules and payment scenario

Figure 1 illustrates a scenario, called "payment", in which the borrower fails to satisfy the obligations, on lines 22 and 23, to pay the lender 550 on 2015-06-01 and 525 on 2016-06-01. The lender does not notice the first failure, but notices the second failure, and gives notice to the borrower of the second failure on 2016-06-04. The borrower attempts to cure the failure, by paying the correct amount 525 and by notifying the lender of the payment within the two day period of grace. But unfortunately, the borrower notifies the lender incorrectly that the payment was made on the date of notification rather than on the date of payment.

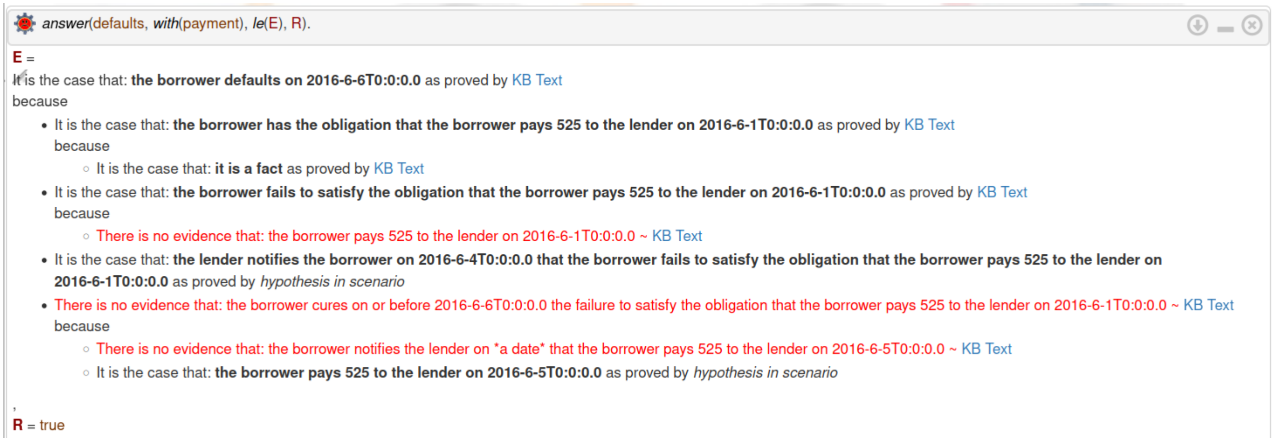


```
Answer: the borrower defaults on 2016-6-6T0:0:0.0
true
?- answer defaults with payment.
```

Figure 2: Querying LE

Figure 2 illustrates the result of answering the combination of the stored query, called "defaults" with the scenario. An LE document can contain several stored scenarios and several stored queries, which can be combined in the SWISH query pane.

The SWISH implementation also generates explanations in response to commands of the form $answer(defaults, with(payment), le(E), R)$ as shown in figure 3. For VSC users there exist extensions for syntax highlighting and remote execution on a SWISH server. It is also possible to call the LE parser without the SWISH environment, as a standalone Prolog application. All the sources and further information are available at `https://github.com/LogicalContracts/LogicalEnglish/`.



```
answer(defaults, with(payment), le(E), R).
E =
It is the case that: the borrower defaults on 2016-6-6T0:0:0.0 as proved by KB Text
because
  • It is the case that: the borrower has the obligation that the borrower pays 525 to the lender on 2016-6-1T0:0:0.0 as proved by KB Text
    because
      ○ It is the case that: it is a fact as proved by KB Text
  • It is the case that: the borrower fails to satisfy the obligation that the borrower pays 525 to the lender on 2016-6-1T0:0:0.0 as proved by KB Text
    because
      ○ There is no evidence that: the borrower pays 525 to the lender on 2016-6-1T0:0:0.0 ~ KB Text
  • It is the case that: the lender notifies the borrower on 2016-6-4T0:0:0.0 that the borrower fails to satisfy the obligation that the borrower pays 525 to the lender on
    2016-6-1T0:0:0.0 as proved by hypothesis in scenario
  • There is no evidence that: the borrower cures on or before 2016-6-6T0:0:0.0 the failure to satisfy the obligation that the borrower pays 525 to the lender on 2016-6-1T0:0:0.0 ~ KB Text
    because
      ○ There is no evidence that: the borrower notifies the lender on *a date* that the borrower pays 525 to the lender on 2016-6-5T0:0:0.0 ~ KB Text
      ○ It is the case that: the borrower pays 525 to the lender on 2016-6-5T0:0:0.0 as proved by hypothesis in scenario

,
R = true
```

Figure 3: Explanations in LE

Acknowledgements

Many thanks to Miguel Calejo, Galileo Sartor, Andrew Noble, John Cummins, Fariba Sadri and Nilokai Merritt for their support and contributions to this work.

References

[1] Flood, M.D. and Goodenough, O.R., 2017. Contract as automaton: The computational representation of financial agreements. Office of Financial Research Working Paper, (15-04). doi:10.1007/s10506-021-09300-9

[2] Kowalski, R.: Logical English, In Proceedings of Logic and Practice of Programming (LPOP) (2020). `http://www.info.ucl.ac.be/~pvr/LPOP_2020_proceedings.pdf`

[3] Kowalski, R., Datoo, A.: Logical English meets legal English for swaps and derivatives. Artificial Intelligence and Law, 30(2), 163-197 (2022). doi:10.1007/s10506-021-09295-3.

[4] Kowalski, R., Dávila, J., Calejo, M., Logical English for legal applications. XAIF, Virtual Workshop on Explainable AI in Finance (2021).

[5] Kowalski, R., Dávila, J., Sartor, G., Calejo, M.: Logical English for Law. In Proceedings of the Workshop on Methodologies for Translating Legal Norms into Formal Representations (LN2FR), JURIX, (2022). `https://research.nii.ac.jp/~ksatoh/LN2FRproceedings.pdf`

[6] Kowalski, R., Sadri, F., Calejo, M., Dávila, J.: Combining Logic Programming and Imperative Programming in LPS. In: Warren, D., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years. LNCS, vol. 13900. Springer, Heidelberg (2023).

[7] Sartor, G., Dávila, J., Billi, M., Contissa, G., Pisano, G., Kowalski, R.: Integration of Logical English and s(CASP), 2nd Workshop on Goal-directed Execution of Answer Set Programs (GDE'22) (2022). `https://ceur-ws.org/Vol-3193/paper5GDE.pdf`

ForgettingWeb

Matti Berthold
ScaDS.AI, Dresden/Leipzig,
Universität Leipzig,
Germany
berthold@informatik.uni-leipzig.de

Matthias Knorr
NOVA LINCS
Universidade Nova de Lisboa,
Portugal
mkn@fct.unl.pt

Daphne Odekerken
Utrecht University,
The Netherlands
National Police Lab AI,
Netherlands Police
d.odekerken@uu.nl

The relatively young area of *forgetting* is concerned with the removal of selective information, while preserving other knowledge. This might be useful or even necessary, for example, to simplify a knowledge base or to tend legal requests. In the last few years, there has been an ample amount of research in the field, in particular with respect to logic programs, spanning from theoretical considerations to more practical applications, starting at the conceptual proposal of forgetting, to suggestions of properties that should be satisfied, followed by characterizations of abstract classes of operators that satisfy these properties, and finally the definition of concrete forgetting procedures.

In this work we present novel Python implementations of all the forgetting procedures that have been proposed to date on logic programs. We provide them in a web interface, and hope to thereby give anybody who is interested a low-barrier overview of the landscape.

An Overview

While dynamics in knowledge representation are often examined with respect to the addition of new information, there is an increasing amount of research on how it may be removed. *Forgetting*, or variable elimination, seeks to make a knowledge base independent of elements of the underlying formal language, while keeping as many logical connections between the remaining elements as possible. Though it was initially proposed as a semantical notion over classical formulas [LR94], in the context of logic programming forgotten atoms are usually also required to be removed syntactically. More broadly, there have been several suggestions for properties that a plausible forgetting procedure should satisfy [EW08, ZZ09, WZZZ12, WWZ13, KA14, DW15], cf. [GKL16b] for an extensive overview.

Along the way, there have also been a number of suggestions for concrete syntactical transformations to forget atoms from a program [ZF06, EW08, KA14, BGKL19, GJKL21, ACF$^+$22b, ACF$^+$22a, Ber22] that satisfy some of the proposed properties, or at least satisfy them whenever possible [GKL16a]. The hope for such operators is to produce forgetting results that in some way resemble their origin, while avoiding a computational blow-up as much as possible. Given a semantic definition of a class of forgetting operators, e.g. [GKLW17, GJK$^+$19], the construction of a program from the desired HT-models remains a baseline that can be employed to confirm the existence of a concrete forgetting operator [CF07].

The definitions of the syntactic forgetting procedures are rather involved, sometimes spanning over several pages of text. In order to make them more accessible, we compiled encodings as well as short explanations, of the operators (those that are defined over logic programs) and the construction of canonical programs into a web interface.[1] We hope to thereby give anybody who is interested a good overview of the material.

A screenshot of the interface is shown in Figure 1. In the leftmost column, users can specify a logic program and the atoms to be forgotten, or select a predefined sample input. Subsequently, they select the

---

[1] ForgettingWeb can be accessed here. Its source code is available here.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 321–323, doi:10.4204/EPTCS.385.32

ForgettingWeb

Input

Select a predefined example from the list below or insert your own program and atoms to be forgotten.

demofile

Input program P

```
a :- q
b :- ~q
q :- c
```

Forgetting atoms V

q

Operator

Choose the operator to be applied.

f^*_{SP} (Berthold 2022)

Result

New program

```
a :- c
b :- ~a
b :- ~c
```

By Matti Berthold and Daphne Odekerken

Figure 1: Screenshot of ForgettingWeb.

forgetting operator of their choice from the dropdown in the middle column. These are implementations from operators proposed in the literature. For more details, the user presses the "Operator Explanation" button: this triggers a pop-up with an explanation of the operator and a reference to the paper in which it was proposed. After pressing the "Forget it!" button, the result program appears in the rightmost column.

An input program P is specified by one rule per line, where the head atoms are separated by ‘`;`’, the head and the body are separated by ‘`:-`’, and the body literals are separated by commas. As negation signs both the traditional ‘`not`’ and the shorter ‘$\sim$’ can be used. No dot at the end of a rule is required. The atoms of the forgetting set V are separated by commas. In the “Result” column on the right side, the new program has the same syntax as the input program.

The input of the counter-models construction aux_{cm} are tuples with delimiters ‘<’ and ‘>’, e.g.:

```
<ab,ab><a,ab><b,ab><,ab>
<a,a><,>
```

Each character within an element of a tuple is assumed to be a single element of a set.

Acknowledgements

The authors have been arranged alphabetically. The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany and by the Sächsische Staatsministerium für Wissenschaft Kultur und Tourismus in the program Center of Excellence for AI-research "Center for Scalable Data Analytics and Artificial Intelligence Dresden/Leipzig", project identification number: ScaDS.AI. This work was partially supported by FCT project FORGET (PTDC/CCI-INF/32219/2017) and by FCT project NOVA LINCS (UIDB/04516/2020).

References

[ACF$^+$22a] F. Aguado, P. Cabalar, J. Fandinno, D. Pearce, G. Pérez & C. Vidal (2022): *A polynomial reduction of forks into logic programs*. Artif. Intell. 308, p. 103712, doi:10.1016/j.artint.2022.103712.

[ACF$^+$22b] F. Aguado, P. Cabalar, J. Fandinno, D. Pearce, G. Pérez & C. Vidal (2022): *Syntactic ASP Forgetting with Forks*. In: Proceedings of (LPNMR-22), Lecture Notes in Computer Science 13416, Springer, pp. 3–15, doi:10.1007/978-3-031-15707-3_1.

[Ber22] M. Berthold (2022): *On Syntactic Forgetting with Strong Persistence*. In: Proceedings of (KR-22), doi:10.24963/kr.2022/5.

[BGKL19] M. Berthold, R. Gonçalves, M. Knorr & J. Leite (2019): *A Syntactic Operator for Forgetting that Satisfies Strong Persistence*. Theory and Practice of Logic Programming 19(5-6), p. 1038–1055, doi:10.1017/S1471068419000346.

[CF07] P. Cabalar & P. Ferraris (2007): *Propositional theories are strongly equivalent to logic programs*. Theory and Practice of Logic Programming 7(6), pp. 745–759, doi:10.1017/S1471068407003110.

[DW15] J.P. Delgrande & K. Wang (2015): *A Syntax-Independent Approach to Forgetting in Disjunctive Logic Programs*. In: Proceedings of (AAAI-15), pp. 1482–1488, doi:10.1609/aaai.v29i1.9402.

[EW08] T. Eiter & K. Wang (2008): *Semantic forgetting in answer set programming*. Artificial Intelligence 172(14), pp. 1644–1672, doi:10.1016/j.artint.2008.05.002.

[GJK$^+$19] R. Gonçalves, T. Janhunen, M. Knorr, J. Leite & S. Woltran (2019): *Forgetting in Modular Answer Set Programming*. In: Proceedings of (AAAI-19), AAAI Press, pp. 2843–2850, doi:10.1609/aaai.v33i01.33012843.

[GJKL21] R. Gonçalves, T. Janhunen, M. Knorr & J. Leite (2021): *On Syntactic Forgetting Under Uniform Equivalence*. In: Proceedings of (JELIA-21), Lecture Notes in Computer Science 12678, Springer, pp. 297–312, doi:10.1007/978-3-030-75775-5_20.

[GKL16a] R. Gonçalves, M. Knorr & J. Leite (2016): *The Ultimate Guide to Forgetting in Answer Set Programming*. In: Proceedings of (KR-16), pp. 135–144.

[GKL16b] R. Gonçalves, M. Knorr & J. Leite (2016): *You Can't Always Forget What You Want: On the Limits of Forgetting in Answer Set Programming*. In: Proceedings of (ECAI-16), pp. 957–965, doi:10.3233/978-1-61499-672-9-957.

[GKLW17] R. Gonçalves, M. Knorr, J. Leite & S. Woltran (2017): *When you must forget: Beyond strong persistence when forgetting in answer set programming*. Theory and Practice of Logic Programming 17(5-6), pp. 837–854, doi:10.1017/S1471068417000382.

[KA14] M. Knorr & J.J. Alferes (2014): *Preserving Strong Equivalence while Forgetting*. In: Proceedings of (JELIA-14), LNCS 8761, Springer, pp. 412–425, doi:10.1007/978-3-319-11558-0_29.

[LR94] F. Lin & R. Reiter (1994): *Forget it*. In: Working Notes of AAAI Fall Symposium on Relevance, pp. 154–159.

[WWZ13] Y. Wang, K. Wang & M. Zhang (2013): *Forgetting for Answer Set Programs Revisited*. In: Proceedings of (IJCAI-13), IJCAI/AAAI, pp. 1162–1168. Available at `http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6807`.

[WZZZ12] Y. Wang, Y. Zhang, Y. Zhou & M. Zhang (2012): *Forgetting in Logic Programs under Strong Equivalence*. In: Proceedings of (KR-12), AAAI Press.

[ZF06] Y. Zhang & N.Y. Foo (2006): *Solving logic program conflict through strong and weak forgettings*. Artificial Intelligence 170(8-9), pp. 739–778, doi:10.1016/j.artint.2006.02.002.

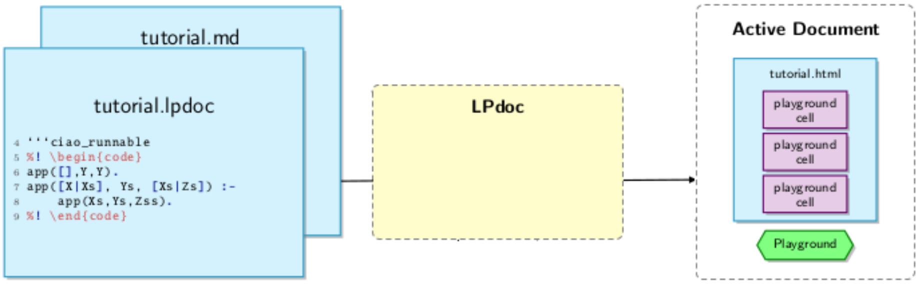
[ZZ09] Y. Zhang & Y. Zhou (2009): *Knowledge forgetting: Properties and applications*. Artificial Intelligence 173(16), pp. 1525–1537, doi:10.1016/j.artint.2009.07.005.

Demonstrating (Hybrid) Active Logic Documents and the Ciao Prolog Playground, and an Application to Verification Tutorials[*]

[1,2]Daniela Ferreiro, [1,2]José F. Morales, [3] Salvador Abreu and [1,2]Manuel V. Hermenegildo
[1] Universidad Politécnica de Madrid (UPM) and [2]IMDEA Software Institute, Madrid, Spain
[3]NOVA LINCS / University of Évora, Portugal
`{daniela.ferreiro,josef.morales,manuel.hermenegildo}@imdea.org; spa@uevora.pt`

We propose a demonstration of the Active Logic Documents (ALDs) approach and the `Ciao` Playground, as well as a recent extension to ALDs to facilitate the integration of other tools into the system for creating Hybrid Active Logic Documents (HALD), and a concrete application of these technologies.

Active Logic Documents (ALD) [10, 11] are web pages which incorporate embedded Prolog engines. ALD's are generated using the `LPdoc` documentation generation system [7, 6], extended to be able to include embedded `Ciao` Playground cells in documents, so that interaction with Prolog is possible within the documents produced (Fig. 1). Documents can contain integrated environments that allow for the processing of code blocks, including compiling, parsing, and analyzing them. This facilitates the creation of web-based materials with editable and runnable educational resources such as activities, runnable examples, programming exercises, etc. With the ability to directly edit and evaluate code within the document, these interactive documents can act as oracles, providing valuable feedback and supporting self-evaluation mechanisms. In comparison with other tools [14, 2, 1], two fundamental aspects of the Active Logic Documents approach are that a) all the reactive parts run locally on the user's web browser without any dependency on a central server or a local Prolog installation, and that b) output is generated from a single, easy to use source that can be developed with any editor. We argue that the ALD approach has multiple advantages from the point of view of scalability, low maintenance cost, security, privacy, ease of packaging and distribution, etc. over other approaches. As an example, the ALD source and output for a simple Prolog exercise[1] is shown in Fig. 4. Active Logic Documents are used for development of materials for teaching logic programming [8], embedding runnable code and exercises in slides [2], manuals, etc., and in other projects, such as, for example, in the development of a Programming course for young children (around 10 years old) within the Year Of Prolog initiatives.



```
'''ciao_runnable
%! \begin{code}
app([],Y,Y).
app([X|Xs], Ys, [Xs|Zs]) :-
    app(Xs,Ys,Zss).
%! \end{code}
```

Figure 1: Generating Active Logic Documents.

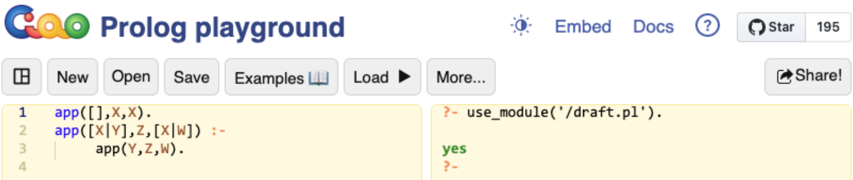
---

[*] This research has received funding from the *PROCODE* Project (PID2019-108528RB-C21/MCIN/AEI/10.13039/501100011033) and the *PRODIGY* Project (TED2021-132464B-I00) funded by MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU/ PRTR.

[1] `http://ciao-lang.org/ciao/build/doc/ciao_playground.html/factorial_peano_iso.html`

[2] E.g., Course material in Computational Logic: `https://cliplab.org/~logalg`

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 324–329, doi:10.4204/EPTCS.385.33



```
app([],X,X).
app([X|Y],Z,[X|W]) :-
    app(Y,Z,W).
```

```
?- use_module('/draft.pl').

yes
?-
```

Figure 2: The `Ciao` Prolog Playground

The `Ciao` Prolog Playground [3, 4] is a key component of our approach. In its standalone form,[3] it allows editing and running code locally on the user's web browser (See Fig. 2). To this end, the playground uses modern Web technology (WebAssembly and Emscripten) to run an off-the-shelf Prolog engine and top level *directly in the browser*, with access to browser-side local resources, including a full interface with JavaScript that enables graphical input and output, interactivity, etc. This engine is contained in `Ciaowasm`, a `Ciao` Prolog bundle that compiles a variant of the standard `Ciao` engine with all necessary `Ciao` bundles and manuals. The main advantage of this general architecture over the widely used server-based approaches such as SWISH [14, 2] or Jupyter notebooks [1], is that it is easily reproducible and significantly alleviates maintenance effort and cost, as it essentially eliminates the server-side infrastructure. Previously, compilation of Prolog to JavaScript [9] enabled the use of Prolog to develop also the client side of web applications, running fully on the browser. This functionality is also provided by, e.g., Tau Prolog [13] and the tuProlog playground [4] which are recent Prolog interpreters written in JavaScript. However, the approach of compilation to WebAssembly and Emscripten allows the immediate reuse of existing, well-developed engines (which include many non trivial optimizations and features) and tooling.

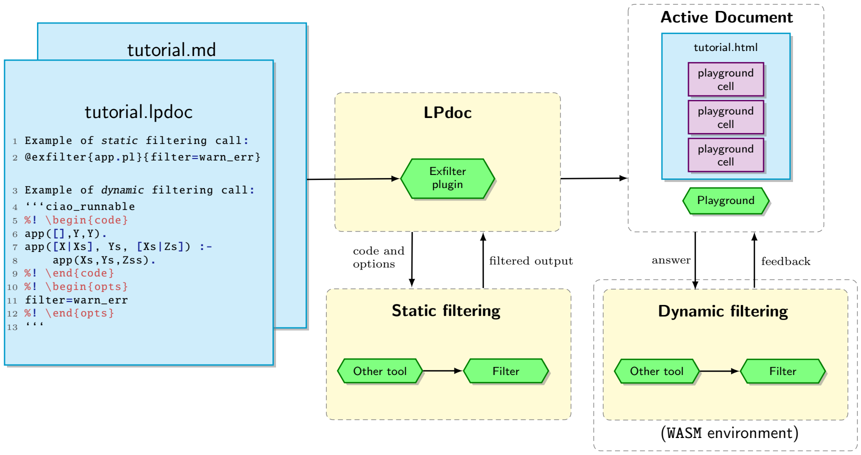
Hybrid Active Logic Documents (HALDs). This addition to the ALD architecture, illustrated in Fig. 3, allows for the integration into ALDs of output from other tools, that can be run either at document generation time (to produce *static* content) or during the user interactions with the generated documents (producing the content *dynamically*). The overall input is, as in Fig. 1, a set of `LPdoc` source files (e.g., in `.md` markdown format or `.lpdoc` documentation files) including code blocks, narrative text, etc. In the **static phase**, center of Fig. 3, `LPdoc` scans and composes all these files, processing the different elements and inserting embedded Prolog playground instances (editors, engine instances, query blocks, etc.) as necessary. In addition, in the case of a *hybrid* document, one or more auxiliary tool(s) are additionally run by `LPdoc` while generating the ADL so that selected output from such tools can be incorporated. We refer to this process of projection of tool output as **filtering**. There are a number of library filters available for this purpose, and this set can be extended by the user. When `LPdoc` finds filtering calls (again, center of Fig.3, and see also Fig. 5), it sends requests to the corresponding tool (including code fragments, command line options, queries at its top level, etc.), and then *filters* the output obtained and incorporates this projected output statically into the document being produced. This is very useful for example when writing lecture notes, tutorials, manuals, etc.: when including, e.g., the results to a query or exercise, instead of pasting them in manually, these results can be generated automatically from the tool being used or documented. This ensures that the contents and format of the outputs produced by the tool are kept automatically in sync with any changes in the tool, which is a tedious and error-prone task without automation. This includes also for example obtaining from the tool the results to compare to in student exercises.

The **dynamic phase** (right of Fig.3) occurs once the HTML pages produced by `LPdoc` are deployed, and users have loaded them into their browsers. When processing some user input in one of the interactive elements of the pages –e.g., when a student issues a solution to an exercise–, the embedded playground framework issues the intervening filtering calls *dynamically*. This process involves loading the program, calling the corresponding tool (via command line interface or interacting with its top level), obtaining the

---

[3] `https://ciao-lang.org/playground`

[4] `https://pika-lab.gitlab.io/tuprolog/2p-kt-web`



```
Example of static filtering call:
@exfilter{app.pl}{filter=warn_err}

Example of dynamic filtering call:
‘‘‘ciao_runnable
%! \begin{code}
app([],Y,Y).
app([X|Xs], Ys, [Xs|Zs]) :-
    app(Xs,Ys,Zss).
%! \end{code}
%! \begin{opts}
filter=warn_err
%! \end{opts}
‘‘‘
```

Figure 3: Overall architecture of hybrid ALDs.

output from this execution, and processing it with the specified filter in order to select the relevant part before presenting it to the user. Filtering can be customized to accommodate various types of tasks, such as "fill in the blanks" style exercises or to facilitate understanding and remediation of testing and verification failures, by displaying warning or error messages. Thus, students can test their code rapidly and effectively and receive useful feedback, all within the browser. As mentioned before, the embedded Prolog is a full system with all necessary `Ciao` bundles, manuals, etc, so that not just filtering but also the tool called or any user applications can be loaded and called to run locally on the browser, provided they are written in Prolog, or also of course JavaScript, compiled to wasm. Alternatively, calls can also be issued to tools available at some server or installed locally. Overall, the framework greatly facilitates the whole process, saving much coding.

An example application of HALDs. We will also present a concrete application in the generation of hybrid, interactive, web-based documentation, tutorials, courses, etc., for a *program verification tool*, `CiaoPP` [5, 12]. The aim is assisting students in learning how to use an advanced verification tool like `CiaoPP`, as well as learning more about Prolog. The HALD approach is a great aid here for adding solutions to exercises, examples, etc. automatically, while keeping the content synchronized with the system, greatly reducing the tracking of what documentation needs to be changed when an update to the system is made. `CiaoPP` [5, 12] performs several program debugging, analysis, and source-to-source transformation tasks for Prolog programs, and also for other high- and low- level languages. The output produced by `CiaoPP` generally contains significant amounts of information, including transformations, static analysis information, results of assertion checking or testing, counterexamples, etc. These results are typically presented as a new version of the source file annotated with (additional) assertions. The full analysis results produced by `CiaoPP` can be quite large, and cover the whole file or program. When writing the documentation it is interesting to show only a small fraction of this information at a time: the particular part that helps to understand the topic or step being explained. The filters are used here to extract, e.g., particular properties of a concrete predicate, particular types of assertions, etc. Analysis information can easily be embedded in human-readable explanations. A set of tutorials is available at `https://ciao-lang.org/ciao/build/doc/ciaopp_tutorials.html/` and we also refer again to Fig. 5 in the appendix.

The **source code** of all components of our system is available at `https://github.com/ciao-lang/`, including the build/install instructions. When the `Ciao` playground is installed, all the bundles and documentation will be run and hosted on your localhost server. Additional examples and instructions can be found in the `Ciao` playground manual [3] [5]. Although we have developed and described our work for concreteness in the context of the `Ciao` system, we believe the proposed mechanisms are of general

---

[5] `https://ciao-lang.org/ciao/build/doc/ciao_playground.html/`

applicability and readily adaptable to other systems.

References

[1] Anne Brecklinghaus & Philipp Koerner (2022): *A Jupyter Kernel for Prolog*. In: *Proc. 36th Workshop on (Constraint) Logic Lrogramming (WLP 2022)*, Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn.

[2] Peter Flach, Kacper Sokol & Jan Wielemaker (2023): *Simply Logical - The First Three Decades*. In David S. Warren, Veronica Dahl, Thomas Eiter, Manuel Hermenegildo, Robert Kowalski & Francesca Rossi, editors: *Prolog - The Next 50 Years*, *LNCS* 13900, Springer, doi:10.1007/978-3-031-35254-6_15.

[3] G. Garcia-Pradales, J.F. Morales & M. V. Hermenegildo (2021): *The Ciao Playground*. Technical Report, Technical University of Madrid (UPM) and IMDEA Software Institute. Available at `http://ciao-lang.org/ciao/build/doc/ciao_playground.html/ciao_playground_manual.html`.

[4] G. Garcia-Pradales, J.F. Morales, M. V. Hermenegildo, J. Arias & M. Carro (2022): *An s(CASP) In-Browser Playground based on Ciao Prolog*. In: *ICLP'22 Workshop on Goal-directed Execution of Answer Set Programs*.

[5] M. Hermenegildo, G. Puebla, F. Bueno & P. Lopez Garcia (2005): *Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)*. *Science of Computer Programming* 58(1–2), pp. 115–140, doi:10.1016/j.scico.2005.02.006.

[6] M. V. Hermenegildo (2000): *A Documentation Generator for (C)LP Systems*. In: *International Conference on Computational Logic, CL2000*, *LNAI* 1861, Springer-Verlag, pp. 1345–1361.

[7] M. V. Hermenegildo & J.F. Morales (2011): *The LPdoc Documentation Generator. Ref. Manual (V3.0)*. Technical Report, UPM. Available at `http://ciao-lang.org`.

[8] M.V. Hermenegildo, J.F. Morales & P. Lopez-Garcia (2023): *Some Thoughts on How to Teach Prolog*. In David S. Warren, Veronica Dahl, Thomas Eiter, Manuel Hermenegildo, Robert Kowalski & Francesca Rossi, editors: *Prolog - The Next 50 Years*, *LNCS* 13900, Springer, doi:10.1007/978-3-031-35254-6_9. Available at `http://cliplab.org/papers/TeachingProlog-PrologBook.pdf`.

[9] J. F. Morales, R. Haemmerlé, M. Carro & M. V. Hermenegildo (2012): *Lightweight compilation of (C)LP to JavaScript*. *Theory and Practice of Logic Programming, 28th Int'l. Conference on Logic Programming (ICLP'12) Special Issue* 12(4-5), pp. 755–773.

[10] J.F. Morales, S. Abreu, D. Ferreiro & M.V. Hermenegildo (2022): *Teaching Prolog with Active Logic Documents*. Technical Report, Technical University of Madrid (UPM) and IMDEA Software Institute.

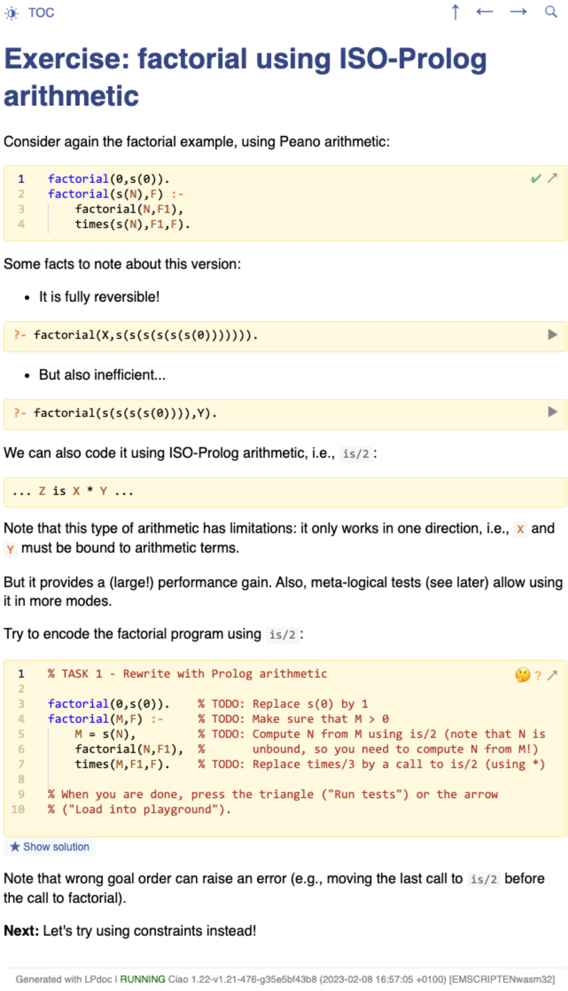
[11] J.F. Morales, Salvador Abreu, D. Ferreiro & M.V. Hermenegildo (2023): *Teaching Prolog with Active Logic Documents*. In David S. Warren, Veronica Dahl, Thomas Eiter, Manuel Hermenegildo, Robert Kowalski & Francesca Rossi, editors: *Prolog - The Next 50 Years*, *LNCS* 13900, Springer, doi:10.1007/978-3-031-35254-6_14. Available at `http://cliplab.org/papers/ActiveLogicDocuments-PrologBook.pdf`.

[12] G. Puebla, F. Bueno & M. V. Hermenegildo (2000): *Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs*. In: *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, *LNCS* 1817, Springer-Verlag, pp. 273–292, doi:10.1007/10720327_16.

[13] (2021): *τProlog — An open source Prolog interpreter in JavaScript*. `http://tau-prolog.org`. Last access: August 29, 2023.

[14] Jan Wielemaker, Fabrizio Riguzzi, Robert A. Kowalski, Torbjörn Lager, Fariba Sadri & Miguel Calejo (2019): *Using SWISH to Realize Interactive Web-based Tutorials for Logic-based Languages*. *Theory Pract. Log. Program.* 19(2), pp. 229–261, doi:10.1017/S1471068418000522.

Appendix A – ALD example



Exercise: factorial using ISO-Prolog arithmetic

Consider again the factorial example, using Peano arithmetic:

```
factorial(0,s(0)).
factorial(s(N),F) :-
    factorial(N,F1),
    times(s(N),F1,F).
```

Some facts to note about this version:

- It is fully reversible!

```
?- factorial(X,s(s(s(s(s(s(0))))))).
```

- But also inefficient...

```
?- factorial(s(s(s(s(0)))),Y).
```

We can also code it using ISO-Prolog arithmetic, i.e., `is/2`:

```
... Z is X * Y ...
```

Note that this type of arithmetic has limitations: it only works in one direction, i.e., `X` and `Y` must be bound to arithmetic terms.

But it provides a (large!) performance gain. Also, meta-logical tests (see later) allow using it in more modes.

Try to encode the factorial program using `is/2`:

```
% TASK 1 - Rewrite with Prolog arithmetic

factorial(0,s(0)).    % TODO: Replace s(0) by 1
factorial(M,F) :-     % TODO: Make sure that M > 0
    M = s(N),         % TODO: Compute N from M using is/2 (note that N is
    factorial(N,F1),  %       unbound, so you need to compute N from M!)
    times(M,F1,F).    % TODO: Replace times/3 by a call to is/2 (using *)

% When you are done, press the triangle ("Run tests") or the arrow
% ("Load into playground").
```

Note that wrong goal order can raise an error (e.g., moving the last call to `is/2` before the call to factorial).

Next: Let's try using constraints instead!

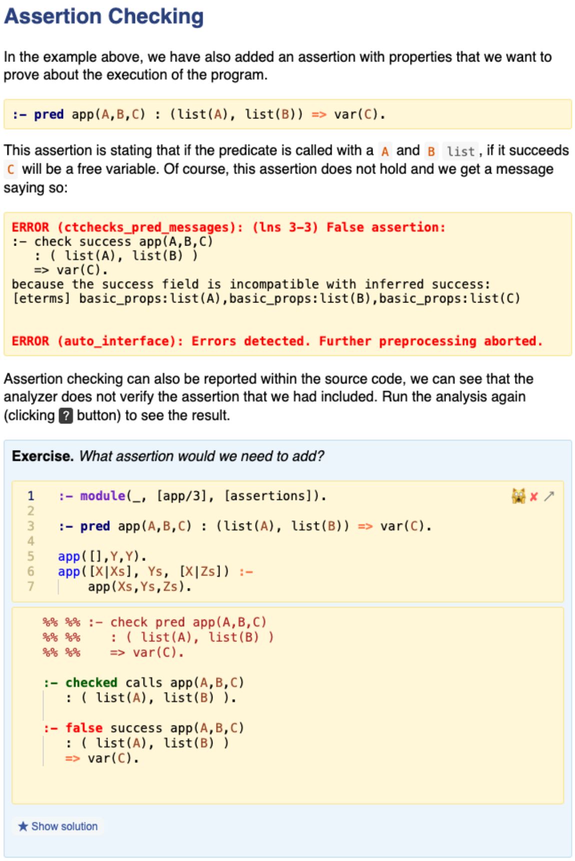
Generated with LPdoc I RUNNING Ciao 1.22-v1.21-476-g35e5bf43b8 (2023-02-08 16:57:05 +0100) [EMSCRIPTENwasm32]

```
\title Exercise: factorial using ISO-Prolog arithmetic

Consider again the factorial example, using Peano
arithmetic:
‘‘‘ciao\_runnable
:- module(_, _, [assertions,library(bf/bfall)]).
%! \begin{focus}
factorial(0,s(0)).
factorial(s(N),F) :-
    factorial(N,F1),
    times(s(N),F1,F).
%! \end{focus}

nat_num(0).
nat_num(s(X)) :- nat_num(X).

times(0,Y,0) :- nat_num(Y).
times(s(X),Y,Z) :- plus(W,Y,Z), times(X,Y,W).

plus(0,Y,Y) :- nat_num(Y).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
‘‘‘
Some facts to note about this version:
  - It is fully reversible!
‘‘‘ciao_runnable
?- factorial(X,s(s(s(s(s(s(0))))))).
‘‘‘
  - But also inefficient...
‘‘‘ciao_runnable
?- factorial(s(s(s(s(0)))),Y).
‘‘‘
We can also code it using ISO-Prolog arithmetic,
i.e., ‘is/2‘:
‘‘‘ciao
 ... Z is X * Y ...
‘‘‘
Note that this type of arithmetic has limitations:
it only works in one direction, i.e., ‘X‘ and ‘Y‘
must be bound to arithmetic terms.

But it provides a (large!) performance gain.  Also,
meta-logical tests (see later) allow using it in more
modes.

Try to encode the factorial program using ‘is/2‘:
‘‘‘ciao_runnable
:- module(_, _, [assertions]).
:- test factorial(5, B) => (B = 120) + (not_fails).
:- test factorial(0, 0) + fails.
:- test factorial(-1,B) + fails.
%! \begin{hint}
% TASK 1 - Rewrite with Prolog arithmetic

factorial(0,s(0)).    % TODO: Replace s(0) by 1
factorial(M,F) :-     % TODO: Make sure that M > 0
    M = s(N),         % TODO: Compute N from M using is/2
                      % (note that N is unbound,
    factorial(N,F1),  % so you need to compute N from M!)
    times(M,F1,F).    % TODO: Replace times/3 by a call
                      % to is/2 (using *)

% When you are done, press the triangle ("Run tests") or
% the arrow ("Load into playground").
%! \end{hint}
%! \begin{solution}
factorial(0,1).
factorial(N,F) :-
    N > 0,
    N1 is N-1,
    factorial(N1,F1),
    F is F1*N.
%! \end{solution}
‘‘‘
Note that wrong goal order can raise an error (e.g.,
moving the last call to ‘is/2‘ before the call to
factorial).
**Next:** Let's try using constraints instead!
```

Figure 4: The full source and LPdoc output for the Active Logic Document for a simple factorial exercise.

Appendix B – HALD example

`Output (exercise.html)`



Assertion Checking

In the example above, we have also added an assertion with properties that we want to prove about the execution of the program.

```
:- pred app(A,B,C) : (list(A), list(B)) => var(C).
```

This assertion is stating that if the predicate is called with a `A` and `B` `list`, if it succeeds `C` will be a free variable. Of course, this assertion does not hold and we get a message saying so:

```
ERROR (ctchecks_pred_messages): (lns 3-3) False assertion:
:- check success app(A,B,C)
   : ( list(A), list(B) )
   => var(C).
because the success field is incompatible with inferred success:
[eterms] basic_props:list(A),basic_props:list(B),basic_props:list(C)


ERROR (auto_interface): Errors detected. Further preprocessing aborted.
```

Assertion checking can also be reported within the source code, we can see that the analyzer does not verify the assertion that we had included. Run the analysis again (clicking ? button) to see the result.

Exercise. *What assertion would we need to add?*

```
:- module(_, [app/3], [assertions]).

:- pred app(A,B,C) : (list(A), list(B)) => var(C).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
```

```
%% %% :- check pred app(A,B,C)
%% %%    : ( list(A), list(B) )
%% %%    => var(C).

:- checked calls app(A,B,C)
   : ( list(A), list(B) ).

:- false success app(A,B,C)
   : ( list(A), list(B) )
   => var(C).
```

`Input (exercise.md)`

~~~
# Assertion Checking

In the example above, we have also added an assertion
with properties that we want to prove about the execution
of the program.
```ciao
:- pred app(A,B,C) : (list(A), list(B)) => var(C).
```
This assertion is stating that if the predicate is called
with a A and B list, if it succeeds C will be a free
variable. Of course, this assertion does not hold and we
get a message saying so:

@exfilter{app_assrt_false.pl}{V,filter=warn_error}

Assertion checking can also be reported within the source
code, we can see that the analyzer does not verify the
assertion that we had included. Run the analysis again
(clicking ? button) to see the result.

Exercise. What assertion would we need to add?
```ciao_runnable
%! \begin{code}
:- module(_, [app/3], [assertions]).

:- pred app(A,B,C) : (list(A), list(B)) => var(C).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
%! \end{code}
%! \begin{opts}
solution=verify_assert
%! \end{opts}
%! \begin{solution}
:- module(_, [app/3], [assertions]).

:- pred app(A,B,C) : (list(A), list(B)) => list(C).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
%! \end{solution}
```
~~~

Figure 5: The full source and `LPdoc` output for a Hybrid Active Logic Document containing a simple assertion checking exercise. Note the use of a filtering command in the source to call `CiaoPP` during the static phase and select certain outputs that are incorporated in the page (the box with the error messages). Also, an interactive exercise is embedded and filtering is used again (this time dynamically) to present a selected part of the analysis output for the program entered by the student and to compare it to the expected results.

Anthem-P2P: Automatically Verifying the Equivalent External Behavior of ASP Programs

Zachary Hansen
University of Nebraska Omaha (UNO)
Nebraska, USA
zachhansen@unomaha.edu

Answer Set Programming (ASP) enables the solving of challenging search and optimization problems with concise, declarative programs. However, the most human-readable encoding does not always coincide with the most efficient encoding from the standpoint of grounding and solving. In the process of making a program more efficient the programmer may substantially refactor the initial program. The system presented here can automatically provide guarantees that the new versions adhere to the original "specification" version of the program. While refactoring is the primary intended use of this system, it can in general be used to confirm that certain pairs of ASP programs share the same external behavior under a given set of assumptions.

Introduction The tool presented here is designed to support the development of correct logic programs under the stable model semantics. For a broad class of *tight* programs, `anthem-p2p` can automatically provide a formal guarantee that a pair of programs have the same *external behavior*. A precise definition of this concept may be found in [3]. Broadly speaking, two logic programs have equivalent external behavior if the extents of their output/shown predicates coincide for any inputs that conform to the intended use of the program.

The `anthem-p2p` system assists in the process of refactoring (modifying a program while preserving its external behavior), which is an important strategy in the construction of provably correct programs. One of the envisioned uses of this system is the following methodology. First, a highly declarative, human-readable program is shown to be correct via a meta-level proof (see, for instance [1, 2]). Next, subsequent refactorings improving the program's efficiency at the expense of clarity can be proven to be equivalent via object-level proofs constructed by `anthem-p2p`. The remainder of this paper presents the implementation details behind `anthem-p2p`, the theoretical foundations and key concepts of which are studied in [3].

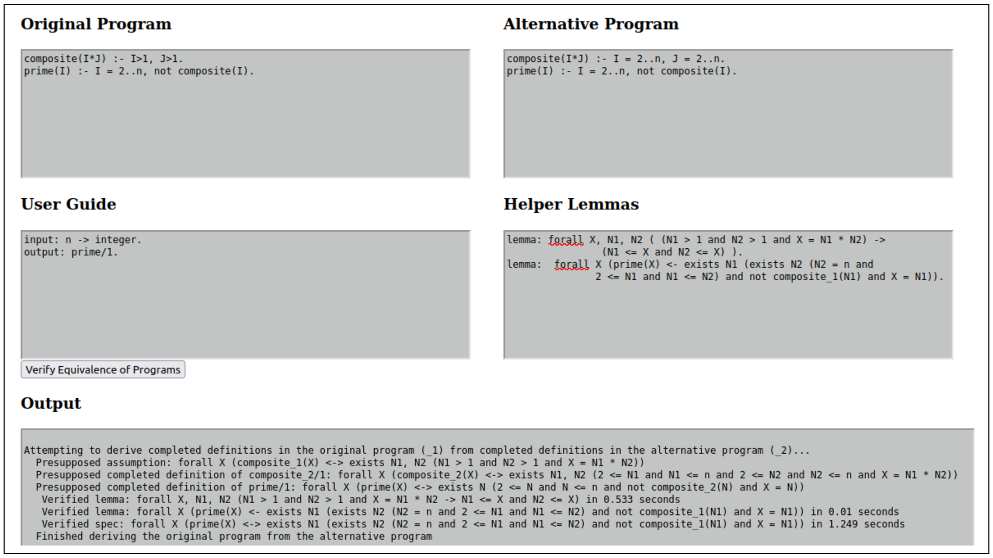
Using Anthem-P2P To get started with program-to-program verification, readers may wish to first visit the AP2P website[1]. This website allows users to experiment with the `anthem-p2p` system[2] without installing its software sub-components such as `vampire` [5] and `anthem` [4] on their personal computers. Figure 1 shows the text editing portion of the AP2P homepage, which is populated by default with an example showcasing equivalent prime numbers programs. Users may enter a time limit for the proof search process (one direction of which derives the original program from the alternative, the other direction derives the alternative from the original). The "Verify Equivalence of Programs" button invokes the `anthem-p2p` system on a UNO server (Oracle Linux 8, 4 Intel(R) Xeon(R) Gold 6248 CPUs, 4 GB RAM) and displays the result.

---

[1] Available at `https://ap2p.unomaha.edu/`.

[2] The tool is also available for download at `https://github.com/ZachJHansen/anthem-p2p`.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 330–332, doi:10.4204/EPTCS.385.34



```
composite(I*J) :- I>1, J>1.
prime(I) :- I = 2..n, not composite(I).
```

```
composite(I*J) :- I = 2..n, J = 2..n.
prime(I) :- I = 2..n, not composite(I).
```

```
input: n -> integer.
output: prime/1.
```

```
lemma: forall X, N1, N2 ( (N1 > 1 and N2 > 1 and X = N1 * N2) ->
                (N1 <= X and N2 <= X) ).
lemma:  forall X (prime(X) <- exists N1 (exists N2 (N2 = n and
               2 <= N1 and N1 <= N2) and not composite_1(N1) and X = N1)).
```

```
Attempting to derive completed definitions in the original program (_1) from completed definitions in the alternative program (_2)...
  Presupposed assumption: forall X (composite_1(X) <-> exists N1, N2 (N1 > 1 and N2 > 1 and X = N1 * N2))
  Presupposed completed definition of composite_2/1: forall X (composite_2(X) <-> exists N1, N2 (2 <= N1 and N1 <= n and 2 <= N2 and N2 <= n and X = N1 * N2))
  Presupposed completed definition of prime/1: forall X (prime(X) <-> exists N (2 <= N and N <= n and not composite_2(N) and X = N))
   Verified lemma: forall X, N1, N2 (N1 > 1 and N2 > 1 and X = N1 * N2 -> N1 <= X and N2 <= X) in 0.533 seconds
   Verified lemma: forall X (prime(X) <- exists N1 (exists N2 (N2 = n and 2 <= N1 and N1 <= N2) and not composite_1(N1) and X = N1)) in 0.01 seconds
   Verified spec: forall X (prime(X) <-> exists N1 (exists N2 (N2 = n and 2 <= N1 and N1 <= N2) and not composite_1(N1) and X = N1)) in 1.249 seconds
  Finished deriving the original program from the alternative program
```

Figure 1: AP2P Online Interface: Verifying the prime numbers example.

The GitHub repository contains a number of examples, which may be tested using variations of the following command:

```
python3 anthem-p2p <program file> <program file> <user guide file>
```

The preceding command may be optionally extended with the following arguments:

```
-l <lemmas file> -t <time limit (seconds)> -c <number of cores>
```

System Design The system design of `anthem-p2p` is summarized in Figure 2. Given two logic programs written in the mini-GRINGO [4, Section 2] subset of the input language of `clingo`, and a user guide written in a custom specification language [4, Section 5], `anthem-p2p` will verify that the programs have equivalent external behavior under the assumptions of the user guide. This is accomplished by transforming the problem into an input (consisting of a logic program and a specification) to the system `anthem` [4]. This transformation is justified by the theory presented in [3]. Similarly, `anthem` transforms the task of proving that a logic program implements a specification into a theorem-proving task for `vampire` [5]. The proof search can often be accelerated by providing helper lemmas (optional) that direct the search. We now describe how the example shown in Figure 1 is processed by the system depicted in Figure 2.

The first task of `anthem-p2p` is to identify the public (input and output) predicates occurring in *User Guide* (in this case just *prime/1*). Any predicates occurring in *Original* (*Alternative*) that are not public are considered private to that program. The pre-processing step creates new programs where private predicate constants are replaced with fresh predicate constants obtained by appending an "_1" to constants from *Original* and an "_2" to constants from *Alternative*. We will call the lists of such new predicates **p** and **q**, respectively. In this example, $\mathbf{p} = \{composite\_1/1\}$ and $\mathbf{q} = \{composite\_2/1\}$. Next, `anthem` is used to generate the formula representations of rules from *New Original*. Specifically, we obtain the *completed definitions* [4] of predicates from **p** (call this list of formulas **a**), and the completed definitions of public predicates from *New Original* (call this list of formulas **s**). To **s** we also add the formula representations of constraints from *New Original*. *Final specification* is generated by extending the user

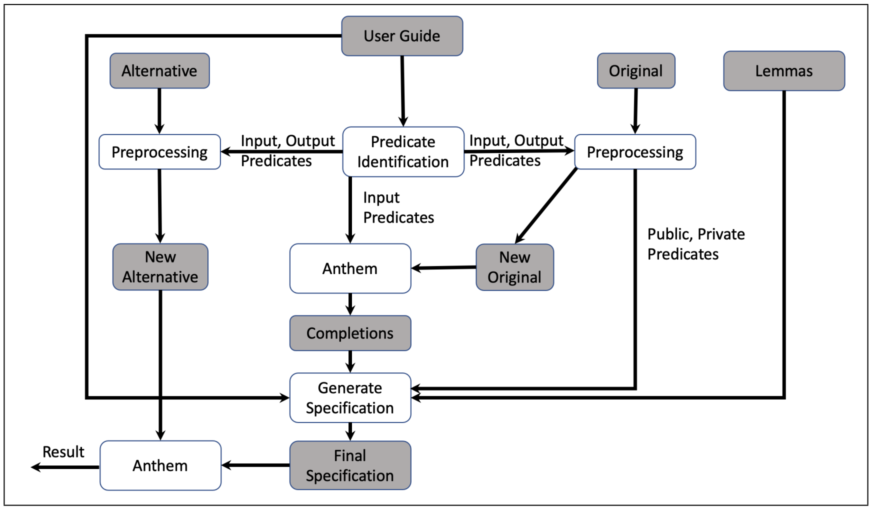


Figure 2: Anthem-P2P System Design: Verifying the equivalence of *Original* and *Alternative* programs.

guide with predicate symbols from **p** as input symbols, formulas from **a** as assumptions, and formulas from **s** as specs (conjectures that must be proven). The `anthem` system is invoked a second time to verify that *New Alternative* implements *Final Specification*, which is equivalent to proving that *Original* and *Alternative* have the same external behavior under the assumptions of *User Guide*.

Acknowledgements

Thanks go to Yuliya Lierler and Vladimir Lifschitz for their feedback on this draft, and on this system as it was being developed. Thanks also to Nathan Temple for reporting the bugs he encountered while using early versions, and the College of IS&T for sponsoring the AP2P server.

References

[1] Pedro Cabalar, Jorge Fandinno & Yuliya Lierler (2020): *Modular Answer Set Programming as a Formal Specification Language*. *Theory and Practice of Logic Programming*, doi:10.1017/S1471068420000265.

[2] Jorge Fandinno, Zachary Hansen & Yuliya Lierler (2022): *Arguing Correctness of ASP Programs with Aggregates*. In Georg Gottlob, Daniela Inclezan & Marco Maratea, editors: *Logic Programming and Nonmonotonic Reasoning*, Lecture Notes in Computer Science, Springer International Publishing, Cham, p. 190–202, doi:10.1007/978-3-031-15707-3_15.

[3] Jorge Fandinno, Zachary Hansen, Yuliya Lierler, Vladimir Lifschitz & Nathan Temple (2023): *External Behavior of a Logic Program and Verification of Refactoring*. *Theory and Practice of Logic Programming*, doi:10.1017/S1471068423000200.

[4] Jorge Fandinno, Vladimir Lifschitz, Patrick Lühne & Torsten Schaub (2020): *Verifying Tight Logic Programs with anthem and vampire*. *Theory and Practice of Logic Programming* 20(5), p. 735–750, doi:10.1017/S1471068420000344.

[5] Laura Kovács & Andrei Voronkov (2013): *First-Order Theorem Proving and Vampire*. In Natasha Sharygina & Helmut Veith, editors: *Computer Aided Verification*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, p. 1–35, doi:10.1007/978-3-642-39799-8_1.

Nemo: First Glimpse of a New Rule Engine

Alex Ivliev[1] Stefan Ellmauthaler[1] Lukas Gerlach[1] Maximilian Marx[1]
Matthias Meißner Simon Meusel Markus Krötzsch[1]
Knowledge-Based Systems Group, Faculty of Computer Science / cfaed / CeTI / ScaDS.AI
TU Dresden, Germany
[1] `{firstname.lastname}@tu-dresden.de`

This system demonstration presents *Nemo*, a new logic programming engine with a focus on reliability and performance. Nemo is built for data-centric analytic computations, modelled in a fully declarative Datalog dialect. Its scalability for these tasks matches or exceeds that of leading Datalog systems. We demonstrate uses in reasoning with knowledge graphs and ontologies with $10^5 – 10^8$ input facts, all on a laptop. Nemo is written in Rust and available as a free and open source tool.

From the early days of logic programming, it has been clear that declarative rules can also be useful for data analysis and query answering. *Datalog* can either be viewed as the core of virtually every logic programming language, or as the generalisation of conjunctive queries with recursion [2]. This bridge between rule-based systems and databases has become even more important recently, since it fits well with the growing demand for data analytics, graph-based data management, and declarative processing.

Accordingly, there is a great number of Datalog-based rule engines, with widely different goals and features. The following overview of relevant system types is far from complete:

1. logic programming systems, esp. for ASP [8, 3] and Prolog [10],
2. knowledge graph and deductive database engines like RDFox [11], VLog [12], and Vadalog [5],
3. specialised data-analytics systems like Soufflé [9], LogicBlox [4], or EmptyHeaded [1], and
4. data management frameworks such as Datomic, Google Logica, and CozoDB.

Our new system *Nemo* is most closely related to tools of type (2) and (3). From tools of type (2), it inherits the focus on scalability (especially with regards to data size), compatibility with open data standards like RDF, and its support for *existential rules* (a.k.a. tuple-generating dependencies), which are important in databases and rule-based ontologies. At the same time, it aims to support a broader range of datatypes, built-in operations, and aggregates, as are typically used in data analytics tools of type (3). A commonality shared with most tools above (except those of type (4)) is that Nemo runs in memory, without a persistent database backend (though using input data from such databases is planned).

Nemo is still at an early stage of development, but already able to solve real and synthetic benchmarking tasks at speeds that can compete with other tools mentioned above. This system demonstration offers a first glimpse at the current functionalities and planned upcoming features. Nemo is developed in Rust. Its source code, releases, and documentation are at `https://github.com/knowsys/nemo/`. A live demo of Nemo can be tried online at `https://tools.iccl.inf.tu-dresden.de/nemo/`.

Supported Datalog Dialect Nemo works on a custom Datalog dialect that modifies common notation from logic programming to accommodate the more flexible and accurate data model from the W3C RDF and SPARQL standards. The syntax is widely compatible with that of Rulewerk [7, formerly *VLog4j*], and with the Datalog fragment of RDFox [11]. An example is shown in Figure 1.

The program in Figure 1 integrates two data sources – public `trees` in Dresden and `taxon`omic information about plant species from Wikidata – to find old lime (linden) trees, i.e., trees of any subspecies of genus *Tilia*. The datasets are loaded in `@source` directives, where the comments note the

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 333–335, doi:10.4204/EPTCS.385.35

```
@declare tree(any,any,integer,integer) .
@source tree[4]: load-csv("dresden-trees.csv") . % location,species,age,height
@source taxon[3]: load-csv("wikidata-taxons.csv.gz") . % taxon,label,supertaxon

lime(?id, "Tilia") :- taxon(?id, "Tilia", ?parentId) .
lime(?id, ?name)   :- taxon(?id, ?name, ?parentId), lime(?parentId, ?parentName) .
oldLime(?loc,?species,?age) :- tree(?loc,?species,?age,?height), ?age>200, lime(?id,?species) .
```

Figure 1: Finding Dresden's oldest lime (linden) trees in Nemo

	Doctors-1M	**Ontology-256**	**LUBM-01k**	**Deep200**	**Galen EL**	**SNOMED CT**
Inferred facts	792,500	5,674,201	186,742,694	725,457	1,858,810	24,117,991
Nemo (sec)	3.2	13.4	163.3	5.1	3.6	62.1
VLog (sec)	2.5	22.2	199.4	*timeout*	45.2	*oom*

Table 1: Selected benchmarking results (loading+reasoning); *timeout*: 60min; *oom*: out of memory

meaning of the parameters. For `tree`, we `@declare` specific datatypes, where *any* is the most general type that supports all data (the default if no declaration is given), whereas *integer* loads numeric values. The first two rules find all species of lime tree by recursively collecting all taxons below the genus *Tilia* in the tree of life. The third rule then finds trees of some such species and an age of over 200 years. Finding Dresden's seven old limes (the oldest a small-leaved lime of 337 years) from the >88,000 city trees with known age and >3.6M taxons takes about 7 sec on a laptop, of which 200 msec are used to apply rules (the rest is for data loading). The example data and program is available online at `https://github.com/knowsys/nemo-examples` in directory examples/lime-trees.

In addition to the features illustrated above, Nemo also supports stratified negation (denoted ~), conjunctions in rule heads (denoted `,`), further datatypes and built-ins (esp. floating point numbers), and existentially quantified head variables (using `!` instead of `?` in front of the variable name).

System Overview The underlying reasoning procedure is based on materialisation (forward chaining of rules) using semi-naive evaluation [2] and the restricted chase [6]. Key to overall performance is a combination of columnar data structures (introduced for Datalog by Urbani et al. [12]), a multiway join algorithm based on *leapfrog triejoin* [13], and own new optimisation techniques based on careful computation planning. The columnar design also allows for efficient support for values of different types at the lowest level. The system aims at maximal declarativity and syntax-independent performance (e.g., the order or parameters in predicates or the order of atoms in rules has no effect on performance).

As a system, Nemo can be invoked through a command-line client `nmo`, which includes options for storing results. Various input and output formats are supported, currently CSV and TSV, RDF, and logic programming facts. We strive to support both the elaborate type system and data representation forms of RDF, and the more basic data schemes often found in CSV or classical logic programming, without a burden on the user. Nemo is implemented in Rust and can also be used as a Rust library (a *crate*).

Experiments We compare runtimes (loading and reasoning) of Nemo (v0.2.0) and VLog (v1.3.6) on established benchmarks and real-world tasks. Times were measured on a notebook (Dell XPS 13; Ubuntu Linux 22.04; Intel i7-1165G7@2.80GHz; 16GB RAM; 512 GB SSD). Table 1 presents an overview of the results obtained. The first four result columns are existential rule benchmarks from ChaseBench [6]; the final two columns used an unoptimised encoding of a well-known OWL EL reasoning calculus on two different ontologies. Nemo matched or outperformed VLog in all experiments, with notable advantages on hard cases (Deep200 is a synthetic stress test; SNOMED is one of the largest real-world ontologies). Full details are at `https://github.com/knowsys/nemo-examples/` under evaluations/iclp2023.

Outlook Nemo is still in its early stages, and many additional features are under development. They include further datatypes, built-in functions, and (stratified) aggregates; support for structured data (functional terms, sets, frames, etc.); and interface improvements (programming APIs, extended client functionality). Moreover, we are researching new optimisation and explanation approaches for rule reasoning.

Acknowledgements This work was supported in DFG grant 389792660 (TRR 248), by BMBF in grants ITEA-01IS21084 (InnoSale) and 13GW0552B (KIMEDS), and in DAAD grant 57616814 (SECAI).

References

[1] Christopher R. Aberger, Susan Tu, Kunle Olukotun & Christopher Ré (2016): *EmptyHeaded: A Relational Engine for Graph Processing*. In Fatma Özcan, Georgia Koutrika & Sam Madden, editors: *Proc. 2016 ACM SIGMOD Int. Conf. on Management of Data*, ACM, pp. 431–446, doi:10.1145/3129246.

[2] Serge Abiteboul, Richard Hull & Victor Vianu (1994): *Foundations of Databases*. Addison Wesley.

[3] Mario Alviano, Francesco Calimeri, Carmine Dodaro, Davide Fuscà, Nicola Leone, Simona Perri, Francesco Ricca, Pierfrancesco Veltri & Jessica Zangari (2017): *The ASP System DLV2*. In Marcello Balduccini & Tomi Janhunen, editors: *Proc. 14th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'17)*, *LNCS* 10377, Springer, pp. 215–221, doi:10.1007/978-3-319-61660-5_19.

[4] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen & Geoffrey Washburn (2015): *Design and Implementation of the LogicBlox System*. In T.K. Sellis, S.B. Davidson & Z.G. Ives, editors: *Proc. 2015 ACM SIGMOD Int. Conf. on Mngmt of Data*, pp. 1371–1382, doi:10.1145/2723372.2742796.

[5] Luigi Bellomarini, Emanuel Sallinger & Georg Gottlob (2018): *The Vadalog System: Datalog-based Reasoning for Knowledge Graphs*. *Proc. VLDB Endowment* 11(9), pp. 975–987, doi:10.14778/3213880.3213888.

[6] Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro & Efthymia Tsamoura (2017): *Benchmarking the Chase*. In: *Proc. 36th Symp. on Principles of Database Systems (PODS'17)*, ACM, pp. 37–52, doi:10.1145/3034786.3034796.

[7] David Carral, Irina Dragoste, Larry González, Ceriel Jacobs, Markus Krötzsch & Jacopo Urbani (2019): *VLog: A Rule Engine for Knowledge Graphs*. In Chiara Ghidini et al., editor: *Proc. 18th Int. Semantic Web Conf. (ISWC'19, Part II)*, *LNCS* 11779, Springer, pp. 19–35, doi:10.1007/978-3-030-30796-7_2.

[8] Martin Gebser, Benjamin Kaufmann & Torsten Schaub (2012): *Conflict-driven answer set solving: From theory to practice*. *Artif. Intell.* 187, pp. 52–89, doi:10.1016/j.artint.2012.04.001.

[9] Herbert Jordan, Bernhard Scholz & Pavle Subotic (2016): *Soufflé: On Synthesis of Program Analyzers*. In Swarat Chaudhuri & Azadeh Farzan, editors: *Proc. 28th Int. Conf. on Computer Aided Verification (CAV'16), Part II*, *LNCS* 9780, Springer, pp. 422–430, doi:10.1007/978-3-319-41540-6_23.

[10] Philipp Körner, Michael Leuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V. Hermenegildo, José F. Morales, Jan Wielemaker, Daniel Diaz & Salvador Abreu (2022): *Fifty Years of Prolog and Beyond*. *Theory Pract. Log. Program.* 22(6), pp. 776–858, doi:10.1017/S1471068422000102.

[11] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu & Jay Banerjee (2015): *RDFox: A Highly-Scalable RDF Store*. In Marcelo Arenas et al., editor: *Proc. 14th Int. Semantic Web Conf. (ISWC'15), Part II*, *LNCS* 9367, Springer, pp. 3–20, doi:10.1007/978-3-319-25010-6_1.

[12] Jacopo Urbani, Ceriel Jacobs & Markus Krötzsch (2016): *Column-Oriented Datalog Materialization for Large Knowledge Graphs*. In Dale Schuurmans & Michael P. Wellman, editors: *Proc. 30th AAAI Conf. on Artificial Intelligence (AAAI'16)*, AAAI Press, pp. 258–264, doi:10.1609/aaai.v30i1.9993.

[13] Todd L. Veldhuizen (2014): *Triejoin: A Simple, Worst-Case Optimal Join Algorithm*. In N. Schweikardt, V. Christophides & V. Leroy, editors: *Proc. 17th Int. Conf. on Database Theory (ICDT'14)*, pp. 96–106, doi:10.5441/002/icdt.2014.13.

PySpArX - A Python Library for Generating Sparse Argumentative Explanations for Neural Networks

Ioana Mihailescu*† icm20@ic.ac.uk
Alan Weng*† aw920@ic.ac.uk
Siddharth Sharma*† ss2120@ic.ac.uk
Mihnea Ghitu*† mg2720@ic.ac.uk
Dilshaan Grewal*† dsg20@ic.ac.uk
Khemi Chew*† kjc20@ic.ac.uk
Hamed Ayoobi‡† h.ayoobi@ic.ac.uk
Nico Potyka‡† n.potyka@ic.ac.uk
Francesca Toni‡† f.toni@ic.ac.uk

1 Introduction

The increasing use of data-centric techniques in autonomous intelligent systems in areas such as healthcare, automotive, and finance raises concerns about their predictions' reliability, fairness, and transparency. In the case of systems empowered by black-box machine learning models, the user has no access to the inner mechanics of the systems, thus making it essential to obtain a human-readable interface providing explanations of the models' operations.

Explainable Artificial Intelligence (XAI) techniques aim to provide user-friendly explanations for models. SpArX (Sparse Argumentative Explanations for Neural Networks [2]) is an XAI method that faithfully explains the mechanics of neural networks in the form of Multi-Layer Perceptrons (MLPs) by exploiting the correspondence between MLPs and Quantitative Argumentation Frameworks (QAF) [11]. Intuitively, QAFs are a form of argumentation framework amounting to weighted graphs that hold arguments as their nodes and relations (of attack or support) between arguments as their edges. To provide an explainable interface of MLPs using QAFs, the MLPs are first sparsified, through a per-layer clustering of neurons based on output-similarity. The sparsified MLPs are then transformed into QAFs and then global (input-independent) and local (within the vicinity of a target input) explanations are generated, while faithfully preserving the mechanics of the original MLPs.

Building upon SpArX, we have provided users with tools that help them prove and understand how MLPs, trained on specific datasets, work. The tools allow users to customize explanations to their needs.

2 System Overview

The architecture comprises of three primary components, namely a Python library, a GUI for generating models, and a web-based application for visualization. We have provided technical users with a Python Pip package [7] in which a modular implementation of SpArX can be found, as well as a GUI [5] in which

---

*Equal contributors.

†Imperial College London, London, United Kingdom

‡Supervisors.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 336–338, doi:10.4204/EPTCS.385.36

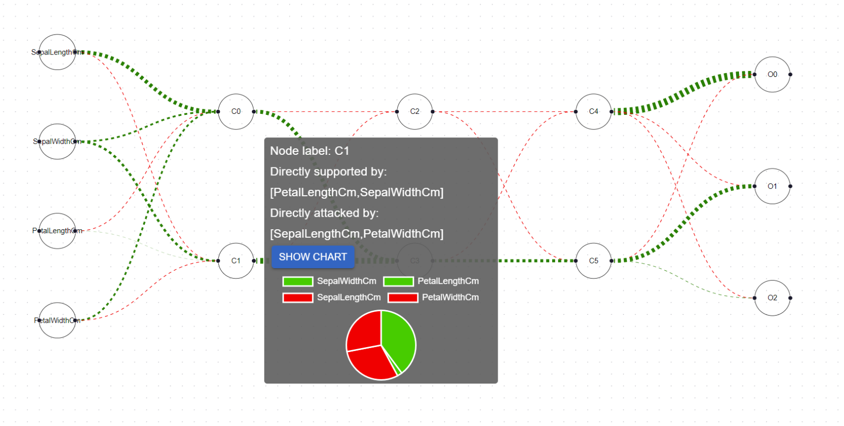
they can input their own dataset, and generate a customizable MLP. This is forwarded to our framework pipeline, which performs the aforementioned transformations (sparsification and mapping onto a QAF). The final QAF is then stored in a database and made available online to other users via a shareable link [10] where it can be visualised.

2.1 sparx-lib

The Python Pip package, sparx-lib [7], is a custom library developed to handle the heavy computations required for generating SpArX explanations. It is designed with a modular architecture that allows for the flexible integration of different components of the SpArX algorithm. It enables users to generate explanations for a variety of models, which can be either user-specified, with a dataset, or imported. Tensorflow and Keras are utilized to develop and train custom models. Users can generate two types of visualisations: pure JSON, which can be viewed as text or parsed, or a GUI using NetworkX and Bokeh. We used a documentation generation tool called Sphinx to create documentation for the sparx-lib pip package [6].

2.2 Model generation GUI

The model generation GUI [5] is a tool developed using the React framework. It provides a platform for administrators or technical users to design MLPs for a selected dataset, generate explanations, and store visualizations in a remote database for access by other users. The model generation process is currently performed locally due to the computational demands involved in training the model and calculating explanations. The requirements and instructions for its execution are provided in the **SpArX-GUI repository** [5].



Node label: C1

Directly supported by:

[PetalLengthCm,SepalWidthCm]

Directly attacked by:

[SepalLengthCm,PetalWidthCm]

Figure 1: Web app visualisation [10] for the Iris dataset [3]

2.3 Visualisation

The customizable visualization component [9] provides features such as:

- displaying node and edge information such as attackers/supporters and their weight using hover-cards, and visualising node dependencies using charts;
- filtering graph information by progressively rendering layers and restricting the nodes shown;

- sharing generated visualisations via a link/QR code.

We have utilised React with React Flow to create interactive graphs that can be easily modified (See Figure 1). Our server has been deployed on the web to enhance accessibility to the app. Administrators can simply share a link to the dedicated visualisation page with other non-technical users.

3 Demo requirements

To run the project, follow the instructions in the README file from the SpArX-GUI repository [5]. The model generation GUI accepts datasets in CSV format where the first line represents the name of the parameter/column. A test dataset is provided in the SpArX-GUI repository under `data/iris.data`. For demo purposes, an already generated visualisation of a SpArX explanation can be found online [10]. A demonstration video can be found at [4].

Acknowledgments

This research was partially funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101020934, ADIX) and by J.P. Morgan and by the Royal Academy of Engineering under the Research Chairs and Senior Research Fellowships scheme. Any views or opinions expressed herein are solely those of the authors.

References

[1] H. Ayoobi, N. Potyka & F. Toni (2022): *SpArX legacy code*. Available at `https://github.com/H-Ayoobi/SpArX`.

[2] H. Ayoobi, N. Potyka & F. Toni (2023): *SpArX: Sparse Argumentative Explanations for Neural Networks*, doi:10.48550/ARXIV.2301.09559. Available at `https://arxiv.org/abs/2301.09559`.

[3] D. Dua & C." Graff (2017): *UCI Machine Learning Repository*. Available at `http://archive.ics.uci.edu/ml`.

[4] I. Mihailescu (2023): *PySpArX - A library for SpArX explanations of Neural Networks - Demo*. Available at `https://youtu.be/0JHl4zegFvU`.

[5] I. Mihailescu, K. Chew, M. Ghitu, S. Sharma, D. Grewal & A. Weng (2022): *SpArX-GUI*. Available at `https://github.com/SpArX-Group-10/SpArX-GUI`.

[6] I. Mihailescu, K. Chew, M. Ghitu, S. Sharma, D. Grewal & A. Weng (2022): *Sparx-lib documentation*. Available at `https://sparx-group-10.github.io/SpArX/`.

[7] I. Mihailescu, K. Chew, M. Ghitu, S. Sharma, D. Grewal & A. Weng (2022): *Sparx-lib pip package*. Available at `https://pypi.org/project/sparx-lib/`.

[8] I. Mihailescu, K. Chew, M. Ghitu, S. Sharma, D. Grewal & A. Weng (2022): *SpArX-lib Repository*. Available at `https://github.com/SpArX-Group-10/SpArX`.

[9] I. Mihailescu, K. Chew, M. Ghitu, S. Sharma, D. Grewal & A. Weng (2022): *SpArX-Visualisation*. Available at `https://github.com/SpArX-Group-10/SpArX-Visualisation`.

[10] I. Mihailescu, K. Chew, M. Ghitu, S. Sharma, D. Grewal & A. Weng (2022): *SpArX-Visualisation Web app example*. Available at `https://sparx-vis.herokuapp.com/63ac3bd37963d9af247d8102`.

[11] N. Potyka (2021): *Interpreting Neural Networks as Quantitative Argumentation Frameworks*. In: *AAAI Conference on Artificial Intelligence, (AAAI)*, AAAI Press, pp. 6463–6470, doi:10.1609/aaai.v35i7.16801.

Deep Inductive Logic Programming meets Reinforcement Learning

Andreas Bueff
University of Edinburgh

Vaishak Belle
University of Edinburgh

One approach to explaining the hierarchical levels of understanding within a machine learning model is the symbolic method of inductive logic programming (ILP), which is data efficient and capable of learning first-order logic rules that can entail data behaviour. A differentiable extension to ILP, so-called differentiable Neural Logic (dNL) networks, are able to learn Boolean functions as their neural architecture includes symbolic reasoning. We propose an application of dNL in the field of Relational Reinforcement Learning (RRL) to address dynamic continuous environments. This represents an extension of previous work in applying dNL-based ILP in RRL settings, as our proposed model updates the architecture to enable it to solve problems in continuous RL environments. The goal of this research is to improve upon current ILP methods for use in RRL by incorporating non-linear continuous predicates, allowing RRL agents to reason and make decisions in dynamic and continuous environments.

1 Introduction

One approach to explaining the hierarchical levels of understanding within a machine learning model is the symbolic method of inductive logic programming (ILP) [15], which is data efficient and capable of learning first-order logic rules that can entail data behaviour. Recent contributions to the field have expanded the ILP framework to allow for end-to-end learning, resulting in hybrid models that can be classified as neuro-symbolic [6, 16, 22, 3]. The recent developments in neuro-symbolic ILP have expanded the potential applications of these models to a wider range of learning challenges, including reinforcement learning [17]. The ILP-based neuro-symbolic model implemented in this proposal is a differentiable extension to ILP, so-called differentiable Neural Logic (dNL) networks [16]. dNL networks are able to learn Boolean functions as their neural architecture includes symbolic reasoning. The primary neural layers in the dNL model contain weighted neurons associated with conjunctions as well as weighted activation neurons associated with disjunctions, providing a means of logical reasoning on the input as well as a means of optimisation via gradient descent.

Relational RL (RRL) is concerned with learning policies for decision-making tasks in complex, discrete relational environments [4]. In RRL, the environment is characterised by a set of entities and relationships between them, and the agent's actions can affect the state of these entities and relationships. Due to its focus on relational problems, RRL has leveraged the progress made in ILP but is not limited to one method. Recent efforts have enabled RRL to effectively tackle more complex RL problems by transitioning from purely symbolic reasoning to a more neuro-symbolic approach that leverages neural systems. Earlier research in RRL primarily focused on planning [4], emphasising model-based learning. Later research, however, focused on model-free methods that combined neuro-symbolic models and RRL [23] or on the derivation of interpretable FOL policies [10, 17]. Despite these advancements, there is limited research in applying RRL to more complex learning challenges, such as continuous state dynamics. This gap in knowledge motivates the proposed research.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 339–352, doi:10.4204/EPTCS.385.37

The work by Payani et al. extended RRL to learn FOL policies using a dNL agent [17]. Taking the concepts of dNL, Payani et al. combined their dNL-ILP framework with RRL [17]. The author tested on the block world gaming environment and took advantage of the declarative bias with provided background knowledge. The authors expanded RRL to handle complex scene interpretations and used their dNL-ILP differentiable inductive engine to give RRL an end-to-end learning framework, so-called dNL-RRL, where the authors focused on policy gradients in order to improve interpretability and expert constraints to improve the speed of convergence.

The following paper is an extension that incorporates both a continuous and non-linear interpretation of [16] and the dNL-RRL agent found in [17] resulting in a dNL-ILP based agent that can both learn in dynamic continuous environments typically seen in RL control problems and extract FOL rules which define the agent policy. Various RL algorithms were explored in evaluating our dNL-ILP agent and ultimately Soft Actor-Critic was found to perform the best when solving problems on continuous state spaces. As the contribution bridges the application of dNL-ILP agents in discrete state spaces with that of continuous, we evaluated the model on a dynamic RL environment where the optimal policy captured rules such as those seen with classical mechanics in physics. In our evaluation, the model was applied to two control problems including the Cart Pole problem and Lunar Lander problem [1]. From our initial results, we were able to obtain agent policies which incorporate continuous predicate functions as well as non-linear continuous predicate functions. We also add that our proposed agent, while primarily focused on solving classical RL problems, is referred to as an RRL agent due to its use of relational language in the derived FOL rules and the ability to incorporate background knowledge through the use of non-linear predicate functions.

2 Background

2.1 Reinforcement Learning

Reinforcement learning (RL) aims to find an optimal action sequence for an agent to maximize its expected future reward. This is typically done by modelling the environment as a Markov Decision Process (MDP), which is defined as a tuple $\mathscr{M} = \langle \mathscr{S}, \mathscr{A}, \mathscr{R}, \mathscr{T}, \mathscr{E} \rangle$. $\mathscr{S}$ is the set of states, $\mathscr{A}$ is the set of actions that can be taken, $\mathscr{R}(s_t, a_t, s_{t+1})$ is the reward function that takes the current state s_t, current action a_t, and returns the reward from transitioning to the state s_{t+1}, $\mathscr{T}(s_t, a_t, s_{t+1})$ is the transition probability function which represents the probability of transitioning to state s_{t+1} from state s_t given action a_t was taken, and $\mathscr{E} \subset \mathscr{S}$ is the set of terminal states [20, 11]. A discount factor $\gamma \in [0,1]$ determines the importance of receiving a reward in the current state versus the future state, where $R_t = \sum_{k=0}^{\infty} \gamma^k r(a_{t+k}, s_{t+k})$ is the total accumulated return from the time step. The value function $V^{\pi}(s) = \mathbb{E}_{\pi}[R_t]$ of a policy is the measure of the expected sum of discounted rewards. The goal is to find the optimal policy $\pi^* = \arg\max_{\pi} \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r(a_t, s_t) | s_0 = s]$, which maps a history of observations to the next action[20, 19, 11, 18, 9, 13]. In practice, at each time step, the agent selects an action based on its current state and the policy, and receives a reward based on the transition to the next state. In our investigation of RL methods, we found that Soft Actor-Critic (SAC) is the best approach for our dNL based agent. SAC combines the strengths of both value-based and policy-based methods by alternating between updates to the policy and updates to the value function and Q-function. The entropy regularisation term is added to the policy update step to encourage exploration, and the algorithm uses a temperature parameter, α, to control the trade-off between exploration and exploitation. As we evaluate our agent on environments consisting of discrete actions, we use a discrete action variant of SAC [2]. The main difference is in the policy update step, where the objective becomes

$\pi^* = \arg\max_\pi \sum_{t=0}^{T} \mathbb{E}_{s_t \sim D}[\mathbb{E}_{a_t \sim \pi}[r(s_t, a_t) + \alpha \mathbb{H}(\pi(\cdot|s_t))]]$. Where $\mathbb{H}(\pi(a|s))$ is the entropy of the policy, γ is the discount factor, D is the replay buffer, and α is the temperature parameter. In the discrete setting, π maps states to a vector of probabilities with $|A|$ elements. For the actor cost function, we use the following equation $J_\pi(\phi) = \mathbb{E}_{s_t \sim D}[\mathbb{E}_{a_t \sim \pi}[\alpha \log(\pi_\phi(a_t|s_t)) - Q_\theta(s_t, a_t)]]$. Where π_ϕ is the policy parameterized by ϕ and Q_θ is the Q-function parameterized by θ. Discrete SAC policy maximises the probability of discrete actions as opposed to the continuous SAC policy which optimises two parameters of a Gaussian distribution.

2.2 Inductive Logic Programming (ILP)

Inductive logic programming (ILP) is a method of symbolic computing which automatically constructs logic programs given a background knowledge base (KB) [15]. An ILP problem is represented as a tuple $(\mathscr{B}, \mathscr{P}, \mathscr{N})$ of ground atoms, with $\mathscr{B}$ being the background assumptions, $\mathscr{P}$ being a set of positive instances which help define the target predicate to be learned, and $\mathscr{N}$ being the set of negative instances of the target predicate. The goal of ILP is to construct a logic program that explains all provided positive sets and rejects the negative ones. Given an ILP problem $(\mathscr{B}, \mathscr{P}, \mathscr{N})$, the goal is to identify a set of hypotheses (clauses) $\mathscr{H}$ such that $\mathscr{B} \wedge \mathscr{H} \models \gamma$ for all $\gamma \in \mathscr{P}$ and $\mathscr{B} \wedge \mathscr{H} \not\models \gamma$ for all $\gamma \in \mathscr{N}$, where $\models$ denotes logical entailment. In other words, the conjunction of the background knowledge and hypothesis should entail all positive instances and not entail any negative instances.

2.3 Differentiable Neural Logic (dNL)

The core component of the dNL network is their use of differentiable neural logic layers to learn Boolean functions [16]. The dNL architecture uses membership weights and conjunctive and disjunctive layers to learn a target predicate or Boolean function. Learning a target predicate p requires the construction of Boolean function $\mathscr{F}_p$ which passes in a Boolean vector $\mathbf{x}$ of size N with elements $x^{(i)}$, into a neural conjunction function f_{conj} (see equation 1a) which is defined by a conjunction Boolean function F_{conj} (see equation 1b). A predicate defined by Boolean function in this matter is extracted by parsing the architecture for membership weights ($w^{(i)}$) above a given threshold, where membership weights are converted to Boolean weights via a sigmoid $m^{(i)} = \sigma(cw^{(i)})$ with constant $c \geq 1$. Membership weights are paired with continuous lower and upper bound predicate functions (see equation 3) which are eventually interpreted as atoms in the body of the predicate being learned. These same Boolean predicate functions are used to transform non-Boolean data into a Boolean format for the logic layers.

$$f_{conj}(\mathbf{x}) = \prod_{i=1}^{N} F_{conj}(x^{(i)}, m^{(i)}) \tag{1a}$$

$$F_{conj}(x^{(i)}, m^{(i)}) = \overline{\overline{x^{(i)}} m^{(i)}} = 1 - m^{(i)}(1 - x^{(i)}) \tag{1b}$$

Following induction by the conjunctive layer, outputs are fed into a neural disjunction function f_{disj} (see equation 2a) which is constructed from disjunctive Boolean functions F_{disj} (see equation 2b). The disjunctive layer provides multiple definitions for our target predicate if necessary.

$$f_{disj}(\mathbf{x}) = 1 - \prod_{i=1}^{N} (1 - F_{disj}(x^{(i)}, m^{(i)})) \tag{2a}$$

$$F_{disj}(x^{(i)}, m^{(i)}) = x^{(i)} m^{(i)} \tag{2b}$$

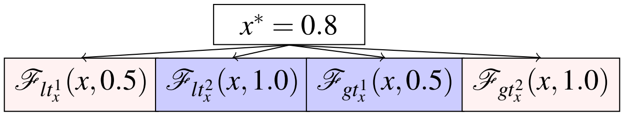


Figure 1: Given $k = 2$ and feature instance x^* with value 0.8, would result in two distinct activations of bounded Boolean predicates. Stating x^* *is less than* 1.0 *and* x^* *is greater than* 0.5.

As mentioned, a target predicate function $\mathscr{F}p$ is defined by the cascading architecture of the dNL network, and within the network bounded continuous Boolean predicates associated with a continuous feature are used to define our target predicate. Boolean predicate functions are used to handle continuous input by partitioning input into a series of lower and upper-bound predicates, referred to as continuous predicates as they capture the interval bounds of continuous features. Similarly, Boolean predicates mapped to discrete features are referred to as discrete predicates. These bounded continuous predicates return either true or false when a continuous value meets the condition. A continuous input x is associated with k pairs of upper and lower boundary predicates, each pair corresponding to a bounded range $(l_{xi} < x < u_{xi})$, where $i \in 1, 2, \cdots, k$. A Boolean upper boundary predicate $gt_x^i(x, lxi)$, states whether *"x is greater than l_{xi}"* is true, and a Boolean lower boundary predicate $lt_x^i(x, u_{xi})$ states whether *"x is less than u_{xi}"* is true (see equation 3). The lower and upper boundary values l_{xi} and u_{xi} are treated as trainable weights which can be optimised during training. To clarify the need for both lower and upper predicate bounds, the regions are complete and disjoint, meaning that each input value belongs to exactly one region. Although most of the upper and lower bounds may coincide, having both upper and lower predicates allows for greater flexibility in defining the regions and ensures that no input values fall outside the defined regions. A diagram illustrating this partitioning of the input domain can be found in Figure 1.

$$\mathscr{F}_{gt_x^i} = \sigma(c(x - u_{xi})), \quad \mathscr{F}_{lt_x^i} = \sigma(-c(x - l_{xi})) \tag{3}$$

3 Methodology

Developing a neuro-symbolic RL agent that incorporates continuous predicate functions posed several non-trivial challenges. Firstly, the work of Payani et al. [16, 17], which our model extends, did not explore non-linear predicate functions, so we had to develop new functions to capture more complex relationships in the data (see equation 7). Additionally, while Payani et al. did introduce continuous predicates in a supervised setting, they only tested dNL-ILP on two datasets (Wine and Sonar datasets [16, 5]), whereas we made the integration of continuous predicates a focus of our work. [1] Another challenge was the fact that the original dNL-RRL model explicitly dealt with discrete predicates, so we had to adapt the model to incorporate continuous predicates. Finally, while Payani et al. used the REINFORCE RL algorithm [21], we had to test out various other RL algorithms, including SAC [8], to find the most effective implementation for our model.[2]

In the dNL-RL interpretation, the Boolean target predicate function $\mathscr{F}_p$ is referred to as a *discrete action predicate function*. The actions of the agent are the target predicates and so must remain discrete. For example, an agent that can take an action $turnLeft()$, has the corresponding *discrete action predicate*

---

[1] Moreover, from an engineering standpoint, the dNL code was originally written in TensorFlow 1, which had to be converted to TensorFlow 2 for use in the RL environment.

[2] Extensive coding was required to convert the SAC algorithm, which typically has a discrete action space, to TensorFlow 2, as all available examples online were coded in PyTorch.

function $\mathscr{F}_{turnLeft}$. Each *discrete action predicate function* is defined by an input matrix $\mathbf{I}$, which is composed of continuous lower and upper bound predicate functions as well as discrete Boolean predicate functions, all with associated weights. The set of the continuous state features cnt_p and the set of the discrete state features dsc_p for the associated RL environment are given Boolean interpretations after being passed through the input matrix, as seen in equation 4. The row dimensions of $\mathbf{I}$ are defined by the batch size $\mathbf{b}$ and the column dimensions are defined by $N_e = (2 \times k \times |cnt_p|) + (2 \times |dsc_p|)$ (assuming discrete features are Boolean). The state feature ranges used to define the Boolean predicate functions are determined by equal-width binning, and we define the batch size as $\mathbf{b}$. The $\mathscr{F}_e$ function takes in a one-hot encoded Boolean discrete input.

$$\mathbf{I} = \begin{bmatrix} \mathscr{F}_{gt^1_{e_1}}, \mathscr{F}_{lt^1_{e_1}} & \cdots & \mathscr{F}_{gt^k_{e_1}}, \mathscr{F}_{lt^k_{e_1}} & \cdots & \mathscr{F}_{gt^k_{e_{|cnt_p|}}}, \mathscr{F}_{lt^k_{e_{|cnt_p|}}} \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \bigoplus \begin{bmatrix} \mathscr{F}_{e_1}, \mathscr{F}_{\overline{e_1}} & \cdots & \mathscr{F}_{e_{|dsc_p|}}, \mathscr{F}_{\overline{e_{|dsc_p|}}} \\ \vdots & \vdots & \vdots \end{bmatrix} \tag{4}$$

In defining the *discrete action predicate function* $\mathscr{F}_p$ for an action p, the function takes as input the input matrix $\mathbf{I}$ as well as the disjunction layer F_{disj} and the conjunction layer F_{conj} where N_e is used to define the number of layers in the conjunction layer and N_p is used to define the number of layers in the disjunction layer, seen in equation 5. As an agent learns rules for multiple actions, we define the set of discrete action predicates as the *predicate action policy* $\pi_{\mathscr{F}}$ which is the set $\{\mathscr{F}_{p^1}, \mathscr{F}_{p^2}, \cdots, \mathscr{F}_{p^n}\}$ for all actions in the action space $p \in \mathscr{A}$ for the RL environment.

To prevent the gradient from becoming excessively small during training, initial weights $\mathbf{W}$ are initialised using a negative mean random Gaussian distribution, which ensures that they are close to zero. To keep membership weights $\mathbf{m}$ between 0 and 1, a constant $c \geq 1$ is applied, followed by a sigmoid function. In equations 6a and 6b, we apply the conjunction function to the input matrix using the corresponding membership weights. Note that $\mathbf{W}^{conj}$ is a matrix with dimensions N_p and N_e while $\mathbf{W}^{disj}$ is a vector of size N_p. The disjunction function takes the output of the conjunction function as input as we cascade our neural architecture.

$$\mathscr{F}_p|_{\mathbf{I},N_e,N_p} = F_{disj}(N_p, F_{conj}(N_e, \mathbf{I})) \tag{5}$$

$$F_{conj} = \sum_{i=1}^{N_e} [1 - m_i^{conj}(1 - \mathbf{I}_i)] \quad \text{where} \quad \mathbf{m}^{conj} = \sigma(c\mathbf{W}^{conj}),\ \mathbf{W}^{conj}_{N_p,N_e} \sim \mathscr{N} \tag{6a}$$

$$F_{disj} = 1 - \sum_{i=1}^{N_p} [1 - m_i^{disj} F^i_{conj}] \quad \text{where} \quad \mathbf{m}^{disj} = \sigma(c\mathbf{W}^{disj}),\ \mathbf{W}^{disj}_{1,N_p} \sim \mathscr{N} \tag{6b}$$

We incorporate non-linear transformations on some of the state features. As in the case of various control problems in RL, the calculation of state transitions is based on non-linear transformations from the current state. We define the following k upper-boundary and lower-boundary functions as non-linear transformation predicates in equation 7.

$$\mathscr{F}_{gt^i_{f(x)}} = \sigma(c(f(x) - u_{xi})),\ \mathscr{F}_{lt^i_{f(x)}} = \sigma(-c(f(x) - l_{xi})) \,\text{where}\, f(x) \in \{sqr(x), sin(x), \cdots\} \tag{7}$$

Our dNL based agents can be differentiated by the type of prior information used. The baseline dNL agent uses only continuous and discrete Boolean predicates and is referred to as a dNL RL continuous (dNLRLc) agent. It learns the continuous state features without using any prior information from a knowledge base KB_T which contains transformation functions mapped to specific continuous state features. However, when prior information such as non-linear equations or transformations between features is included, the agent becomes a dNL RL non-linear continuous (dNLRLnlc) agent. This involves adding non-linear continuous Boolean predicates to the input matrix as new predicate functions in $\pi_{\mathscr{F}}$. In the experiment section, we specify which non-linear transformations were used for the dNLRLnlc agent in each RL environment.

In Algorithm 1 we provide a pseudo-code implementation for a dNL-RL agent. The agent takes in a state $\mathbf{s}_t$, at time step t and returns an action $\mathbf{a}_t$ for that time step. The algorithm iterates through each feature of the state, $s_t^{(i)}$, and separates the elements into two cases: discrete and continuous. If the element is discrete, it adds the mapping of the discrete predicate $dsc_p[i]$ to the state element $s_t^{(i)}$ in a processed state set $\mathbb{S}_t$. If the element is continuous, it checks if a transformation function for the feature exists in the knowledge base $\mathsf{KB}_T[i]$. If it does, it adds the mapping of the continuous predicate $cnt_p[i]$ to the function applied to the state feature $s_t^{(i)}$ in the state set $\mathbb{S}_t$. If the knowledge base does not exist, it adds the mapping of the continuous predicate $cnt_p[i]$ to the state feature $s_t^{(i)}$ instead. After iterating through all state features, the policy is returned by applying the dNLRL policy function to the set $\mathbb{S}_t$, and then sampling an action $\mathbf{a}_t$ from the *predicate action policy*.

Algorithm 1 dNL-RL agent

```
Input: $\mathbf{s}_t$-state at time step $t$
Output: $\mathbf{a}_t$-action at time step $t$
for $s_t^{(i)} \in \mathbf{s}_t$ do
    if $s_t^{(i)}$ is discrete then
        $\mathbb{S}_t \cup \{dsc_p[i] \mapsto s_t^{(i)}\}$
    end if
    if $s_t^{(i)}$ is continuous then
        if $\mathsf{KB}_T[i]$ exists then
            $\mathbb{S}_t \cup \{cnt_p[i] \mapsto \mathsf{KB}_T[i](s_t^{(i)})\}$
        else
            $\mathbb{S}_t \cup \{cnt_p[i] \mapsto s_t^{(i)}\}$
        end if
    end if
end for
$\pi_{\mathscr{F}} \leftarrow$ dNLRL($\mathbb{S}_t$)
$\mathbf{a}_t \sim \pi_{\mathscr{F}}(\mathscr{F}_{a_t} | \mathbb{S}_t)$
```

To derive the predicate action policy $\pi_{\mathscr{F}}$, we perform Boolean reasoning on the set of processed state features with associated predicates as seen in Algorithm 2. Here we take as input a set of discrete predicates dsc_p, a set of continuous predicates cnt_p, a set of target action predicates $\mathbb{P}$, and a set of processed state feature predicates $\mathbb{S}_t$. We loop through each target action predicate in the set $\mathbb{P}$ and for each predicate, it creates two empty sets used for constructing the input matrix, one for the continuous predicates $\mathbf{I}_c$ and one for the discrete predicates $\mathbf{I}_d$. The dNLRL policy then loops through each continuous predicate in cnt_p and for each predicate, it creates a disjunction of predicates, which are added to the continuous input matrix set $\mathbf{I}_c$. Then it does the same for each discrete predicate in dsc_p but instead, the discrete predicates are added to the discrete input matrix set $\mathbf{I}_d$. We take the union of these two sets to get the final input matrix $\mathbf{I}$. Finally, it creates an evaluated target action predicate $\mathscr{X}_p$ by reasoning on target predicate function $\mathscr{F}_p$ as defined in equation 5. This target action predicate $\mathscr{X}_p$ is then added to the predicate action policy $\pi_{\mathscr{F}}$.

Algorithm 2 Single Step Forward Chain Model/ dNLRL policy

```
Input: $dsc_p$-set of discrete predicates, $cnt_p$-set of continuous predicates, $\mathbb{P}$-set of target action predicates, $\mathbb{S}_t$-set of state feature predicates at time step $t$
Output: $\pi_{\mathscr{F}}$-predicate action policy
for $p \in \mathbb{P}$ do
    $\mathbf{I}_c$=[]
    $\mathbf{I}_d$=[]
    for $e \in cnt_p$ do
        $\mathbf{I}_c \cup \bigvee_{i=1}^{k} \left( \mathscr{F}_{gt_e^i}(\mathbb{S}_t[e]) \vee \mathscr{F}_{lt_e^i}(\mathbb{S}_t[e]) \right)$
    end for
    for $e \in dsc_p$ do
        $\mathbf{I}_d \cup \left( \mathscr{F}_e(\mathbb{S}_t[e]) \vee \mathscr{F}_{\overline{e}}(\mathbb{S}_t[e]) \right)$
    end for
    $\mathbf{I} \leftarrow \mathbf{I}_d \cup \mathbf{I}_c$
    $\mathscr{X}_p \leftarrow \mathscr{F}_p|_{\mathbf{I},N_e,N_p}$
    $\pi_{\mathscr{F}} \cup (\mathscr{X}_p)$
end for
```

4 Experiments and Results

4.1 Environments and Tasks

The following RL environments are continuous control problems developed by Open AI Gym [1]. The Cart Pole problem is a benchmark for evaluating RL algorithms [12]. The goal of the problem is to balance a pole in the upright position by selecting between two discrete actions (move left, move right). The state space is 4-dimensional, all continuous. The features are the following x : Cart Position, x' : Cart Velocity, θ : Pole Angle, and θ' : Pole Angular Velocity. In the Lunar Lander environment, an agent learns a policy for throttling the side and main engines to control the descent of a lander such that it lands on a landing pad. The discrete actions include activating the right thruster engine, activating the left thruster engine, activating the main engine, and doing nothing. The state space is 8-dimensional, 6 inputs are continuous and 2 are discrete. The continuous state inputs include including x : Lander Position on the X-axis, y : Lander Position on the Y-axis, v_x : Horizontal velocity, v_y : Vertical velocity, θ : Angular orientation in space, and v_θ : Angular velocity. The remaining two features are Boolean, Left: which indicates if the left leg is touching the ground and Right: which indicates if the right leg is in contact with the ground.

4.1.1 Baselines

During our investigation into the development of our continuous logic-based algorithm, we sought first to see if the dNLRLc agent could in fact learn interpretable policies on continuous environments. As stated this led to evaluation of the dNLRLc architectures on various RL algorithms. This includes the policy gradient algorithm REINFORCE [21], the model-free algorithm Deep Q-Network (DQN) [14], the on-policy method of Advantage Actor-Critic (A2C) [13], and the Soft Actor-Critic (SAC) [8]. In the case of SAC, we use a variant designed to handle discrete actions [2]. In cases where we inject prior knowledge in the form of non-linear functions added to the transformation knowledge base, we designate the model as (SAC-NL).

4.2 Algorithm Performance Comparison

We present algorithmic comparisons of the various RL algorithms combined with the dNLRLc agent where evaluation is carried out on the Cart Pole problem. For the evaluation, only SAC is used to test the dNLRLnlc agent, as it was the best-performing algorithm with the dNLRLc agent. In the Lunar Lander environment, we evaluate a dNLNLc agent and dNLRLnlc agent using SAC entirely. Each algorithmic framework used the same binning scheme for the continuous features, that being ‘equal-width binning’ where bins for each feature were predefined for the learning environment and kept constant between the various RL algorithms.

4.2.1 Task: Cart Pole Problem

For the Cart Pole problem, each algorithmic framework used the same binning scheme for ‘equal-width binning’ with the number of bins set to $[4,4,4,4]$ for each feature respectively. In Figure 2, we can observe the results comparing the performances of each dNLRLc agent and paired RL architecture in the Cart Pole Problem domain. As can be observed, the best performing model is that of the SAC and SAC-NL. Performance of dNL based agents is noticeably poor in the cases of the REINFORCE algorithm where rewards cap at 22.0. DQN and A2C perform better but rewards plateau far below what is to be expected. The non-linear addition for the dNLRLnlc agent is a simple transformation of the Pole Angle θ, as the calculation of the next state s_{t+1} within the code for the Cart Pole problem is dependent on the *sine* transformation of θ. For SAC-NL, we apply a *sine* transformation and so policies can contain atoms designated *PoleAngleSine* with corresponding inequalities. The performance of the dNLRLnlc agents is not significantly better than the dNLRLc agent when using SAC. Both agent variants are able to achieve the maximum reward of 300.0 consistently with periodic fluctuations. In figure 4, we can observe the moving standard deviation for the dNLRLc and dNLRLnlc when using SAC. Although the fluctuations in the standard deviations are not in sync, the variation between the two agents is not significant enough to conclude that one agent is more stable.

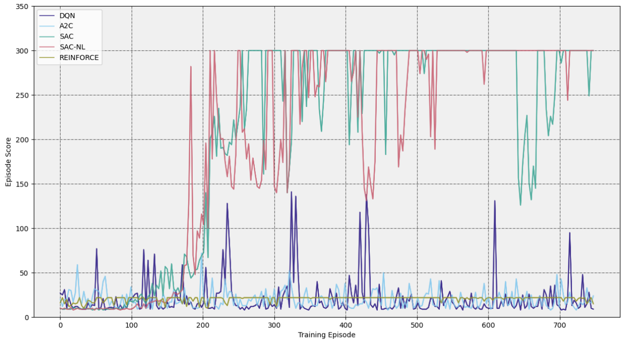


Figure 2: Episode scores for RL algorithms relying on dNLRLc and dNLRLnlc agents.

Figure 3 presents a comparison of the performance of the various RL algorithms using standard neural networks in solving the Cart Pole problem. The figure shows that, while SAC is able to solve the problem and stabilise, the neural network agents learn faster and are more stable than dNLRLc agents. Both agents using A2C and DQN exhibit inconsistent performance with rewards fluctuating greatly and both the neural networks and dNLRLc agents perform poorly when using REINFORCE. These findings

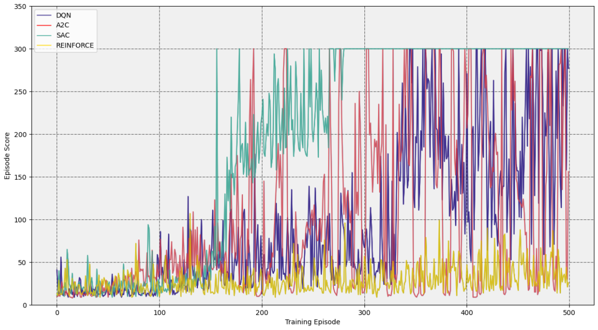


Figure 3: Episode scores for RL algorithms relying on neural network based agents.

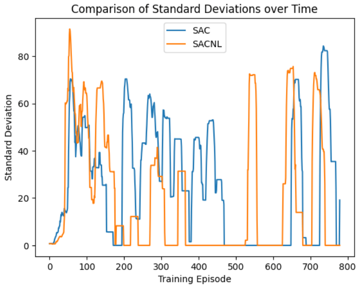


Figure 4: Moving (window size 20) standard deviation for SAC relying on dNLRLc and dNLRLnlc agents.

suggest that while dNLRLc models may offer greater interpretability, they may come at the cost of decreased performance in some RL tasks. Given that, it is also evident that SAC is the primary contributor to solving the control problem as even standard neural network based agents failed otherwise. We also note that the black-box neural network-based SAC agent outperforms the interpretable dNL-RL SAC agent in terms of speed, achieving an average convergence time of 2168.31 seconds, compared to the average convergence time of 10807.94 seconds for the dNL-RL agent, highlighting a trade-off between interpretability and learning efficiency, which could be addressed in future research by optimising the algorithm for faster convergence.

The main advantage of using dNL agents is that returned policies are interpretable. These policies provide predicate logic interpretations for the individual discrete actions, in the case of the Cart Pole problem, we have rules for when to move *left()* and when to move *right()*. The policy in our case is defined as *predicate action policy*, that is individual actions are treated as target predicates where the clausal body states what inequalities should be true for a state feature in order for the action to be considered by the agent. We also clarify that the clausal body is defined by a conjunction of atoms where the associated membership weight is placed before the atom in brackets, extracted from the conjunction neuron. If the membership weight is above 0.95, i.e. the model is confident it should be in the definition, then the weight is not listed next to the atom. Similarly, for the disjunction neurons, the value in brackets before the rule represents the membership weights.

CartPole	Policy rules for dNLRLc
mean reward: 290.3 ± 32.3	**left()**
	$: -([0.56]\mathit{CartPos} < 2.83 \wedge [0.64]\mathit{CartVeloc} > -1.19 \wedge \mathit{CartVeloc} > 0.18$
	$\wedge \mathit{PoleAngle} > -0.62 \wedge \mathit{PoleAngleVeloc} > -0.24)$
	$: -([0.81]\mathit{CartPos} < 2.82 \wedge \mathit{PoleAngle} > -0.06 \wedge \mathit{PoleAngleVeloc} > 0.08)$
	right()
	$: -(\mathit{CartVeloc} > -1.19 \wedge \mathit{PoleAngle} < -0.03 \wedge \mathit{PoleAngleVeloc} < 0.34 \wedge [0.56]\mathit{PoleAngleVeloc} < 2.48)$
	$: -(\mathit{CartPos} < 0.16 \wedge [0.52]\mathit{CartVeloc} < 2.11 \wedge [0.50]\mathit{PoleAngle} > -0.56$
	$\wedge \mathit{PoleAngle} < 0.11 \wedge \mathit{PoleAngleVeloc} < 0.25)$

Table 1: The FOL policy rules for the dNLRLc agent, trained using SAC. Extracted continuous inequality predicates define rules for each action.

In table 1, the predicate action policy for the dNLRLc agent trained using SAC is provided. In the body of the discrete action predicate *left()*, it is observed that two definitions are returned wherein which we see both associated disjunction neuron membership weights pass the threshold. The three state features present in the second definition are Cart Position ($CartPos < 2.82$), Pole Angle ($PoleAngle > -0.06$), and Pole Angular Velocity ($PoleAngleVeloc > 0.08$). Similarly, the body for the discrete action predicate *right()* contains two predicate definitions. In both cases, the agent is confident in either definition as indicated by the absence of a membership weight for the corresponding disjunction neuron.

CartPole	Policy rules for dNLRLnlc
mean reward: 294.7 ± 25.8	**left()**
	$: -([0.60]CartPos < 2.57 \wedge PoleAngleSine > 0.00 \wedge PoleAngleVeloc > -0.38)$
	right()
	$: -(PoleAngleSine < 0.04 \wedge PoleAngleVeloc < 0.00)$
	$: -(CartPos < 0.74 \wedge [0.55]CartVeloc > -1.64 \wedge CartVeloc < -0.11$ $\wedge PoleAngleSine < 0.65 \wedge [0.66]PoleAngleVeloc > -2.04 \wedge PoleAngleVeloc < 0.28)$

Table 2: The FOL policy rules for the dNLRLnlc agent, trained using SAC. Extracted continuous inequality predicates define rules for each action with the inclusion of non-linear continuous inequality predicates.

In table 2, the predicate action policy for the dNLRLnlc agent trained using SAC is provided. The definitions of the predicate action policy are comparable to those given by the dNLRLnlc agent. Note that the discrete action predicate *left()* is defined by a single predicate rule. For both discrete action predicates, we see the presence of the non-linear continuous predicate in the definitions, that is $(PoleAngleSine > 0.00)$ of the discrete action predicate $left()$ and $(PoleAngleSine < 0.00)$ for the first definition of $right()$ and $(PoleAngleSine < 0.65 \wedge PoleAngleSine < 0.28)$ for the second definition. In figure 1 and figure 2 the mean reward is also provided with the standard deviation. The mean reward in this case refers to the mean of the last 100 episodes.

4.2.2 Task: Lunar Lander

At present, the Lunar Lander stands as a more challenging environment for the dNL-RL agents. As the previous control problem environment has demonstrated that SAC is the best performing algorithm, only SAC was tested on the Lunar Lander environments. In setting up the experiments, it was found that both the discretisation scheme and the initial high and low values for the state features had a significant impact. In the context of the Lunar Lander, a reward of 200 indicates the lander successfully landed on the platform. The current binning scheme being deployed is $[3,3,3,3,3,3,-,-]$ for each feature, where $(-)$ indicates no binning performed as it is a discrete/Boolean feature. In order to evaluate the consistency and performance of the dNLRLc and dNLRLnlc agents, multiple trials were conducted using different random seeds. We ran each experiment 5 times and the resulting policies were analysed. While only one representative policy is presented for each agent, any variations or discrepancies across the trials are discussed. The non-linear change was again a *sine* transformation on the state feature angular orientation in space θ associated with the transformation predicate *AngleSine*.

In Figure 5, we can observe the results comparing the performance of the dNLRLc agent where dark purple corresponds to the mean reward. The results indicate the agent does in fact learn, however, it is challenging for the dNL agent to successfully land the lander consistently. The episode rewards show the agent can land the lander but the mean rewards skew below a true success of 200. We note the mean reward across all trials fluctuates around rewards of a hundred with significant deviations from the mean. In some initial states, the learned policy does result in successful landings, often over 200. The rules in the predicate action policy for the Lunar Lander environment are given in table 3. As the Lunar Lander has four actions, we find that all disjunction layers pass the parameter threshold and only in a few cases do we have conjunction weights for specific atoms. For example for the rule $fireLeft$, the first rule contains a weighted atom $[0.65]CoordX > -1.52$. For the action predicate, *fireRight()* the agent learned four rules, where only the second rules contain listed conjunction layer weights for individual atoms $[0.60]CoordX > -1.53$ and $[0.94]AngularVeloc > -2.58$. We observe as well, that while the bounded atom $CoordX$ appears in the definitions for each action, each is associated with a conjunction weight. Indicating the agent was less confident about the inclusion of the $CoordX$ continuous predicate function.

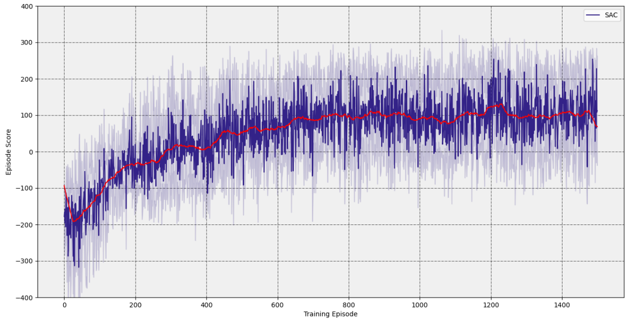


Figure 5: Episode scores for SAC algorithms relying on dNLRLc. Dark purple corresponds to the mean reward. Light purple corresponds to the standard deviation and the red line is the moving average for 50 episodes.

LunarLander	Policy rules for dNLRLc
mean reward: 162.1 ± 110.9	**doNothing()**
	$: -(CoordY < 1.28 \wedge LinearVelocX < -0.14)$
	$: -([0.80]CoordX > -1.53 \wedge [0.76]CoordX < 1.70 \wedge LinearVelocY < -0.12$ $\wedge [0.66]Angle > -1.60)$
	$: -([0.82]CoordX < 1.70 \wedge LinearVelocY < -0.04 \wedge RightLegContactFalse)$
	fireLeft())
	$: -([0.65]CoordX > -1.52 \wedge LinearVelocX < -0.14 \wedge [0.73]Angle > -0.09)$
	$: -([0.72]CoordX < 1.70 \wedge [0.68]CoordY > -0.99 \wedge LinearVelocX < 1.93 \wedge Angle > -0.09)$
	fireMain()
	$: -(LinearVelocX > 0.20 \wedge Angle < 0.17 \wedge AngularVeloc < 0.10)$
	$: -([0.58]CoordY > -0.91 \wedge LinearVelocY > -0.11)$
	$: -([0.75]CoordX < 1.70 \wedge LinearVelocX < -0.11 \wedge Angle > 0.01 \wedge AngularVeloc > -0.14)$
	fireRight()
	$: -(LinearVelocX > -0.63 \wedge Angle > -1.60 \wedge Angle < 0.17 \wedge AngularVeloc < -0.02)$
	$: -([0.60]CoordX > -1.53 \wedge Angle < 0.17 \wedge [0.94]AngularVeloc > -2.58$ $\wedge LeftLegContactTrue)$
	$: -(CoordY < 1.28 \wedge LinearVelocX > -0.05 \wedge Angle < 0.32 \wedge AngularVeloc > -1.98$ $\wedge RightLegContactFalse)$
	$: -(LinearVelocX > -0.05 \wedge Angle > -1.60 \wedge Angle < 0.17)$

Table 3: The FOL policy rules for the dNLRLc agent, trained using SAC. Extracted continuous bounded predicates define rules for each action.

In Figure 6, we can observe the results comparing the performance of the dNLRLnlc agent. The inclusion of the *sine* is a very minor addition, but one that does take into account the state transition mechanics. Performance is better than dNLRLc in later episodes. The moving average shows mean rewards rising above 100.0 consistently. The rules in the predicate action policy for the Lunar Lander environment are given in table 4. For each discrete action, all rules pass the parameter threshold for the disjunction layer except for a single rule associated with $fireLeft()$. For the predicate actions *fireLeft()* and *fireRight()*, which have four rules each, we observe the majority of definitions include the *sine* transformed state feature θ.

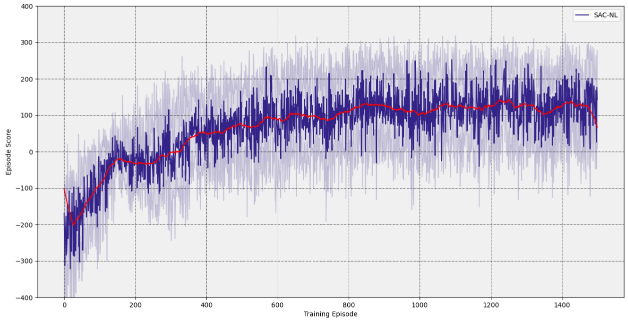


Figure 6: Episode scores for SAC algorithms relying on dNLRLnlc.

4.2.3 Discussion

The performance of both the dNLRLc and dNLRLnlc agents is comparable. In the CartPole problem, see Figure 2, the dNLRLnlc appears more stable in later episodes than dNLRLc as evidenced by the higher occurrence of total rewards of 300. The Lunar Lander problem, with a larger action and state space, proved more challenging for the dNLRL agents. Both agents were unable to produce mean rewards of 200, although dNLRLnlc achieved higher rewards than dNLRLc in later episodes. While the deviation was high, indicating both dNLRLc and dNLRLnlc landed successfully on occasion, this also signified that the Lander would crash periodically. The various hyperparameters were also found to impact the model. Specifically, the instantiation of the binning scheme would sway performance. We note, that in [7], equal-width binning was not necessarily an optimal approach for the Lunar Lander except in specific regions with respect to the positional axis, and we leave this investigation for future research. Multiple trials of both agents were conducted, with performance across all episodes averaging around 100 for dNLRLc and slightly higher for dNLRLnlc, and performance fluctuations resulted in a variety of final policy rules. The position of the lander along the X and Y axis, as signified by atoms associated with $CoordX$ or $CoordY$, would often have corresponding conjunction weights, indicating uncertainty in these predicates. The fluctuating bounded values during training might have caused confusion for the agent, as the position of the lander is a significant factor in landing success. To address this issue, future investigations could explore alternative schemes for training the bound weights, and consider adding additional non-linear transformation predicates or operation predicates between state features.

5 Conclusion

We aimed to incorporate both continuous and non-linear interpretations of dNL networks into an RRL framework, creating a dNL-ILP based agent that can learn in dynamic continuous environments. SAC was found to be the best among RL algorithms for evaluating the agent. The agent produced policies

LunarLander	Policy rules for dNLRLnlc
mean reward: 133.4 ± 113.9	**doNothing()**
	$:-(CoordX < 1.66 \wedge LinearVelocY < -0.61)$
	$:-([0.90]CoordX < 1.66 \wedge [0.65]CoordY > -0.98 \wedge [0.91]CoordY < 1.42 \wedge LeftLegContactFalse))$
	$:-(LinearVelocY < -0.22)$
	$:-([0.61]CoordX < 1.51 \wedge AngleSine < -0.25)$
	$:-([0.86]CoordY < 2.12 \wedge AngularVeloc > 0.23)$
	fireLeft())
	$:-(LinearVelocX < -0.15)$
	$:-([0.88]CoordY > -0.98 \wedge LinearVelocX < 0.20 \wedge [0.56]LinearVelocY < 0.92 \wedge AngleSine > -0.08)$
	$:-([0.53]AngleSine > -1.00 \wedge AngleSine > -0.08 \wedge AngularVeloc > 0.07)$
	$:-[0.65]([0.52]CoordY > -0.98 \wedge CoordY < 0.14 \wedge [0.72]LinearVelocY > -2.32, [0.68]AngleSine > -1.00$
	$\wedge AngleSine > -0.08 \wedge [0.85]AngularVeloc < 1.88)$
	fireMain()
	$:-(LinearVelocX < -0.15 \wedge LinearVelocX < -0.58 \wedge [0.71]LinearVelocX < 0.20 \wedge AngleSine > -0.08 \wedge$
	$AngularVeloc > -1.12 \wedge LeftLegContactFalse)$
	$:-(CoordX > -1.06 \wedge LinearVelocY > -0.11)$
	$:-(CoordY > 0.34 \wedge LinearVelocY > -0.42 \wedge [0.62]AngularVeloc > -1.12)$
	$:-(LinearVelocX > 0.42 \wedge AngleSine < -0.25 \wedge AngularVeloc > -1.12)$
	fireRight()
	$:-(LinearVelocX > -0.25 \wedge LinearVelocY < -0.61 \wedge AngleSine < 0.37, [0.73]AngularVeloc < 2.08)$
	$:-(CoordX > 0.36 \wedge CoordY < 0.14 \wedge [0.64]LinearVelocY > -2.32 \wedge [0.70]AngleSine < 0.37$
	$\wedge AngleSine < 0.25 \wedge [0.74]AngularVeloc > -1.12)$
	$:-([0.52]CoordY > -0.91 \wedge AngleSine < 0.25 \wedge AngularVeloc < 0.05)$
	$:-(CoordX > 0.36 \wedge CoordY < 1.42 \wedge AngleSine < 0.25 \wedge AngularVeloc < 1.88 \wedge LeftLegContactFalse)$

Table 4: The FOL policy rules for the dNLRLnlc agent, trained using SAC. Extracted continuous bounded predicates define rules for each action with the inclusion of non-linear continuous bounded predicates.

incorporating continuous and non-linear continuous predicate functions and was the first to successfully integrate ILP-based reasoning, RRL, and learning in dynamic continuous settings. The Lunar Lander problem was more challenging for the dNLRL agents, resulting in a high deviation from the mean, but our dNLRLc and dNLRLnlc agents still provide a promising starting point for ILP and RL research in continuous domains.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang & Wojciech Zaremba (2016): *OpenAI Gym*. Available at `http://arxiv.org/abs/1606.01540`.

[2] Petros Christodoulou (2019): *Soft Actor-Critic for Discrete Action Settings*, doi:10.48550/ARXIV.1910.07207. Available at `https://arxiv.org/abs/1910.07207`.

[3] Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li & Denny Zhou (2019): *Neural Logic Machines*, doi:10.48550/arXiv.1904.11694. arXiv:1904.11694.

[4] Kurt Driessens (2010): *Relational Reinforcement Learning*, doi:10.1007/978-0-387-30164-8_721.

[5] Dheeru Dua & Casey Graff (2017): *UCI Machine Learning Repository*. Available at `http://archive.ics.uci.edu/ml`.

[6] Richard Evans & Edward Grefenstette (2017): *Learning Explanatory Rules from Noisy Data*, doi:10.48550/arXiv.1711.04574. arXiv:1711.04574.

[7] Soham Gadgil, Yunfeng Xin & Chengzhe Xu (2020): *Solving The Lunar Lander Problem under Uncertainty using Reinforcement Learning*, doi:10.48550/arXiv.2011.11850. arXiv:2011.11850.

[8] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel & Sergey Levine (2018): *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*, doi:10.48550/ARXIV.1801.01290. Available at `https://arxiv.org/abs/1801.01290`.

[9] Rodrigo Toro Icarte, Toryn Klassen, Richard Valenzano & Sheila McIlraith (2018): *Using Reward Machines for High-Level Task Specification and Decomposition in Reinforcement Learning*. Available at `http://proceedings.mlr.press/v80/icarte18a.html`.

[10] Zhengyao Jiang & Shan Luo (2019): *Neural Logic Reinforcement Learning*, doi:10.48550/arXiv.1904.10729. arXiv:1904.10729.

[11] Jan Leike, David Krueger, Tom Everitt, Miljan Martic, Vishal Maini & Shane Legg (2018): *Scalable agent alignment via reward modeling: a research direction*, doi:10.48550/arXiv.1811.07871. arXiv:1811.07871.

[12] Nikita I. Lytkin, Risto Miikkulainen & Nate Kohl (2005): *Reinforcement Learning Benchmarks and Bake-offs II A workshop at the 2005 NIPS conference*.

[13] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver & Koray Kavukcuoglu (2016): *Asynchronous Methods for Deep Reinforcement Learning*, doi:10.48550/ARXIV.1602.01783. Available at `https://arxiv.org/abs/1602.01783`.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra & Martin Riedmiller (2013): *Playing Atari with Deep Reinforcement Learning*, doi:10.48550/ARXIV.1312.5602. Available at `https://arxiv.org/abs/1312.5602`.

[15] Stephen Muggleton & Luc de Raedt (1994): *Inductive Logic Programming: Theory and methods*, doi:10.1016/0743-1066(94)90035-3. Available at `http://www.sciencedirect.com/science/article/pii/0743106694900353`. Special Issue: Ten Years of Logic Programming.

[16] Ali Payani & Faramarz Fekri (2019): *Inductive Logic Programming via Differentiable Deep Neural Logic Networks*, doi:10.48550/arXiv.1906.03523. arXiv:1906.03523.

[17] Ali Payani & Faramarz Fekri (2020): *Incorporating Relational Background Knowledge into Reinforcement Learning via Differentiable Inductive Logic Programming*, doi:10.48550/arXiv.2003.10386. arXiv:2003.10386.

[18] Melrose Roderick, Christopher Grimm & Stefanie Tellex (2017): *Deep Abstract Q-Networks*, doi:10.48550/arXiv.1710.00459. arXiv:1710.00459.

[19] Andrew Silva, Taylor Killian, Ivan Dario Jimenez Rodriguez, Sung-Hyun Son & Matthew Gombolay (2020): *Optimization Methods for Interpretable Differentiable Decision Trees in Reinforcement Learning*, doi:10.48550/arXiv.1903.09338. arXiv:1903.09338.

[20] Richard S Sutton, Andrew G Barto et al. (1998): *Introduction to reinforcement learning*.

[21] Richard S Sutton, David McAllester, Satinder Singh & Yishay Mansour (1999): *Policy Gradient Methods for Reinforcement Learning with Function Approximation*. Available at `https://proceedings.neurips.cc/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf`.

[22] Yuan Yang & Le Song (2019): *Learn to Explain Efficiently via Neural Logic Inductive Learning*, doi:10.48550/arXiv.1910.02481. arXiv:1910.02481.

[23] Vinicius Zambaldi, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David Reichert, Timothy Lillicrap, Edward Lockhart, Murray Shanahan, Victoria Langston, Razvan Pascanu, Matthew Botvinick, Oriol Vinyals & Peter Battaglia (2018): *Relational Deep Reinforcement Learning*, doi:10.48550/arXiv.1806.01830. arXiv:1806.01830.

An xAI Approach for Data-to-Text Processing with ASP [*]

Alessandro Dal Palù
Università di Parma, Italy
GNCS-INdAM, Roma, Italy
alessandro.dalpalu@unipr.it

Agostino Dovier
Andrea Formisano
Università di Udine, Italy
GNCS-INdAM, Roma, Italy
agostino.dovier@uniud.it
andrea.formisano@uniud.it

The generation of natural language text from data series gained renewed interest among AI research goals. Not surprisingly, the few proposals in the state of the art are based on training some system, in order to produce a text that describes and that is coherent to the data provided as input. Main challenges of such approaches are the proper identification of *what* to say (the key descriptive elements to be addressed in the data) and *how* to say: the correspondence and accuracy between data and text, the presence of contradictions/redundancy in the text, the control of the amount of synthesis.

This paper presents a framework that is compliant with xAI requirements. In particular we model ASP/Python programs that enable an explicit control of accuracy errors and amount of synthesis, with proven optimal solutions. The text description is hierarchically organized, in a top-down structure where text is enriched with further details, according to logic rules. The generation of natural language descriptions' structure is also managed by logic rules.

1 Introduction

The last decades witnessed a remarkable attention to the impact of Artificial Intelligence (AI) on many aspects of everyday life. The ensuing debate underscores the social and ethical implications of the use of AI-based systems and is maturing into new regulations in USA and in Europe [15, 4]. The European proposal should be operational by 2024 and it introduces a risk-based classification of AI systems based on the principles of ethical AI [9].

The term *explainable AI* (xAI, see, for instance, [3, 1]) has emerged to capture desirable properties of high-risk systems: e.g., transparency (or glass box approach), ethics, the capability to support results in terms of intelligible descriptions, accountability, security, privacy, and fairness. In the near future, the adoption of AI systems will depend on the capability of providing a high-level description of inner activities, which promotes the interpretability and transparency of the decisions that lead to a result. Hence, domain experts could easily understand and criticize the contributions coming from an AI system, in a similar way colleague's opinions would be considered.

In general terms, the data-to-text generation task consists in automatically generating descriptions from non-linguistic data. In recent years, there has been growing interest in such systems (see for instance the work [6], supported by the Alexa AI organization). Systems able to textually summarize data, such as time-series, can help in making data more accessible to non-experts, compared to traditional forms of presentations. Indeed, natural language descriptions of data can help readers even in cases where data are depicted by means of forms of representations usually considered "self-explanatory", such as charts, histograms, pies, etc, that might be difficult to read if the amount and/or complexity of data is large.

Text-to-speech tools, such as screen readers, might exploit textual generator to extract information from charts contained in a digital document. This is also relevant in situations where the interpretation

---

[*] Partially supported by the *Interdepartmental Project on Artificial Intelligence* (Strategic Plan UniUD–22-25) and by INdAM–GNCS projects CUP E55F22000270001 and CUP E53C22001930001.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 353–366, doi:10.4204/EPTCS.385.38

of visualizations is made difficult or hindered for people with visual impairments or when readers have limited cognitive abilities in comprehending and analyzing complex charts.

Variants of this problem have been widely studied and solutions have been proposed in various domains. For example, [13] and [16] apply a data-to-text generation system to interpret data-records concerning NBA basketball games. In the financial field, [10] proposes an approach to the problem of comment generation for time-series of stock prices. Different solutions have been proposed expressively to treat specific forms of diagrammatic representations of data such as charts and histograms [11, 2] or tables [8]. In the healthcare domain, [12] presents the prototype BT-45 aimed at generating textual summaries of physiological clinical data (acquired automatically from sensors or entered routinely by the medical staff). For space limits, we cannot provide an exhaustive description of the literature concerning data-to-text generation. We refer the reader to the previously mentioned papers and to references therein.

Almost all the recent approaches to data-to-text are based on Machine Learning (ML) and ultimately rely on Deep Learning (DL) techniques, usually exploiting Neural Networks training algorithms. For example, the paper [6], mentioned earlier, proposes DataTuner a neural data-to-text generation system.

There are a few exceptions worth mentioning. Among them we mention the seminal work [7] that proposes a rule-based approach (actually, an expert system written in the OPS5 production system language) to generate descriptions of inputs from a Dow Jones stock quotes database. Another notable exception is the BT-45 system cited earlier. It exploits signal analysis and pattern detection techniques to process input time series and relies on a corpus of human-authored sample linguistic expression of concepts to be used in generating textual summaries. The system is one of the few proposals appeared in the literature that do not rely on ML/DL techniques. In this sense, it is similar to our approach. Nevertheless, it is conceived and designed for a very specific application field. Indeed, it requires the development, by human experts, of an ontology of medical concepts. Such an ontology is essential both for assessing the importance of patterns in input data and for generating expert-oriented output descriptions (see [12] for a detailed presentation of the system).

It should be noted that Machine Learning and Deep Learning methods optimize inner parameters (sub-symbolic representation) to adhere to training targets, while they struggle to produce a high-level description (or symbolic) about how and why the network learned specific parameters (black box approach). This limitation favoured post-hoc explanations [3], e.g., textual captioning and/or visual explanation by examples, still powered by an additional sub-symbolic machinery. Even though explanations in Deep Learning approaches are one of the next big challenges for AI, [14] expresses some concerns about the accuracy achievable by this type of explainability and postulates that interpretable models for high-stakes decisions should be sub-symbolic free systems.

The urge for explainability represents an opportunity of discontinuity with respect to current AI methods. We divert from mainstream methods based on ML and DL, and the recent trend of producing a high-level explanation of such black box systems. The design of an architecture that is explainable and ML/DL free represents an innovative and unexplored research direction.

Our strategic choice is to use robust and off-the-shelf technologies to reach xAI compliance. We select Answer Set Programming (ASP) and Clingo [5] as the explainable core of the system, because they allow knowledge representation at a high-level and enables reasoning about such knowledge. This framework not only is explainable, but also domain experts, in a scientists-in-the-loop fashion, can detect flaws in the automated process and review the knowledge base accordingly. Handling of incoherent points of view, inconsistent rules, uncertainty and lack of knowledge can be explicitly supported by the system.

2 System's Design

Our systems models the selection of *what* to say in (which descriptions are interesting to be narrated) and *how* to say (how to build the qualitative features and how to create a narration structure) the final narration. The core of the system is explainable by design, since we develop a transparent deductive handling of semantic rules that describe features, by exploiting ASP. The descriptions selection is done through two optimization steps performed by two ASP programs which determine, possibly multiple, descriptions for each relevant region of the input. Then, the best candidate descriptions (for different levels of detail) are selected. The outcome of this selection phase is then processed, again by using ASP, to determine structural relations among the fragments (narration order, dependencies, etc) to be used for the generation of the narration. A Python script controls the pre-computation of input, the handling of ASP programs execution and the conversion of ASP's output to text and enriched plots.

2.1 Descriptions and curve fitting

Let us present the overall idea and specific explainability targets we set in the process of conversion of a data series to text. In this work we focus on scalar data acquired on a time domain (represented by a sequence of floats). Each element can be associated to a timestamp. A description of the data is a simple and peculiar feature that generalizes a particular distribution along time. A possible bridge from data to robust descriptions is to rely on fitting functions. We select function prototypes that are fitted to data (i.e., least square curve fitting) and that are described by few and intuitive parameters, in order to promote a simple textual description. In the paper we refer to *description/descriptor* as the function model and its parameters, while we refer to *narration* as the corresponding natural language wording of such description. As an example let us consider the concept of a valley in a series: a sudden decrease followed by another sudden increase of values can be modeled by a specific function (e.g., a V like function). The feature extraction, based of curve fitting, allows to retrieve which optimal parameters provide minimal Root Mean Square Error (RMSE) of the curve w.r.t. the original data. Parameters, that control the degrees of freedom of the shape of the curve, allow to build a high level narration. For example, if the two sides of the valley have a high angular coefficient (or vertical orientation), the valley can be described as sharp. Potential domain-dependent descriptors can be encoded (e.g., specific patterns) and added with no relevant modification of the system.

In our model, we allow an arbitrary large number of functions prototypes, to support a modular and general usage. Each of them allows to investigate particular behaviors. Each prototype requires to encode its fitting function and its conversion to text as natural language description (we discuss about it later). In our system we currently encoded (number of parameters inside brackets): line (2), two segments polyline (4), a tooth like function (5) and the most prominent sinusoid (3). We plan for the future to include the analysis of transformed spaces (e.g., Fourier transform and wavelets) and patterns occurring with characteristic shapes (e.g., EEG tracks).

Such choice allows to accomplish some steps towards explainability. In particular, measuring the accuracy of a description, allows the system to provide a self-judgement feedback. There is a measurable error (the fitting accuracy) and a more qualitative error caused by conversion of descriptions (numerical) to narrations (textual).

When interpreting the evolution of a measure along time, often we extrapolate a multi-resolution description: in the simplest case we focus on the overall trend and also on remarkable details. Therefore, we pre-process descriptors at different sub-intervals, in order to capture potential features at different time scales. Formally, we divide the time domain into regular $n = 2^\ell$ zones. Each zone covers a subset of time samples and it is the smallest unit for RMSE optimizations. The parameter ℓ (levels of divisions) controls the amount of time resolution is provided to descriptors. Each descriptor d covers a zone range from i to j: $Z_d = \{z : 0 \leq i \leq z \leq j < n\}$. In Figure 1 we show an example for some descriptors (bilinear, sinusoid and squared peak), colored by zones size, for a series resulting to the search term "Concert" in the last 5 years (from Google Trends) and normalized between 0 and 1. Given a descriptor d, we measure RMSE for each zone $i \in Z_d$, namely $Err(d,i)$. The error is set to infinity for zones not in Z_d.

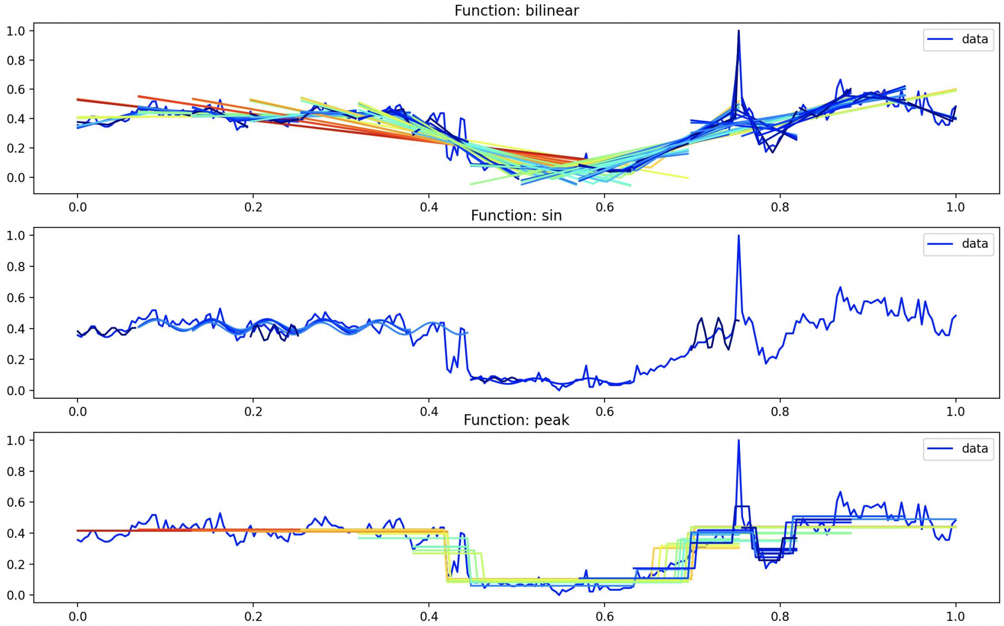


Figure 1: Example of three function prototypes fitting for 16 zones

2.2 Optimal descriptions covering

The problem of finding optimal descriptions that minimize the overall RMSE error is rather trivial and it is the first phase in the overall optimization process. In a practical scenario, we can expect thousands of description candidates (set $\mathscr{D}$) to be processed. They originate from a rather small set of function prototypes combined with a large set of sub-intervals of 2^ℓ to be investigated.

The goal is to represent the series through a set of descriptions $D \subseteq \mathscr{D}$ out of the description pool. Each description d has an associated set of zones Z_d, the curve type, optimal parameters and the RMSE $Err(d,i)$.

We wish to control the level of synthesis by an explicit parameter. In other DL approaches this is rather complex to handle, especially in case of rather long descriptions because of the incoherence in long text generation. We introduce the concept of *verbosity* ($v = |D|$) that defines the maximal number of descriptions to be employed to describe the series. The verbosity corresponds to the number of sentences used in the final narration.

The problem can be stated as follows. Given a verbosity v, select a set D_v such that $|D_v| = v$ and the sum of errors $Err(d,i)$ of each description $d \in D_v$ for each zone $i \in Z_d$ is minimal. We also ask for no zones overlaps ($Z_d \cap Z'_d = \emptyset$, for each $d, d' \in D_v$) and that each description covers at least $n/2^v$ zones.

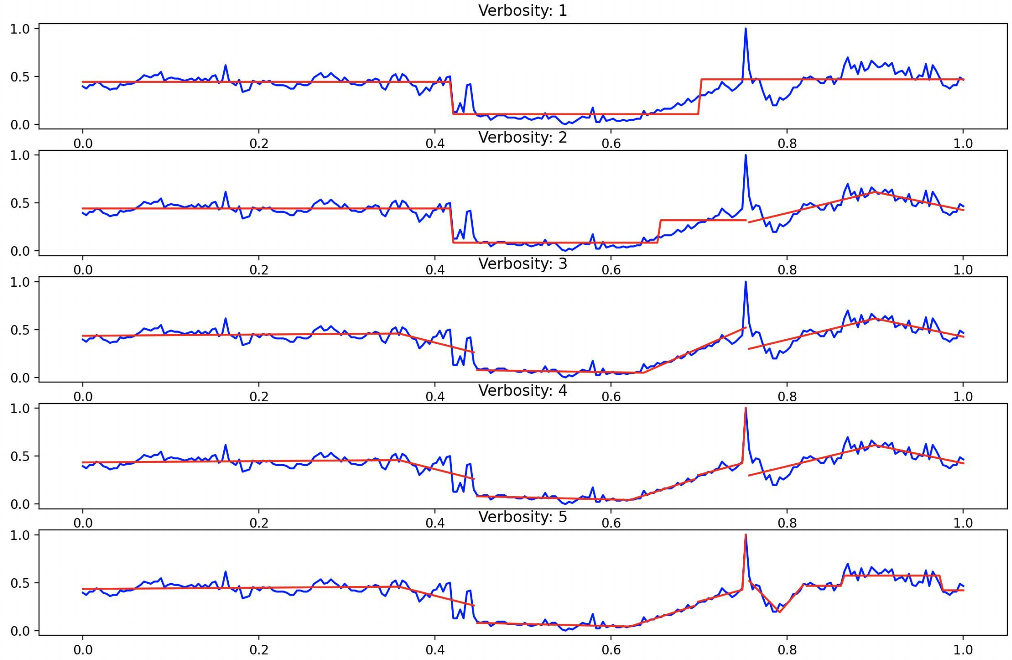


Figure 2: Example of optimal descriptions for verbosity 1–5

This last requirement ensures that for low verbosity the descriptions cover a sufficient time span, as a preference for general descriptions rather than extremely small details.

In Figure 2 it is shown, for the same dataset of Figure 1, the optimization results for $v \in [1..5]$. It can be noted that the red functions (descriptors) provide a better accuracy on details as more descriptors can be included. E.g., starting by a tooth function ($v = 1$), the sharp peak reaching 1.0 is better approximated with $v > 3$. Figure 3 shows in a black-red-yellow palette the RMSE for each zone (on the left, 16 zones, $\ell = 4$) and for each v. It can be noted how the error is significantly reduced by higher verbosity.

We believe that a fluent and effective description should find a balance between general synthesis and offered details. Details should be selected in order to avoid redundant information and/or no RMSE improvement w.r.t. larger descriptions. Finally, relationships among descriptions (hierarchical structure) could improve the structure of the narration. In conclusion, selecting a low (general description) and a high (details) verbosity, e.g., 1 and v, would result into an unnatural flat list of sentences, with the possible outcome of repetitions and loss of interesting details. Unfortunately, extending the previous model by these additional requirements creates a combinatorial explosion that can not be easily handled, essentially because every descriptor is a valid candidate.

2.3 The right balance between summary and details

We present the second optimization problem that models the creation of a summary and a set of details that describe the series only where necessary and accounting for the relationships among the other details.

We define *summary* descriptors as the ones that cover the series with a minimal verbosity and a specified accuracy. This selection allows to control the minimum accuracy for cases in which the fitting functions encounter a noisy dataset and/or a distribution very different from the prototypes. Formally, the set $S = D_s$ of summary descriptors is selected among the output of the previous optimization (sets

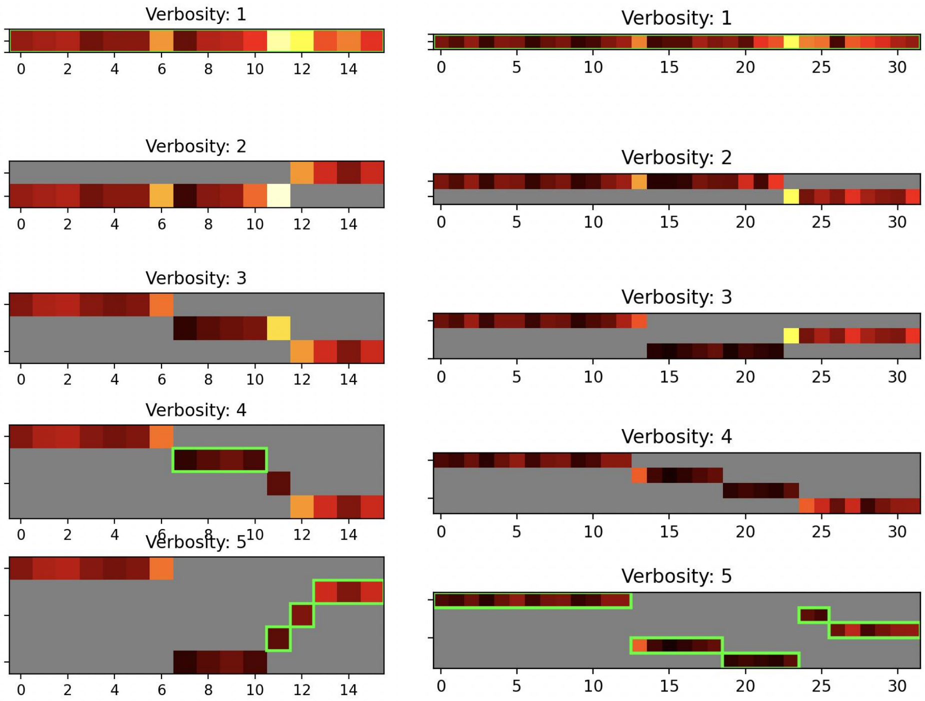


Figure 3: Example of optimal RMSE for verbosity 1–5 (left 16 zones, right 32 zones)

$D_1 \dots D_v$, for verbosity up to v). We select the minimal verbosity level s such that a threshold on maximal RMSE allowed (max_thr) is satisfied by all zones: $Err(d,i) < max\_thr$ for $i \in Z_d$ and $d \in S$.

The decision variables of the optimization are the *detail* descriptors. We ask for at most v details that will compose the set of details of the narration. In principle, such details could be taken from D_v (as mentioned above), being $|D_v| = v$. We prefer selecting details from $\cap_{i \le v} D_i$, with the introduction of some constraints among them and a cost function that accounts for accuracy improvements. The inclusion of lower verbosity levels allows us to consider coarser descriptions of details that may be suitable for narration as well as a more hierarchical narration structure that covers some overlaps. Moreover, some details from D_v can be redundant and other aspects captured by lower verbosity could be introduced. Finally, comparing different verbosity configurations, the introduction of details can be balanced in terms of minimal amount of RMSE improvement as opposed to the cost of the introduction itself.

Let us start modeling the introduction of details. If two descriptions (including summary) taken from two different verbosity intersect on some zones, i.e., there is an additional more detailed description, there must be a minimal RMSE improvement on any zone in order to be willing to talk about the detail. This threshold (min_thr) ensures that a new detail is introduced for a minimal improvement. Formally, $d \in D_i$ and $d' \in D_j$ with $i < j \; \exists z. z \in Z_d \cap Z'_d, Err(d,z) - Err(d',z) > min\_thr$.

Another constraint is that if a detail d is selected, it can not be completely covered in all of its zones Z_d by other details of higher verbosity. In this case d would be completely re-worded and thus redundant in the final textual description.

Finally, we design a maximization problem with a cost function related to the amount of RMSE improvement provided by details. Ideally, we try to select those details that improve the RMSE coverage of the summary. In particular, we focus on each zone. If a zone i is covered by at least a detail, the cost is the difference between the best and the worst $Err(d,i)$ $\forall d.i \in Z_d$. We also reduce the cost by a tiny penalty p proportional to the number of details that cover the same zone ($p = |d.i \in Z_d|$) in order to favor more distributed coverings.

Let us conclude with a consideration about the decision variables size. The current model needs only $\Sigma_{i \le v}|D_i| = v(v-1)/2$ candidates. We believe that exploiting this hierarchy provided by verbosity levels is rather beneficial for an efficient selection of most relevant descriptors. If the two optimization problems were joined, the decision variables would become in the order of thousands, depending on the number of zones and function prototypes.

2.4 Towards natural language

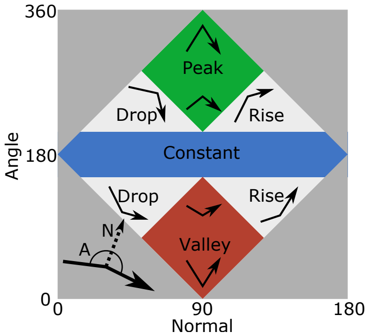
We present now the final processing performed by the last ASP program. The goal is to create a narration structure that orders and relates the descriptions computed by the last optimization and to retrieve qualitative properties associated to descriptions. This ASP program allows a single Answer Set and it does not require any optimization. This crucial step allows to clearly define the properties and rules in use, to deliver a transparent program.

The paragraph structure is retrieved by computing lexicographic ordering of summary and details descriptions, based on zones coverage. A `narration` predicate enumerates properties associated to a description. In particular it stores description location inside the narration order, grammar connective relation with next description (immediate/separated), inclusion relation with a description from lower verbosity and specific qualitative properties.

The larger part of this ASP code is devoted to the conversion of a description into a set of qualitative properties (accompanied by some quantitative measures). Each function prototype requires a dedicated ASP fragment that produces a `shape` property that describes the type of function in relation to its parameters. Moreover, it classifies qualitatively the extension of the shape.

As an example, let us show in Figure 4, on the left, the mapping of parameters for a description made of two consecutive segments into an interpreted shape. In the lower left gray corner we depict two consecutive segments, characterized by a normal N angle and an aperture angle A. Given the constraints of the three points forming the segments, $N \in [0..180]$ degrees and $A \in [0..360]$ degrees. We report the case when the two segments are rather similar in length; other cases are approximated by a single main line. As an example, the combination $N = 90$ and $A = 270$ would result in a peak (green area) with an smaller angle of 90 degrees. We depict the arrangements of the two segments at various combination of the two angles. Gray areas represent impossible combinations, since one of the two segments would invert the time order along x-axis. The blue area is interpreted as rather constant shape, being A similar to 180. Adverbs that report on the quality the shape (e.g., rather constant, moderately constant, etc.) can be generated and the choice of intensity can be modulated by distances from optimum. Red and green areas represent valleys and peaks, with a rather symmetric normal. White areas represent drop and rise lines, with different flavours. Also, qualitative amount of shape extension along y axis is associated to an ordered list of adjectives (e.g., very mild, mild, steep, very steep) that are selected by quantizing the measures and mapping them on the list. In the same Figure, on the right, we report a fragment of ASP encoding for valley detection with conversion of the numerical quantification of its depth to a qualitative description (`Strength` in the last atom in the body of the first rule).

This *modus operandi*, even if crafted for each function prototype, can result in a robust general library. From the explainability point of view, the error introduced from translating a description (still a mathematical function with associated RMSE) to a qualitative narration, even if not easily measurable, it can be manually assessed by ASP code analysis and conversion maps like the one shown in Figure 4.



```
select_bilin_type(ID,Angle,Normal,valley,Strength):-
      angle(Angle), angle2(Normal),  sol_desc(ID,_),
      |Angle-180|>=30,                  %%% not too flat
      Angle<180,       %%% there is a second order zero
      Normal+Angle/2 < 170,            %%% valley region
      10 < Normal+Angle/2,             %%% valley region
      adj_smooth_sharp(Strength,180-Angle,180).

adj_smooth_sharp(very_smooth,V,Max):-   V*6<Max.
adj_smooth_sharp(smooth,V,Max):-        V*6>=1*Max, V*6<2*Max.
adj_smooth_sharp(rather_smooth,V,Max):- V*6>=2*Max, V*6<3*Max.
...
```

Figure 4: Example of mapping parameters (normal and angle) to type of curve

2.5 Narration of descriptions

Once the narration structure is output by last ASP program, the facts are ready to be embedded in a text generation.

As a proof of concept, we used a Python procedure that follows the structure and merges strings, to show that the only problem left is the actual string processing. Connectives are handled according to the information received and notably, no processing of relevant adjectives and nouns is performed anymore. This is to show that with minimal imperative intervention is possible to produce the text. A more fluent version could be reached if grammar rules, variants, style rules and synonyms would be implemented in a more general ASP program. This is planned as future work.

Another positive side effect of our processing is the capability of plotting an enriched and summarized version of the series, based on the selected descriptions, as shown in Figure 5. On the left we depict the summary function (in this case a valley) and on the right we highlight some peaks and valleys. On the bottom of the figures there is a red-green colored bar that reports the relative errors of each zone (16 for this example), where the red value corresponds to *max_thr* (0.15 in this examples).

The text associated to the Figure 5 is:

```
In general, the series presents a very deep valley reaching an average of 0.09 between
0.42 and 0.69 out of a general average of 0.42 among the whole dataset.  In detail, a
rather sharp valley reaching a value of 0.24 occurs at 0.69; followed by a very steep
increase reaching a value of 0.99 at 0.75; then by a very sharp valley reaching a
value of 0.45 at 0.81; and finally by a sharp peak reaching a value of 0.39 at 1.0.
```

Notice that, as depicted in Figure 5, in processing the series we used a normalized scale between 0 and 1 for the x axis. A further improvement could map positions to timestamps for a more fluent processing of ranges and positions in time.

3 Implementation

In implementing our tool, we took advantage of Potassco Clingo Python's API. The process is started by a Python script that downloads a tagged time series from Google Trends service.

The main program consists of a Python configurator that sends the descriptor facts to the Control object for grounding jointly with the first optimization ASP program. For the control structure of the first optimization program, we opted for an explicit branch and bound inner loop, since we want to recover

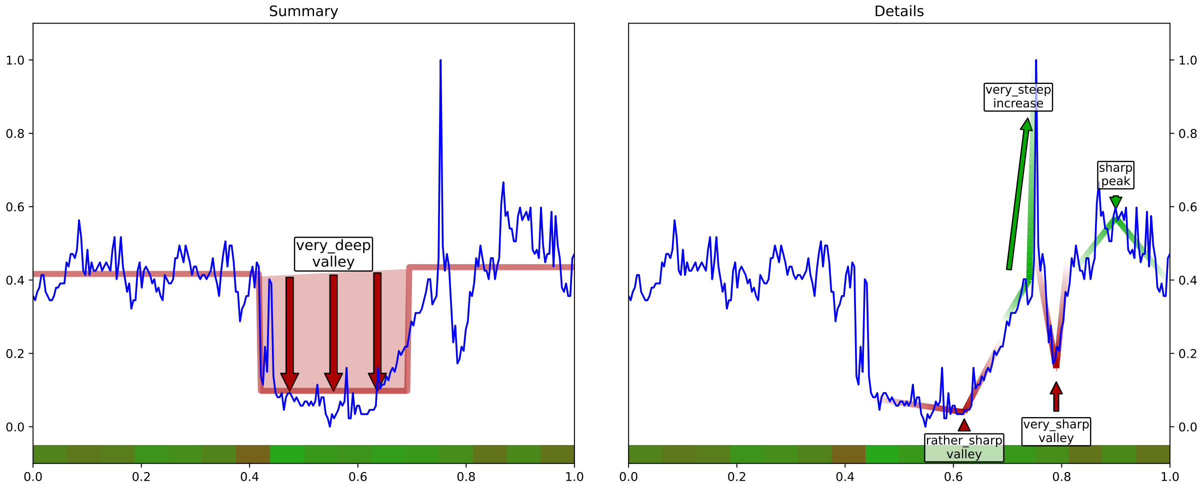


Figure 5: Example of enriched chart. On the left the Summary and on the right the Details

optimal covering at different verbosity levels. As soon as a solution is found, the verbosity value is incremented by means of an `external` feature in the outer loop.

The second optimization and description structured are located on the same ASP program, while each function prototype conversion is stored on a separate ASP file.

Results are plotted with Matplotlib and enriching visual description are computed directly from the description structure output by the program.

The code is available at `ahead-lab.unipr.it/files-for-iclp2023/`

4 Results

Let us present some results and discuss the impact of some parameters changes. Defaults parameters are $max\_thr = 0.15$ and $min\_thr = 0.02$.

4.1 Impact of zones on accuracy

The number of zones clearly affects the quality of fitting functions as well as the number of descriptors. We compare the results in terms of overall accuracy of the narration (the sum of best contributions for each zone position, as optimized by the second program). We compare 16 vs 32 zones, that give rise to 408 and 1584 descriptors respectively (we used 3 functions prototypes).

Starting with our running example of “Concert” series, we can see on Figure 3 the *heatmaps* for 16 (left) and 32 (right) zones as results of the first optimization. In general colors for 32 zones are darker, meaning that descriptors are more accurate. Measuring the global RMSE (the sum for each position of the best error covered by any selected description) gives a metric to judge the accuracy of the output. Recall that the optimization goal included a penalty for overlapped descriptions. Therefore, global RMSE could be improved with a cost of more awkward narration. Nevertheless, penalties are rather low and therefore global RMSE is not too distant from the optimization cost.

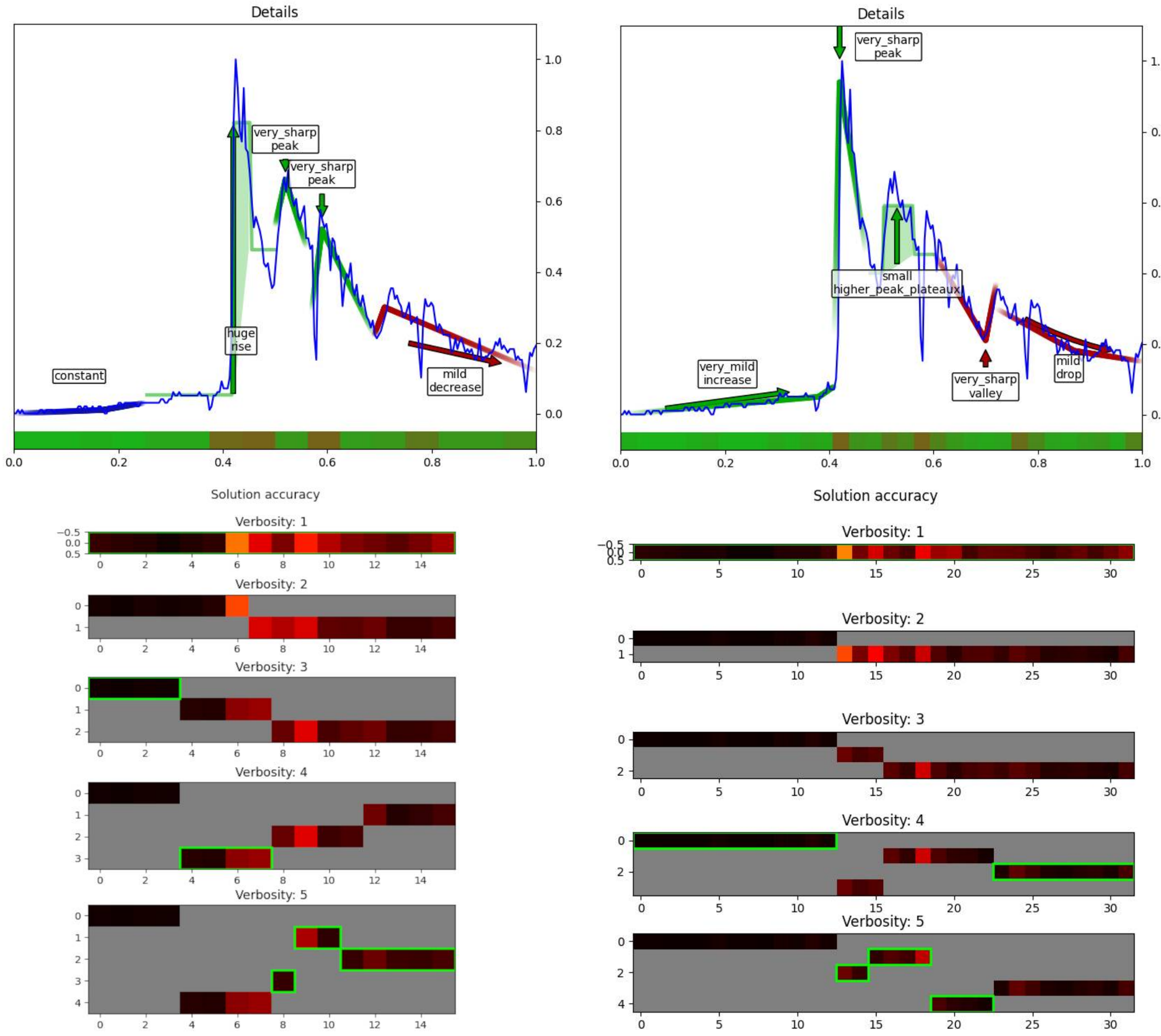


Figure 6: Comparison between 16 zones (left) and 32 zones (right). On top details on the enriched plot and at bottom the heatmaps with accuracy of descriptor candidates

Solution with 32 zones allows a selection of better descriptors and the ones selected by second optimization improve the global RMSE (0.041 with 16 zones and 0.036 with 32 zones). The global RMSE for 8 zones is 0.048, which is still acceptable, given the ability of descriptors to find optimal fitting inside the zone. However, when data has interesting patterns across zones, the ability of capturing them is lost if too few zones are used. We found that a good compromise could be 16 zones.

From a computational time point of view the complete program wall time, including function fitting (80% of the time) is a couple of seconds for 8 zones, around 30 seconds for 16 zones and around 2 minutes for 32 zones on a standard laptop.

Let us present a second test case where the “microsoft teams” search term was retrieved on Google Trends. In this case global RMSE for verbosity 5 is 0.043 (8 zones), 0.030 (16 zones) and 0.024 (32 zones). In Figure 6 we depict on top the two detail results for 16 zones (left) and 32 zones (right). It can be noticed that more accurate zones allow to better describe the last two features (valley and drop on the right) rather than corresponding a mild decrease (left). In order to maintain the number of descriptions equal to 5, there is an introduction of a small peak plateau (left) in place of two sharp peaks (right). The overall RMSE is improved, as depicted on the green-red bar at the bottom of the chart. The two heatmaps at the bottom of the figure show the accuracy with a black-red-yellow scale depending on the zones. The descriptors selected by the second optimization are colored green.

4.2 Verbosity comparison and text generation

We show here the impact of verbosity, which is the most influential control for sentence summary.

We setup another test case with the series obtained with the search term “blockchain” from Google Trends (last 5 years) with 16 zones. We launched the program with verbosity 3, 4, and 5. Figure 7 depicts the corresponding enriched plots. On the top left the summary, which is common to all solutions, presents the main description (a deep plateau). On the top left we show verbosity 3 with the presence of an initial drop, a narrow peak and a sharp peak. It is interesting to note on bottom left (verbosity 4) that the initial drop is better approximated by a sequence of two steep drops. At their intersection a new peak is determined. As future work we plan to implement the abstraction of consecutive functions, which could result in a higher level identification of other patterns as post processing. Finally, at bottom right the introduction of another description captures the last emerging peak at the end of the series.

Let us report here the text generation from the selected descriptors.

Verbosity 3:
```
In general, the series presents a very deep lower peak plateau reaching an
average of 0.12 between 0.16 and 0.57 out of a general average of 0.45 among the whole
dataset.  In detail, a steep drop reaching a value of 0.05 occurs at 0.44; followed
by a very high peak reaching an average of 0.59 between 0.58 and 0.66 out of a general
average of 0.17 between 0.44 and 0.69; and finally by a sharp peak reaching a value of
0.1 at 1.0.
```

Verbosity 4:
```
In general, the series presents a very deep lower peak plateau reaching an
average of 0.12 between 0.16 and 0.57 out of a general average of 0.45 among the whole
dataset.  In detail, a very steep decrease reaching a value of 0.32 occurs at 0.25;
followed by a steep drop reaching a value of 0.04 at 0.44; then by a very high peak
reaching an average of 0.59 between 0.58 and 0.66 out of a general average of 0.17
between 0.44 and 0.69; and finally by a sharp peak reaching a value of 0.1 at 1.0.
```

Verbosity 5:
```
In general, the series presents a very deep lower peak plateau reaching an
average of 0.12 between 0.16 and 0.57 out of a general average of 0.45 among the whole
dataset.  In detail, a very steep decrease reaching a value of 0.32 occurs at 0.25;
followed by a steep drop reaching a value of 0.04 at 0.44; then by a very high peak
reaching an average of 0.59 between 0.58 and 0.66 out of a general average of 0.17
between 0.44 and 0.69; a very sharp peak reaching a value of 0.36 at 0.88; and finally
by a very steep decrease reaching a value of 0.13 at 1.0.
```

4.3 Conclusion

As future work we plan to optimize the Python fitting pre-processing, being the most time demanding (several seconds). The goal is to optimize the system to deliver almost real time answers. We also plan to develop a comprehensive interpretation of consecutive descriptions, with the goal of inferring additional descriptions. In this case, some higher-level description could highlight interesting information that emerges, e.g., if two sharp peaks are adjacent, a valley is probably in between. Another ASP program could take care of the translation of the narration structure into natural language, porting some Logic Programming ideas already appeared in the literature. In the results, we showed a simple translation

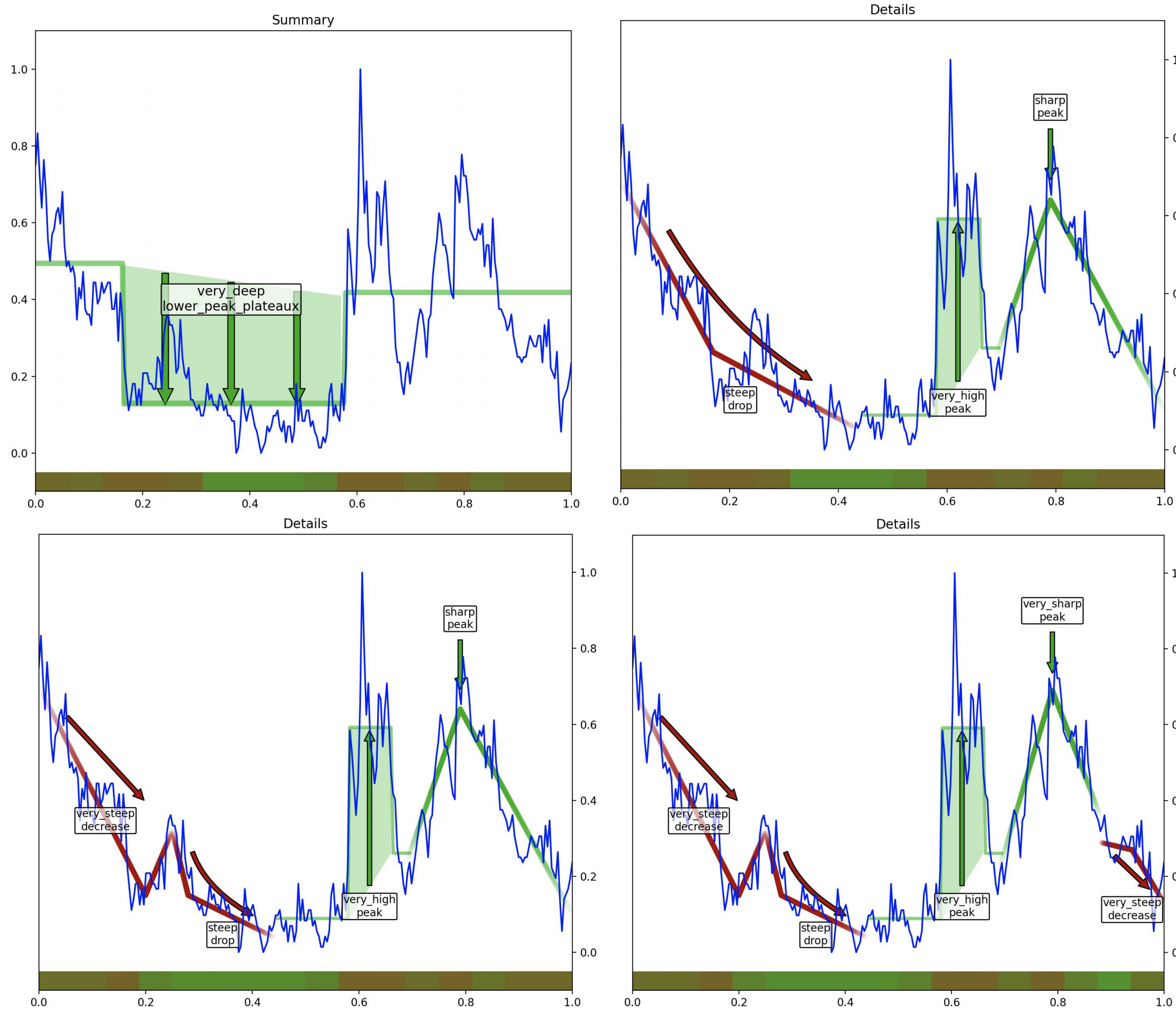


Figure 7: Enriched charts with different verbosity (3 = top right, 4 = bottom left, 5 = bottom right). At top left the summary, shared by all solutions.

to natural language in Python, but a richer rule-based model (e.g., a DCG in Prolog) could handle the variants and nuances of a natural language in a more elegant and effective form.

In general, we plan to move from single time series and to extend the work in order to handle multiple and synchronized time series. Such analysis could be extended to high level interpretation of multiple source events that could be described by an ontology. This could result in critical systems monitoring (e.g., in healthcare), where explainability is fundamental. Moreover, comparisons of series as well as comparisons of multidimensional data are other domains for future investigation.

In conclusion, the paper presented an explainable methodology that produces a natural language text, coming from a structured organization of descriptions, as result of the interpretation of a time series. Reasoning about descriptions is ruled by an ASP core (transparent and domain-independent): choices about *what to say* depend on an optimization problem that is controlled by an ASP concise and transparent program; the definition of *how to say* is handled by an ASP program that organizes descriptions into a narration structure, assigns qualitative evaluations and is able to abstract low level features into more general ones. The output is associated to a RMSE measure on accuracy of each sentence; the conversion from descriptors to qualitative interpretation is described by means of a logic program; full control on minimal and maximal errors is provided; verbosity is a key input parameter that models the whole process; the narration structure is handled by a logic program and its description can be easily converted to natural language and to enriched graphs.

References

[1] A. Adadi & M. Berrada (2018): *Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI)*. *IEEE Access* 6, pp. 52138–52160, doi:10.1109/ACCESS.2018.2870052.

[2] A. Balaji, T. Ramanathan & V. Sonathi (2018): *Chart-Text: A Fully Automated Chart Image Descriptor*. *CoRR* abs/1812.10636. Available at `https://arxiv.org/abs/1812.10636`.

[3] A. Barredo Arrieta, N.D. Rodríguez, J. Del Ser, A. Bennetot, S. Tabik, A. Barbado, S. García, S. Gil-Lopez, D. Molina, R. Benjamins, R. Chatila & F. Herrera (2020): *Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI*. *Inf. Fusion* 58, pp. 82–115, doi:10.1016/j.inffus.2019.12.012.

[4] European Commission (2021): *Proposal for a Regulation laying down harmonised rules on artificial intelligence*. Available at `https://digital-strategy.ec.europa.eu/en/library/proposal-regulation-laying-down-harmonised-rules-artificial-intelligence`.

[5] M. Gebser, R. Kaminski, B. Kaufmann & T. Schaub (2012): *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers.

[6] H. Harkous, I. Groves & A. Saffari (2020): *Have Your Text and Use It Too! End-to-End Neural Data-to-Text Generation with Semantic Fidelity*. In D. Scott, N. Bel & C. Zong, editors: *Proc. of the 28th International Conference on Computational Linguistics*, Int. Committee on Computational Linguistics, Barcelona, Spain (Online), pp. 2410–2424, doi:10.18653/v1/2020.coling-main.218.

[7] K. Kukich (1983): *Design of a Knowledge-Based Report Generator*. In M.P. Marcus, editor: *21st Annual Meeting of the Association for Computational Linguistics, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, June 15-17, 1983*, ACL, pp. 145–150, doi:10.3115/981311.981340.

[8] T. Liu, K. Wang, L. Sha, B. Chang & Z. Sui (2018): *Table-to-Text Generation by Structure-Aware Seq2seq Learning*. In S.A. McIlraith & K.Q. Weinberger, editors: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, AAAI Press, pp. 4881–4888, doi:10.1609/aaai.v32i1.11925.

[9] J. Meszaros, J. Minari & I. Huys (2022): *The future regulation of artificial intelligence systems in healthcare services and medical research in the European Union*. *Frontiers in Genetics* 13, doi:10.3389/fgene.2022.927721. Available at `https://www.frontiersin.org/articles/10.3389/fgene.2022.927721`.

[10] S. Murakami, A. Watanabe, A. Miyazawa, K. Goshima, T. Yanase, H. Takamura & Y. Miyao (2017): *Learning to Generate Market Comments from Stock Prices*. In R. Barzilay & M. Kan, editors: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, ACL, pp. 1374–1384, doi:10.18653/v1/P17-1126.

[11] J. Obeid & E. Hoque (2020): *Chart-to-Text: Generating Natural Language Descriptions for Charts by Adapting the Transformer Model*. In B. Davis, Y. Graham, J.D. Kelleher & Y. Sripada, editors: *Proceedings of the 13th International Conference on Natural Language Generation, INLG 2020, Dublin, Ireland, December 15-18, 2020*, Association for Computational Linguistics, pp. 138–147. Available at `https://aclanthology.org/2020.inlg-1.20`.

[12] F. Portet, E. Reiter, A. Gatt, J. Hunter, S. Sripada, Y. Freer & C. Sykes (2009): *Automatic generation of textual summaries from neonatal intensive care data*. *Artif. Intell.* 173(7-8), pp. 789–816, doi:10.1016/j.artint.2008.12.002.

[13] R. Puduppully, L. Dong & M. Lapata (2019): *Data-to-Text Generation with Content Selection and Planning*. In: *The 33rd AAAI Conference on Artificial Intelligence, AAAI 2019, The 31st Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The 9th AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, AAAI Press, pp. 6908–6915, doi:10.1609/aaai.v33i01.33016908.

[14] C. Rudin (2019): *Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead*. *Nature Machine Intell.* 1(5), pp. 206–215, doi:10.1038/s42256-019-0048-x.

[15] USA Various Agencies (2016-2021): *USA National Artificial Intelligence Initiative*. Available at `https://www.ai.gov/`.

[16] S. Wiseman, S.M. Shieber & A.M. Rush (2017): *Challenges in Data-to-Document Generation*. In M. Palmer, R. Hwa & S. Riedel, editors: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, Association for Computational Linguistics, pp. 2253–2263, doi:10.18653/v1/D17-1239.

Deontic Paradoxes in ASP with Weak Constraints[*]

Christian Hatschka Agata Ciabattoni Thomas Eiter
Institute of Logic and Computation, TU Wien, Vienna, Austria
`firstname.lastname@tuwien.ac.at`

The rise of powerful AI technology for a range of applications that are sensitive to legal, social, and ethical norms demands decision-making support in presence of norms and regulations. Normative reasoning is the realm of deontic logics, that are challenged by well-known benchmark problems (deontic paradoxes), and lack efficient computational tools. In this paper, we use Answer Set Programming (ASP) for addressing these shortcomings and showcase how to encode and resolve several well-known deontic paradoxes utilizing weak constraints. By abstracting and generalizing this encoding, we present a methodology for translating normative systems in ASP with weak constraints. This methodology is applied to "ethical" versions of Pac-man, where we obtain a comparable performance with related works, but ethically preferable results.

1 Introduction

Norms, which involve concepts such as obligation and permission, are an integral part of human society. They are enormously important in a variety of fields – from law and ethics to artificial intelligence (AI). In particular, imposing norms – be they ethical, legal or social – on AI systems is crucial, as these systems have become ubiquitous in our daily routines.

A main difference between norms and other constraints lies in the fact that norms typically allow for the possibility of violation. Reasoning with and about norms (*normative reasoning*) requires deontic logic, the branch of logic that deal with obligation and related concepts. Normative reasoning comes with a variety of idiosyncratic challenges, which are often exemplified by benchmark examples (so called deontic paradoxes). A crucial challenge is reasoning about sub-ideal situations, such as *contrary-to-duty (CTD) obligations*, which are obligations only triggered by a violation. Other challenges are associated, e.g., with defeasibility issues (norms having different priorities, exceptions, etc.).

The first deontic system introduced – Standard Deontic Logic [27] – was failing most of the benchmark examples, (un)deriving formulas which are counterintuitive in a common-sense reading. This has motivated the introduction of a plethora of deontic logics, see, e.g. [10]. These logics have been investigated mainly in connection with philosophy and legal reasoning, and with the exception of Defeasible Deontic Logic DDL [14, 15], they lack defeasibility and efficient provers. Defeasibility and efficient reasoning methods are instead offered by Answer Set Programming (ASP), which is one of the most successful paradigms of knowledge representation and reasoning for declarative problem solving [7]. Indeed, in a long and systematic effort of the knowledge representation community, efficient solvers for fast evaluation of ASP programs have been developed, see, e.g., [18]. Defeasibility is also inherent in ASP, due to its default-negation and also weak constraints. This paper introduces a method for using weak constraints for encoding norms in ASP. We first translate desired basic properties of deontic operators in a common core that will be used in all further encodings. These properties are established by analyzing multiple well-known deontic paradoxes (e.g., *Ross Paradox*, the *Fence Scenario*,...). By abstracting and generalising

---

[*] Work partially supported by the WWTF project TAIGER (ICT22-026).

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 367–380, doi:10.4204/EPTCS.385.39

the encodings of the specific paradoxes, we provide a methodology for encoding normative systems in ASP with weak constraints. The methodology is put to work on a case study from [23, 22] that involves a reinforcement learning agent playing a variant of the Pac-man video game with additional "ethical" rules. Our encoding is used as a "shield" to filter out the non compliant actions of Pac-man and the outcome is compared with [22], that uses DDL. For space reason, we must omit details, for which we refer to [16].

2 Preliminaries

Answer Set Programming. We consider extended logic programs with disjunction [11, 20], which are finite sets of rules r (henceforth referred to as programs)

$$H_1 \vee \ldots \vee H_l \,{:}{-}\, A_1, \ldots A_n,\ not\ B_1, \ldots,\ not\ B_m. \quad l, m, n \geq 0, \tag{1}$$

where all H_i, A_j, and B_k are literals in a first-order language. Here *not* denotes weak (default) negation and $\neg$ (also written $-$) strong negation. Informally, r can be read as: "If all A_i are true and for no B_j there is evidence that it is true, at least one of $H_1, \ldots, H_l$ must be true".

The answer sets of a ground (variable-free) program Π are given in terms of consistent sets S of ground literals as follows. Let $S \models r$ denote that $S \models B(r)$ implies $S \models H(r)$, where $S \models B(r)$ denotes that $\{A_1, \ldots, A_n\} \subseteq S$ and $\{B_1, \ldots, B_m\} \cap S = \emptyset$, and $S \models H(r)$ denotes that $\{H_1 \ldots, H_l\} \cap S \neq \emptyset$. S is an answer set of Π if S satisfies all rules r in $\Pi(S) = \{r \in \Pi \mid S \models B(r)\}$, and no $S' \subset S$ satisfies $\Pi(S)$. The answer sets of a general program Π are those of its grounding $grd(\Pi)$ that consists of all ground instances of its rules. For modeling defeasibility of obligations, we use weak constraints of the form

$$:\sim A_1, \ldots, A_n.\ [w : l]$$

(as in the system DLV[20]) where $A_1, \ldots, A_n$ are literals (that may be weakly negated) and $w, l \geq 0$ are the weight and the (integer) level of the weak constraint, respectively. Informally, weak constraints single out optimal answer sets that have minimal total weights of violated weak constraints with higher levels being more important; see [20] for formal definitions and more background.

SDL. Standard Deontic Logic [27], is the best known system of deontic logic.

SDL formulas are constructed by the following grammar ($\mathscr{A}$ is the set of atomic propositions):

$$\varphi := p \in \mathscr{A} \mid \neg\varphi \mid (\varphi) \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \to \varphi \mid O\varphi \mid P\varphi \mid F\varphi$$

SDL is a monadic deontic logic, as the operators O (obligation), P (permission) and F (prohibition) apply to single formulas; they are read as "it is obligatory that φ", "it is permissible that φ", and "it is forbidden that φ", resp., and inter-definable, e.g., $P\varphi := \neg O(\neg\varphi)$, and $F\varphi := O\neg\varphi$.

The semantics of SDL – also known as the modal logic KD – is based on possible worlds, where the accessibility relation is serial. A Hilbert system for SDL is obtained by adding the following axiom-schemata and rules to any axiomatization of classical propositional logic:

If φ is a theorem, $O\varphi$ is a theorem (RND)

$O(\varphi \to \psi) \to (O\varphi \to O\psi)$ (KD) $O\varphi \to \neg O \neg \varphi$ (DD)

In the following we will also use the derived axiom and rule:

If $\varphi \to \psi$ is a theorem, $O\varphi \to O\psi$ is a theorem (RMD) $\neg O \bot$ (OD)

For more details about SDL and other deontic logics, see, e.g., [10]

3 Deontic Paradoxes

As the foundation for our work, we examine several deontic paradoxes. These consist of (un)derivable formulas which are counter-intuitive when viewed from a common-sense perspective. While referred to in the literature as paradoxes, many of them are not paradoxes *per se*, but rather puzzles or dilemmas. Deontic paradoxes play an important role in deontic logic and normative reasoning; they serve as sanity checks for existing systems, and as driving force for defining new systems. In particular, they exemplify that SDL fails to capture the nuances of normative reasoning expected in certain scenarios. There are many such paradoxes. We categorise them according to the reason for their failure, see e.g. [17, 10], and analyze one example for each class:

1. Paradoxes centering around RMD: *Ross's Paradox*, *Good Samaritan Paradox*, *Åqvist's Paradox of Epistemic Obligation*
2. Puzzles centering around DD and OD: *Sartre's Dilemma*, *Plato's Dilemma*
3. Puzzles centering around deontic conditionals: *Broome's Counterexample*, *Chisholm's Contrary-to-Duty Paradox*, *Forrester's Paradox*, *Considerate Assassin Paradox*, *Asparagus Paradox*, *Fence Paradox*, *Alternative Service Paradox*

Deontic conditionals refer to obligations that arise situationally; written as $O(A/B)$ (to be read as "A is obligatory if B") they have been introduced to cope with contrary-to-duty obligations, i.e., obligations which come into force when another obligation is violated.

Paradoxes centering around RMD show, in general, that SDL is too strong as it derives unwanted consequences. An example is *Ross's Paradox*, which consists of the following two sentences:

It is obligatory that the letter is mailed. (R1)
It is obligatory that the letter is mailed or burned. (R2)

Let $O(m)$ and $O(m \vee b)$ formalize (R1) and (R2), respectively. As $m \to (m \vee b)$ is a theorem in SDL, $O(m \vee b)$ follows from $O(m)$ by RMD and modus ponens. But it seems counterintuitive to derive an obligation that is satisfied by burning the letter, when failing to mail the letter.

Puzzles centering around DD and OD involve conflicting obligations that cannot be obeyed. An example is *Plato's Dilemma*:

It is obligatory that I meet my friend for dinner. (P1)
It is obligatory that I rush my child to the hospital. (P2)

Clearly, it is not possible to satisfy both obligations at the same time. Using common sense reasoning, P2 should override P1, but in SDL the presence of two contradictory obligations would make everything obligatory (deontic explosion).

Puzzles centering around deontic conditionals have as a prominent example the *Fence Scenario* [24], which combines two different weaknesses of SDL regarding CTD obligations and exceptions:

There must be no fence. (F1)
If there is a fence then it must be a white fence. (F2)
If the cottage is by the sea, there may be a fence. (F3)

Figure 1: The common core of our encodings

$$O(X) \lor -O(X) \mathbin{:-} act(X). \tag{1}$$
$$F(X) \lor -F(X) \mathbin{:-} act(X). \tag{2}$$
$$\mathbin{:-} O(X), -Dia(X). \tag{3}$$
$$-Dia(X) \mathbin{:-} -Do(X), act(X). \tag{4}$$
$$\mathbin{:-} O(X), F(X). \tag{5}$$
$$Do(X) \lor -Do(X) \mathbin{:-} act(X). \tag{6}$$
$$\mathbin{:-} F(X), Do(X). \tag{7}$$
$$Happens(X) \mathbin{:-} Do(X). \tag{8}$$
$$\mathbin{:-} Do(X), -Dia(X). \tag{9}$$
$$:\sim O(X).[1:1] \tag{10}$$
$$:\sim F(X).[1:1] \tag{11}$$

Here (F2) serves as a CTD obligation that is active when (F1) is violated, while (F3) serves as an exception to (F1). Note that under this interpretation, if the cottage is by the sea the fence need not be white. The contrary-to-duty obligation (F2) cannot be formalised in SDL: having a white fence implies having a fence, and by (RND) the obligation to have a fence; this contradicts (F1). Moreover, as SDL lacks expressing defeasibility, (F3) cannot be properly formalized.

4 Encoding the Paradoxes

We now proceed to encode the paradoxes from above All encodings share the same common core, shown in Figure 1, that encodes properties of SDL, using the following predicates:

- $O(X)$ resp. $F(X)$ denotes that X is obligatory resp. forbidden;
- $act(X)$ denotes that X is eligible for reasoning about whether X is obligatory or not. While an action by default, in some cases X may not materialize but viewed as such. An example from above would be owning a white fence, as we reason about whether it is obligatory.
- $Do(X)$ denotes that the agent has chosen to take the action X, and $-Do(X)$ denotes that the agent will definitely not take the action X.
- $Dia(X)$ is an auxiliary predicate to denote that an action X is an option resp. possible (in the sense on modal logic). Thus, $-Dia(X)$ can either mean that the agent cannot take the action or that the agent has chosen not to take the action.
- $Happens(X)$ is an auxiliary predicate that denotes an event X happening. It is sometimes used in encodings to denote events happening which are usually outside the agents control.

Intuitively, the common core guesses whether something is obligatory (1), forbidden (2) and whether the agent takes the action (6). The remaining rules then encode connections between predicates and exclude answer sets that we deem inconsistent, e.g., something being obligatory and forbidden or something being obligatory and that action not taken. The weak constraints (10) and (11) are used to eliminate answer sets that derive obligations/prohibitions with no need.

We note that the common core is not a faithful encoding of full SDL and its axioms: while theoretically possible, this would be undesired as SDL axioms lead to multiple paradoxes. Instead, the common core captures some of the SDL axioms, while satisfactory handling the deontic paradoxes. For instance

Proposition 1. *The SDL axiom DD holds in the common core.*

Proof. *DD* is formalized in rule (5), that forbids an action from being both forbidden and obligatory, as $O\varphi \to \neg O \neg \varphi$ which is equivalent to $\neg O\varphi \lor \neg F\varphi$, and to $\neg(O\varphi \land F\varphi)$. ☐

The idea behind the common core is to generate all maximal sets of non-auxiliary predicates that are consistent and filter out suboptimal answer sets using weak constraints. We require that in a consistent set an action is not obligatory and forbidden at the same time and any obligatory action is taken (resp. any forbidden action is not taken). For instance, disregarding the auxiliary predicates *Dia*, *Happens* and *act*, the following are all maximal consistent sets of deontic predicates:

$$\{O(action), -F(action), Do(action)\}, \{F(action), -O(action), -Do(action)\},$$
$$\{-F(action), -O(action), -Do(action)\}, \{-F(action), -O(action), Do(action)\}$$

The next proposition can be seen as a soundness and completeness result for the common core.

Proposition 2. *The rules* (1)–(9) *from Figure 1 allow for all and only the maximal consistent sets of deontic predicates as answer sets.*

This can be seen by inspecting the answers sets output by a solver. Intuitively, soundness of the rules (1) to (9) is achieved, as inconsistent answer sets are excluded. Completeness on the other hand is given, as all answer sets that are considered consistent for an action are generated.

Note that in Prop. 2 the weak constraints (10) and (11) are excluded: as every answer set for them would represent an optimal way to handle given norms, they would eliminate every answer set containing obligations or prohibitions, and completeness would be lost. By including them, preference will be given to unrestricted behaviour, in interplay with possible further rules.

4.1 Paradox encodings

We describe how to extend the common core to encode selected paradoxes from above.

Ross's Paradox. In contrast with what happens in SDL, we do not want to derive (R2) in the ASP encoding. To achieve this, we add the following rule and facts to the core:

$$\boxed{:\sim -O(mail).\ [1:2] \quad (12) \qquad act(mail). \quad (13) \qquad act(burn). \quad (14)}$$

Note that a disjunction over obligations is represented by two different answer sets that each contain one possible way to satisfy the obligation over the disjunction.

The obligation $(R1)$ is created using the weak constraint (12), while the facts (13) and (14) declare *mail* and *burn* as actions to reason about. For *mail* (resp. *burn*) the core encoding guesses it as obligatory or not; (12) may penalise the guess at the highest level if the program does not entail mailing the letter as obligatory. As constraint violation at the highest level is minimised, each optimal answer set includes the obligation to mail the letter, should such an answer set exist.

The program has two answer sets; none of them derive the obligation to burn the letter, and they only differ for the choice of the agent to burn the letter or not. By adding a rule specifying that it is not possible to perform both acions: burn the letter and mail it, the answer set where the agent chooses to burn the letter would not be derived.

Plato's Dilemma. Recall that the desired outcome of this dilemma would be that the agent takes her child to the hospital, thereby violating the obligation of meeting her friend for dinner. The encoding presents two interesting aspects: prioritisation of the obligations and the impossibility of taking both actions. This can be encoded as follows:

$$\boxed{\begin{array}{llrl}
:\sim -O(help), Happens(emergency).[1:3] & (20) & act(help). & (23)\\
:\sim -O(meet).[1:2] & (21) & :- Do(help), Do(meet). & (24)\\
act(meet). & (22) & Happens(emergency). & (25)
\end{array}}$$

The weak constraint (20) is at level 3, the highest in this encoding, and penalises answer sets in which $Happens(emergency)$ is true but the obligation to help is not derived. In other words, it derives the obligation $(P2)$ to help the child in case of an emergency. The weak constraint (21) encodes the obligation $(P1)$ to meet the friend for dinner, but at a lower level (viz. 2), which gives priority to $(P2)$. The constraint (24) encodes the impossibility of taking both actions. With the assertions that an emergency occurs and *meet* and *help* are the possible actions, as desired, a single answer set exists containing the obligation to help the child.

Fence Paradox One might think that CTD obligations could be handled like exceptions to obligations. While one could accordingly state "There may be a fence if it is white", it would not have the same meaning as in the paradox. Handling a CTD obligation as an exception results in a loss the original obligation to a certain degree; it could in this case be seen as *the least thing to do to set things right*. While having a white fence improves the situation, the presence of the fence itself remains undesirable [24].

The important fact to consider is that should the cottage be by the sea, then as (F1) is not active due to (F3), the fence need not be white. To this end, we add the following to the common core:

$$:\sim -F(have\_fence), not\, Location(sea).\ [1:2] \qquad (30)$$

$$:\sim Do(have\_fence), not\, Location(sea), -O(have\_white\_fence).\ [1:2] \qquad (31)$$

$$act(have\_fence). \qquad (32)$$

$$act(have\_white\_fence). \qquad (33)$$

Here (30) caters for (F1) and (31) for (F2); notably, (F3) also affects the CTD obligation (F2). This is needed as the fence has to be white only when the cottage is not by the sea. Otherwise the obligation for the fence to be white would also be derived if the cottage was by the sea.

To check whether the obligation for the fence to be white is deduced when the cottage is by the sea (and we have a fence), we further add:

$$Location(sea). \qquad Do(have\_fence).$$

Then two answer sets exist; both do not derive the obligation for the fence to be white. When testing other constellations, the answer sets obtained also represent the expected results.

5 Generalisation and Methodology

We can classify the obligations that appeared in the paradoxes into the following classes:

- **Regular obligations:** These obligations should be followed as long as possible (without violating a more important obligation).
- **Conditional obligations:** These are obligations that only need to be followed given certain preconditions. E.g., the obligation to wear a suit when at a formal event.
- **Obligations over disjunctions:** obligations that are fulfilled by satisfying any disjunct that constitutes the obligation; e.g., bring dessert or salad.
- **Conjunctions of obligations that all need to be satisfied:** obligations consisting of multiple parts where satisfying all parts is necessary.
- **Obligations with exceptions:** obligations to be followed unless an exception is given.
- **Contrary-to-duty obligations:** obligations that arise as another obligation is violated.

Type of obligation	Encoding
Regular	$:\sim -O(o).\ [w:l]$
Conditional	$:\sim -O(o), condition.\ [w:l]$
Disjunction	$:\sim -O(o_1), -O(o_2), \ldots, -O(o_n).\ [w:l]$
Conjunction	$:\sim not\, Conj.\ [w:l]$
	$Conj : -O(o_1), \ldots, O(o_n).$
Exceptions	$:\sim -O(o),\ not\, Exception.\ [1:2]$
Contrary-to-duty	$:\sim -Do(o_1), -O(o_2).\ [1:2]$

Table 1: Encodings for different types of obligations

Note that prohibitions are viewed as regular *negative* obligations, i.e. the obligation not to do something. The different kinds of obligations are encoded as shown in Table 1. Their encoding uses weak constraints to model defeasibility. An obligation to take an action a that should always hold is encoded in the following way:

$$:\sim -O(a).\ [w:l]$$

Note that the weight w and the level l of the weak constraint depend on the importance of the obligation and conflicting obligations. In most cases, $w = 1$ and merely l is used to encode priorities among obligations. Conflicts between obligations are detected and the priority among obligations is established through weak constraints (more important obligations have higher level).

By generalising the encodings of the considered paradoxes (from Section 3), we propose the following encoding methodology that consists of the following steps:

Step 1. For each of the norms determine what kind of obligation it represents, among the six different kinds of obligations we have considered.

Step 2. Determine which actions are simultaneously incompatible. Knowing which actions are in conflict eases determining the importance of the obligations. Incompatibility needs to be determined through context. E.g., while in general it is possible to watch a movie while browsing the internet, these actions may be incompatible on an old smartphone.

Step 3. Encode the different kinds of obligations and their importance. Here weights as priorities play an important role. There are two cases we need to consider:

Case 1. An obligation (of whatever kind) is more important than the other. In this case, setting the level of one constraint higher than the other is sufficient. E.g., consider the case of two obligations o_1 and o_2 where the latter is more important. If o_1 and o_2 have no special properties, the encoding is: (with $j \geq 1$)

$$:\sim -O(o_1).\ [1:l] \qquad :\sim -O(o_2).\ [1:l+j]$$

Note that j may differ given additional obligations. We account for this as follows.

Suppose the incompatibilities between actions are given, as well as the importance of obligations by a preference $O' \succ O$ stating that the obligation O' is strictly more important than O. We generate a directed graph $G = (V, E)$ whose vertices $V = \{O_1, \ldots, O_n\}$ are the obligations having the edges $E = \{(O_i, O_j) \in V^2 \mid O_i \succ O_j\}$; note that G must be acyclic.[1] The sinks of G, i.e., vertices with no

---

[1] For Non-strict preference $\succeq$, we can use the supergraph of G, whose nodes cluster all equally preferable obligations.

outgoing arcs, are assigned priority $p_1 = 2$. After simultaneously removing all sinks, we iterate the process with increased priority, i.e., assign the new sinks priorities $p_2 = 3$, $p_3 = 4$ etc.; this results in a priorization by levels.

Case 2. Multiple obligations $o_1, \ldots o_n$ are in conflict with an obligation o. While satisfying o is better than satisfying a single o_i, satisfying multiple o_i's may be equally good or better than satisfying o. In this case, we can use the weights of the weak constraints for $o_1, \ldots, o_n$ to encode this: they must then correspond to their importance and o must have a weight that is equal or smaller than the joint weights.

For example, if o_1, o_2 and o_3 are mutually non-exclusive and equally important to o if all are satisfied, the following weights could be chosen, for a number $k \geq 1$ (the level is the same):

$$:\sim -O(o_1).\ [k:l] \qquad :\sim -O(o_2).\ [k:l] \qquad :\sim -O(o_3).\ [k:l] \qquad :\sim -O(o).\ [3k:l]$$

Step 4. Encode the exclusion of combinations of actions found incompatible. If two actions a_1 and a_2 are incompatible, this is encoded by adding the following constraint:

$$:-\,Do(a_1), Do(a_2).$$

Step 5. Encode additional information. This includes denoting constants as actions using the predicate *act*, and specifying dependencies between actions; e.g., if the action running entails the action moving, we add a rule

$$Do(move) :-\, Do(run).$$

Observation 1. *The common core simulates the SDL axioms RND and KD.*

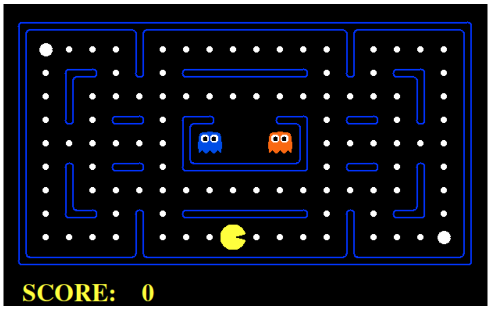
As there are no theorems in our framework, *RND* is not directly implemented. However, if we read a theorem as something than cannot be violated, *RND* becomes: "If it is impossible to violate φ, then φ is obligatory." In our semantics this obligation would not be derived; however the obligation to take the action φ is given indirectly as the agent must do φ (that cannot be violated). For *KD*, recall that if an obligation is in an answer set, the agent has to take the corresponding action. We encode $O(\varphi \rightarrow \psi)$ using a weak constraint that sanctions answer sets including $Do(\varphi)$ but not $O(\psi)$. As every answer set that contains $O(\varphi)$ also contains $Do(\varphi)$, each answer set that contains $O(\varphi)$ but not $O(\psi)$ is also sanctioned. This way we simulate *KD*, as an answer set that contains $O(\varphi)$ must also contain $O(\psi)$, unless the latter conflicts with an obligation of higher or equal importance.

6 A Case Study: Ethical Pac-man

We put the methodology described in the previous section to work on a reinforcement learning agent playing variants of the game Pac-man. Pac-man features a closed environment with simple game mechanics and parameters which are easy to manipulate, and extend with norms that can simulate normative conflicts.

The starting position of the game is depicted in Fig. 2. Pac-man's objective is to eat all the pellets in the maze while avoiding two ghosts (orange and blue) that will kill him upon contact. Pac-man and the ghosts typically move one step at a time, with the ghosts' movements being non-deterministic. If Pac-man ate one of the larger pellets, the ghosts enter a scared state and become vulnerable, allowing Pac-man to eat them. In this state, the ghosts move at half speed. A scared ghost is instantly eaten by Pac-man when their distance is less than 1 on both axes. Points are awarded for consuming pellets and ghosts, with a discount applied based on the duration of the game (the longer the game lasts, the lower the score). Pac-man wins if he collects all the pellets, and a faster completion time results in a higher score.

Figure 2: Pac-man



Following [23, 22], we consider variants of the Pac-man game with additional "ethical norms". *Vegan Pac-man*, in which Pac-man is not allowed to eat any ghost has been introduced in [23] and implemented there using multi-objective Reinforcement Learning (RL) with policy orchestration. Vegan Pac-man and its vegetarian variant were analyzed in [22], that will serve as the benchmark for our work. *Vegetarian Pac-man* can eat the orange ghost (as it would be cheese) but not the blue ghost. The approach used in [22] combines RL with formal tools for normative reasoning. The authors implement a logic-based *normative supervisor* module, which informs the trained RL agent of the ethical requirements in force in a given situation. At each step, Pac-man chooses an action complying with the norms, and a least evil action if there is no such action. Their approach allows to deal with complex normative systems, conflicting obligations, and situations where no compliance is possible. Norms and the current state of the agent's environment are encoded in defeasible deontic logic [14, 15], which is a deontic logic with defeasible rules that specify typical correlations, such as "birds usually fly"; its theorem prover SPINdle is used to check norms compliance. Exceptions are encoded by so-called defeaters, e.g., if the bird is a penguin.

We consider here an alternative realization of the normative supervisor, based on our norms encoding and using the DLV reasoner. We experimentally compare the obtained results and apply our methodology to more intricated "ethical norms" for Pac-man.

Vegan Pac-man: To prohibit Pac-man from eating any ghost, we can state:

$$O(\neg eat(g)) \qquad \text{respectively} \qquad F(eat(g)), \quad \text{for } g \in \{blue\_ghost, blue\_ghost\}.$$

Vegetarian Pac-man: To prohibit Pac-man from eating the blue ghost, we state:

$$O(\neg eat(blue\_ghost)) \qquad \text{respectively} \qquad F(eat(blue\_ghost)).$$

We encode the norm bases by forbidding Pac-man to move in a direction if the ghosts are scared and moving there could lead to eat a ghost. Furthermore, we forbid Pac-man from stopping if a ghost could move into Pac-man (and then be eaten). Pac-man may still eat a ghost. This can happen if both a ghost and Pac-man move towards a larger pellet from perpendicular directions. In this case, Pac-man will first eat the pellet and then the ghost. Furthermore, Pac-man could be cornered between two scared ghosts, leaving him no choice but eating one of them.

The scenarios that can precede Pac-man eating a ghost are the same for both norm bases. As Pac-man and the ghosts can move at most one step at a time, we can derive that the Manhattan distance between Pac-man and a scared ghost must in this case be at least 1 and at most 2 (coordinates are integers). This leaves three possibilities for their relative locations. We encoded the norms by accounting for the locations of the ghosts relative to Pac-man and forbidding Pac-man to make moves that could lead to eating a ghost.

Experimental Results. The vegan norm base was implemented in [23] who trained two different models for Pac-man; in one model he was trained to maximize the game score, and in the other to comply with the norms using respective data. An external function enabled the agent to decide which model to use for choosing the next move. When the importance of the norm-compliance model was low, the agent in general did not comply with the norm, resulting in around 2 ghosts eaten per game. Making the

norm base	% games won	game score avg[max]	avg ghosts eaten (blue/orange)	avg time (s)
Vegan	90.7 | 91.2	1209.86[1708] | 1217[1538]	0.023/0.02 | 0.013/0.018	10.1 | 6.7
Vegetarian	94.0 | 90.6	1413.80[1742] | 1366[1751]	0.01/0.79 | 0.001/0.788	9.8 | 6.5
Weak Vegan	| 89.9	| 1204[1731]	| 0.002/0.043	| 6.7

Table 2: Results for Neufeld et al.'s normative supervisor [22] | our encoding

importance sufficiently high, the number of ghosts eaten did decrease to one (similar to our results, see Table 2). In this approach, however, it is not clear how to enforce more complex norm sets, including, e.g., contrary-to-duty obligations.

In order to compare Neufeld et al.'s approach [22] with our ASP encodings, we utilized their Java-based framework. This framework is built in Java and provides the normative reasoners with a ready-to-use interface to the Pac-man game.[2] The RL agent was trained on 250 games and the normative supervisor was evaluated on 1000 games with the same initial state.

The results reported in [22] for their normative supervisor (using SPINdle) and for our ASP-based normative supervisor (using JDLV, the java framework for DLV) are shown in Table 2. The latter outperformed the original supervisor for both norm bases w.r.t. the ghosts eaten and speed: its average running time was less than 7 seconds, while for Neufeld et al.'s normative supervisor it was some seconds (roughly 50%) longer, using a Lenovo Y50-70 with 8GB RAM and Ubuntu 22.04 LTS. For the vegan norm base, our encoding resulted in a higher winning rate and an improved average score. (A game loss costs 500 points, resulting possibly in a negative score.) The higher maximum score in Neufeld et al.'s results is likely attributed to a game where both ghosts were eaten. In the vegetarian norm base, the lower winning rate and average score is probably due to Pac-man's preference to lose in our framework a game rather than eating a ghost.

Weak Vegan Pac-man. Consider the following more intricated variant of the vegan norm base:

O1 It is obligatory not to eat the blue ghost.

O2 It is obligatory not to eat the orange ghost, unless a ghost has already been eaten.

O3 It is obligatory to stop (for one move) after having eaten a ghost.

Thus, the behaviour may change at some point. To encode it, we follow our methodology:

Step 1: We categorise the obligations.

- O1 is a regular obligation. It has no exceptions and should be followed at all times.
- O2 is an obligation with an exception. Once the latter occurs, it holds until the game is over.
- O3 is a derived obligation. Note that it is not necessarily a CTD obligation, as eating a ghost may not violate an obligation if the ghost is blue.

Step 2: We look at pairs of obligations that cannot be fulfilled simultaneously.

- O1 and O2 could possibly be in conflict, as Pac-man may be stuck between two ghosts moving towards him. This may force him to eat one of them. In this case we prioritize O1, as we want preventing the blue ghost from being eaten to have the highest priority.
- O1 and O3 could be in conflict, as stopping after eating an orange ghost may lead to eating a blue ghost.

---

[2] For complete DLV code and experimental data, see `github.com/Chrisi-boop/DLV-Normative-Reasoning-`.

Figure 3: Encoding for the Pac-man norm base

$$\begin{array}{l}
:\sim -F(eat(blue\_ghost)).\ [1:3] \\
:\sim not\ \ Exception, \\
\quad -F(eat(orange\_ghost)).\ [1:2] \\
:\sim -O(stop), \\
\quad Do(eat(orange\_ghost)).\ [1:2] \\
:\sim -O(stop), \\
\quad Do(eat(blue\_ghost)).\ [1:2]
\end{array}$$

(a) obligations

$$\begin{array}{l}
:- Do(stop),\ \ Do(north). \\
\quad \dots \\
:- F(eat(blue\_ghost)), \\
\quad -F(east), Pacman(A,B), \\
\quad BlueGhost(C,D,1), \\
\quad E = C - A, E \leq 2, \\
\quad G = B - D, G \leq 1. \\
\quad \dots
\end{array}$$

(b) conflicting actions

$$\begin{array}{l}
act(d).\ \text{for } d \in D \\
act(eat(blue\_ghost)). \\
act(eat(orange\_ghost)). \\
:- \bigwedge_{d \in D} F(d).
\end{array}$$

(c) action specs; $D = \{stop, north, east, south, west\}$

– O2 and O3 cannot be in conflict, as after eating a ghost the exception to O2 is empowered. We give priority to O1 over O3 since the preservation of the blue ghost from being consumed has highest priority.

Summarizing the statements above, we obtain the following preferences: O1 $\succ$ O2, O1 $\succ$ O3.

Step 3: We derive the following weights and levels for the weak constraints corresponding to the obligations: O1 is assigned [1 : 3], O2 is assigned [1 : 2], and O3 is assigned [1 : 2].

We next look at the predicates used in the encoding:

- *Pacman(X,Y)* denotes the location of Pac-man. X and Y refer to his coordinates on the map.
- *BlueGhost(X,Y,B)* and *OrangeGhost(X,Y,B)* denotes the location of the blue resp. orange ghost, where X,Y are its coordinates and B is 1 if the ghost is scared and 0 otherwise.
- *Exception* means the agent has already eaten a ghost and may eat orange ghosts. Whenever *Exception* is in an answer set, the normative reasoner will inject it into any later answer set.

The actions to reason about are *stop*, *north*, *east*, *south*, *west*, *eat(blue_ghost)*, *eat(orange_ghost)*:

- *Do(stop)* means that the agent remains stationary for one action.
- *Do(d)*, $d \in \{north, east, south, west\}$, means that the agent will move in direction d.
- *Do(eat(g)*, $g \in \{blue\_ghost, orange\_ghost\}$ means that the agent eats that ghost.

To encode the norms we add the rules and facts in Fig. 3 to the common core. Following our methodology, we encode the obligations as in Fig. 3a. Note that the obligation to eat the blue and orange ghost are encoded as separate actions as two different obligations (O3).

Step 4: The conflicting actions from Step 2 are encoded in Fig. 3b. For Pac-man, we encode that he cannot take two actions at once, e.g., stop and move north. For ghosts, as a scared ghost may move both towards and away from Pac-man, we cannot encode ensuing conflicts directly. However, we can assert that permitting an action that could potentially involve consuming a ghost is inconsistent with the prohibition of consuming that ghost. Fig. 3.b shows one such rule; for space reasons we omit the other rules, as well as further action constraints for Pac-man.[2]

Step 5: We state the actions to reason about (Fig. 3c), and encode that it is not possible to prevent Pac-man from taking any action (i.e., Pac-man must either stop or move).

In fact, our actual encoding omits the actions $eat(blue\_ghost)$ and $eat(orange\_ghost)$; they are substituted by prohibitions on moving towards scared ghosts. We used them here for readability.

Experiments. The implementation of this norm base yielded results that align with our expectations, as depicted in Table 2. Pac-man consumed more ghosts compared to the vegan norm base but fewer than the vegetarian norm base, and mostly orange ghosts. The obligation to halt when Pac-man eats a ghost resulted in a lower average score, as time passage leads to point deductions. This observation also explains the marginal increase in the maximum score and the slight decrease in the winning rate. Despite the added complexity of the norm base, there was no significant difference in the running time.

7 Related Work and Conclusion

Starting with an analysis of well-known deontic paradoxes, we have introduced a methodology to encode normative systems in ASP, using DLV as the system of choice. Our approach determines optimal ways to handle scenarios, using agreed upon prioritizations of obligations.

Multiple approaches to implement normative systems do exist. Some of those related to the multi-agent systems community can be found, e.g., in [2]. We will discuss below the approaches most similar to ours.

One of the earliest works on encoding normative systems is Sergot et al. [25], who did encode the British Nationality Act in logic programming. However, their work focused on determining the applicability of the British Nationality Act to specific individuals, without delving into the reasoning about obligations or seeking optimal courses of action within the framework of given norms.

[26] introduced in IMPACT syntax and semantics for reasoning about obligations and prohibitions among agents in a rule-based language under different semantics, among them stable model semantics. Although they refer to deontic logic, the proposed way of dealing with conflicting obligations is to satisfy a maximal subset of obligations, without considering possible preferences among them.

An abductive logic programming framework has been employed in [19] to encode obligations in various deontic paradoxes. Rather than trying to derive all optimal ways of fulfilling given obligations, the authors focused on finding a best model of a definite Horn logic program that satisfies given goals. When it comes to establishing model preferences, the auxiliary symbols used in the encodings played a significant role. However, their usage requires care, and no systematic way of defining model preference was considered, for which our work could provide inspiration.

Using a combination of input-output logic and game theoretic methods, [6] encoded the behaviour of (multi-)agents subjected to a normative system. In their work, agents possess the ability to reason in a manner more akin to humans, for example, determining if the consequences of violating an obligation are worth the penalty. Their work lacks however computational support.

Temporal Logic has been used to simulate normative reasoning. Temporal Logic approaches offer an advantage in enforcing norms that indirectly restrict certain actions [1, 8]. Moreover there are advanced tools that effectively combine reinforcement learning, with, e.g., LTL (see [4]), and LDLf (see [9]). However, as is well known temporal logic cannot handle all the intricacies of normative reasoning, see, e.g., the discussion in [2, 21].

Different ASP approaches have been proposed in [13, 3, 12]. In these works, logic programs were extended with the SDL modality, and [12] introduces also temporal operators. Alferes et al. [3] further provides a way to check equivalence of two deontic logic programs. These approaches however require an understanding of the embedded logic and some implementation efforts, whereas our method can be used out of the box and with an ASP solver off the shelf.

In our approach, the selection of paradoxes has proven to be of utmost importance, as failure to include certain ones resulted in overlooking crucial aspects of normative systems; e.g., the analysis of the *Fence Paradox* has enabled us to differentiate between contrary-to-duty obligations and exceptions, which is a well-known problem in the field of normative reasoning [24]. An important feature of our approach is the availability of optimized tools (ASP solvers) and the simplicity of the encoding. There is a clear cut common core that is supplemented with defined ways for encoding different kinds of obligations. The approach, described for DLV in this work, can be easily transferred to other ASP solvers, e.g. clasp (`potassco.org/clasp`). A main limitation of our approach is that encoding intricate normative systems can result in large programs. Furthermore, obligations that require the agent to maintain certain conditions might be cumbersome to encode for complex scenarios where taking particular actions might indirectly lead to a violation. Consider for instance "See to it that the child stays dry." Here it is not enough to simply not get the child wet, but one must also take measures to protect the child from getting wet through other means. This may entail preventing the child from going outside if it is about to rain. For such normative systems, approaches extending ASP with temporal logics [12], may be preferable.

Outlook. A peculiar feature of our approach is that all encoded obligations are comparable based on their associated weights and priorities. While in our encoding we did not run into problems, it is possible that certain normative systems may require optimal answer sets encoding solutions that are inherently incomparable. Future work may also look into other ASP solvers with more sophisticated means for filtering out answer sets to model normative reasoning. The clasp extension asprin (`potassco.org/asprin`) or the DLV2 system using WASP [5], could serve as examples.

The Pac-man case study highlighted the fact that even simple obligations can require a substantial number of weak constraints to accurately encode the means of fulfilling those obligations. To address this weakness, we plan to explore the integration of our framework with auxiliary software capable of reasoning about the actions neeeded to fulfill specific obligations. In the context of Pac-man, such software might have to interpret, e.g., how the obligation to not eat a ghost could be fulfilled.

References

[1] Natasha Alechina, Nils Bulling, Mehdi Dastani & Brian Logan (2015): *Practical Run-Time Norm Enforcement with Bounded Lookahead.* In Gerhard Weiss, Pinar Yolum, Rafael H. Bordini & Edith Elkind, editors: *Proceedings of AAMAS 2015*, ACM, pp. 443–451.

[2] Natasha Alechina, Mehdi Dastani & Brian Logan (2018): *Norm Specification and Verification in Multi-Agent Systems. J. of Applied Logics* 5(2), pp. 457–490.

[3] José Júlio Alferes, Ricardo Gonçalves & João Leite (2013): *Equivalence of defeasible normative systems. J. Appl. Non Class. Logics* 23(1-2), pp. 25–48, doi:10.1080/11663081.2013.798996.

[4] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum & Ufuk Topcu (2018): *Safe reinforcement learning via shielding.* In: *Proc. AAAI*, pp. 2669–2678.

[5] Mario Alviano, Carmine Dodaro, João Marques-Silva & Francesco Ricca (2020): *Optimum stable model search: algorithms and implementation. J. Log. Comput.* 30(4), pp. 863–897, doi:10.1093/logcom/exv061.

[6] Guido Boella & Leendert W. N. van der Torre (2004): *Regulative and Constitutive Norms in Normative Multiagent Systems.* In Didier Dubois, Christopher A. Welty & Mary-Anne Williams, editors: *Proceedings of KR2004*, AAAI Press, pp. 255–266.

[7] Gerd Brewka, Thomas Eiter & Miroslaw Truszczyński (2011): *Answer Set Programming at a Glance. Communications of the ACM* 54(12), pp. 92–103, doi:10.1145/2043174.2043195.

[8] Nils Bulling, Mehdi Dastani & Max Knobbout (2013): *Monitoring norm violations in multi-agent systems*. In Maria L. Gini, Onn Shehory, Takayuki Ito & Catholijn M. Jonker, editors: *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '13*, IFAAMAS, pp. 491–498.

[9] Giuseppe De Giacomo & Moshe Y. Vardi (2015): *Synthesis for LTL and LDL on Finite Traces*. In: *IJCAI 2015*, pp. 1558–1564.

[10] Dov Gabbay, John Horty, Xavier Parent, Ron van der Meyden & Leendert van der Torre, editors (2021): *Handbook of Deontic Logic and Normative Systems*. College Publications. Amends Volume 1 (2013).

[11] M. Gelfond & V. Lifschitz (1991): *Classical Negation in Logic Programs and Disjunctive Databases*. *New Generation Computing* 9, pp. 365–385, doi:10.1007/BF03037169.

[12] Laura Giordano, Alberto Martelli & Daniele Theseider Dupré (2013): *Temporal deontic action logic for the verification of compliance to norms in ASP*. In Enrico Francesconi & Bart Verheij, editors: *International Conference on Artificial Intelligence and Law, ICAIL '13*, ACM, pp. 53–62, doi:10.1145/2514601.2514608.

[13] Ricardo Gonçalves & José Júlio Alferes (2012): *An Embedding of Input-Output Logic in Deontic Logic Programs*. In Thomas Ågotnes, Jan M. Broersen & Dag Elgesem, editors: *Deontic Logic in Computer Science - 11th International Conference, DEON 2012, Proceedings*, *LNCS* 7393, Springer, pp. 61–75, doi:10.1007/978-3-642-31570-1_5.

[14] Guido Governatori, Francesco Olivieri, Antonino Rotolo & Simone Scannapieco (2013): *Computing Strong and Weak Permissions in Defeasible Logic*. *J. Phil. Log.* 42(6), pp. 799–829, doi:10.1007/s10992-013-9295-1.

[15] Guido Governatori & Antonino Rotolo (2008): *BIO logical agents: Norms, beliefs, intentions in defeasible logic*. *Auton. Agents Multi Agent Syst.* 17(1), pp. 36–69, doi:10.1007/s10458-008-9030-4.

[16] Christian Hatschka (2022): *Representing Normative Reasoning in Answer Set Programming Using Weak Constraints*. Master's thesis, Technische Universität Wien, doi:10.34726/hss.2022.99420.

[17] Andrew Jones & José Carmo (2002): *Deontic Logic and Contrary-to-Duties*. *Handbook of Philosophical Logic* vol.8, pp. p.265–364, doi:10.1007/978-94-010-0387-2_4.

[18] Benjamin Kaufmann, Nicola Leone, Simona Perri & Torsten Schaub (2016): *Grounding and Solving in Answer Set Programming*. *AI Mag.* 37(3), pp. 25–32, doi:10.1609/aimag.v37i3.2672.

[19] Robert A. Kowalski & Ken Satoh (2018): *Obligation as Optimal Goal Satisfaction*. *J. Phil. Log.* 47(4), pp. 579–609, doi:10.1007/s10992-017-9440-3.

[20] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri & Francesco Scarcello (2006): *The DLV System for Knowledge Representation and Reasoning*. *ACM Transactions on Computational Logic* 7(3), pp. 499–562, doi:10.1145/116825.116838.

[21] Emery Neufeld, Ezio Bartocci & Agata Ciabattoni (2022): *On Normative Reinforcement Learning via Safe Reinforcement Learning*. In: *Proceedings of PRIMA 2022*.

[22] Emery A. Neufeld, Ezio Bartocci, Agata Ciabattoni & Guido Governatori (2021): *A Normative Supervisor for Reinforcement Learning Agents*. In André Platzer & Geoff Sutcliffe, editors: *Automated Deduction - CADE 28*, *LNCS* 12699, Springer, pp. 565–576, doi:10.1007/978-3-030-79876-5_32.

[23] Ritesh Noothigattu, Djallel Bouneffouf, Nicholas Mattei, Rachita Chandra, Piyush Madan, Kush R. Varshney, Murray Campbell, Moninder Singh & Francesca Rossi (2019): *Teaching AI Agents Ethical Values Using Reinforcement Learning and Policy Orchestration*. In: *Proc of IJCAI 2019*, ijcai.org, doi:10.24963/ijcai.2019.

[24] Henry Prakken & Marek J. Sergot (1996): *Contrary-to-Duty Obligations*. *Stud Logica* 57(1), pp. 91–115, doi:10.1007/BF00370671.

[25] Marek J. Sergot, Fariba Sadri, Robert A. Kowalski, F. Kriwaczek, Peter Hammond & H. T. Cory (1986): *The British Nationality Act as a Logic Program*. *Commun. ACM* 29(5), pp. 370–386, doi:10.1145/5689.5920.

[26] V.S. Subrahmanian, Piero Bonatti, Jürgen Dix, Thomas Eiter, Sarit Kraus & Robert Ross (2000): *Heterogeneous Agent Systems*, chapter 6: Agent Programs, pp. 115–170. MIT PR, doi:10.7551/mitpress/3487.001.0001.

[27] Georg Henrik von Wright (1951): *Deontic logic*. *Mind* 60(237), pp. 1–15, doi:10.1093/mind/LX.237.1.

Alternating Fixpoint Operator for Hybrid MKNF Knowledge Bases as an Approximator of AFT
(Extended Abstract)

Fangfang Liu[1] and *Jia-Huai You*[2]
[1]School of Computer Engineering and Science, Shanghai University
Shanghai, China
[2]Department of Computing Science, University of Alberta
Edmonton, Canada

Approximation fixpoint theory (AFT) provides an algebraic framework for the study of fixpoints of operators on bilattices and has found its applications in characterizing semantics for various non-monotonic languages. Here, we show that the alternating fixpoint operator by Knorr et al. for the study of the well-founded semantics for hybrid MKNF is in fact an approximator of AFT in disguise, which, thanks to the abstraction power of AFT, characterizes all two-valued and three-valued semantics for hybrid MKNF. Furthermore, we show an improved approximator for these knowledge bases, of which the least stable fixpoint is information richer than the one formulated from Knorr et al.'s construction, which leads to an improved computation for the well-founded semantics. This work is built on an extension of AFT that supports consistent as well as inconsistent pairs in the induced product bilattice, to deal with inconsistencies that arise in the context of hybrid MKNF. This part of the work generalizes the original AFT from symmetric approximators to arbitrary approximators.

1 Generalization of AFT

Given a complete lattice $\langle L, \leq \rangle$, AFT is built on the induced product bilattice $\langle L^2, \leq_p \rangle$, where $\leq_p$ is the *precision order* and defined as: for all $x,y,x',y' \in L$, $(x,y) \leq_p (x',y')$ if $x \leq x'$ and $y' \leq y$. A pair $(x,y) \in L^2$ is *consistent* if $x \leq y$ and *inconsistent* otherwise, and a pair (x,x) is called *exact*. We denote the set of consistent pairs by L^c. Since the $\leq_p$ ordering is a complete lattice ordering on L^2, $\leq_p$-monotone operators on L^2 possess fixpoints and a least fixpoint. The original AFT is restricted to consistent and symmetric approximators [1, 2], and we generalize it to all $\leq_p$-monotone operators on L^2.

Definition 1 *An operator $A : L^2 \to L^2$ is an approximator if A is $\leq_p$-monotone on L^2 and for all $x \in L$, whenever $A(x,x)$ is consistent, A maps (x,x) to an exact pair.*

The original definition of an approximator requires an $\leq$-monotone operator $A : L^2 \to L^2$ to map an exact pair to an exact pair. For example, consider a complete lattice where $L = \{\bot, \top\}$ and $\leq$ is defined as usual. Let A be an identity function on L^2 everywhere except $A(\top,\top) = (\top,\bot)$. Thus, $A(\top,\top)$ is inconsistent. But it is easy to check that A is $\leq_p$-monotone. Thus, while the operator A is an approximator by our definition, it is not by the original definition.

For the study of semantics, the main focus has been on the *stable revision operator*, which is defined as: Given any pair $(u,v) \in L^2$ and an approximator A, we define $St_A(u,v) = (lfp(A(\cdot,v)_1), lfp(A(u,\cdot)_2))$, where $A(\cdot,v)_1$ denotes the operator $L \to L : z \mapsto A(z,v)_1$ and $A(u,\cdot)_2$ denotes the operator $L \to L : z \mapsto A(u,z)_2$. That is, both $A(\cdot,v)_1$ and $A(u,\cdot)_2$ are projection operators on L. Since A is $\leq_p$-monotone on L^2, both projection operators $A(\cdot,v)_1$ and $A(u,\cdot)_2$ are $\leq$-monotone on L for any pair $(u,v) \in L^2$, and thus the

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 381–383, doi:10.4204/EPTCS.385.40

least fixpoint exists for each. The stable revision operator is therefore well-defined, and furthermore, it is $\leq_p$-monotone as well. The fixpoints of the stable revision operator St_A are called *stable fixpoints* of A.

In general, however, there is no guarantee that under the above notion of stable revision consistent stable fixpoints under consistent AFT are preserved. One can argue that a desirable approximator should not exhibit this "anomaly" so that accommodating inconsistent pairs does not have to sacrifice the preservation of consistent stable fixpoints.

Definition 2 *Let A be an approximator on L^2 and $(u,v) \in L^c$ such that $(u,v) = (lfp(A(\cdot,v)_1), lfp(A(u,\cdot)_2))$ where $A(\cdot,v)_1$ is an operator on $[\bot,v]$ and $A(u,\cdot)_2$ is an operator on $[u,\top]$.*[1] *Approximator A is called* strong *for (u,v) if $(u,v) = (lfp(A(\cdot,v)_1), lfp(A(u,\cdot)_2))$ where both $A(\cdot,v)_1$ and $A(u,\cdot)_2$ are operators on L. Approximator A is called* strong *if it is strong for every $(u,v) \in L^c$ that satisfies the above condition.*

2 Alternating Fixpoint Operator for Hybrid MKNF as an Approximator

The logic of MKNF consists of a first-order language with two modal operators, **K**, for minimal knowledge, and **not**, for negation-as-failure. Motik and Rosati [5] propose hybrid MKNF for integrating nonmonotonic rules with description logics (DLs) under two-valued logic, and Knorr et al. [3] generalize it to three-valued semantics where the information-minimum three-valued MKNF model of an MKNF formulat ϕ, when it exists, is call the *well-founded MKNF model* of ϕ.

In three-valued MKNF, formulas are evaluated by *three-valued MKNF structures* in the form of $(I, \mathscr{M}, \mathscr{N})$, which consists of a first-order interpretation, I, and two pairs, $\mathscr{M} = \langle M, M_1 \rangle$ and $\mathscr{N} = \langle N, N_1 \rangle$, of sets of first-order interpretations, where $M_1 \subseteq M$ and $N_1 \subseteq N$. I is to evaluate object formulas, $\mathscr{M}$ is to evaluate **K**-atoms, and $\mathscr{N}$ to evaluate **not**-atoms. From the two component sets in $\mathscr{M} = \langle M, M_1 \rangle$, three truth values for modal **K**-atoms are defined as: $\mathbf{K}\varphi$ is true w.r.t. $\mathscr{M} = \langle M, M_1 \rangle$ if φ is true in all interpretations in M; it is false if it is false in at least one interpretation in M_1; and it is undefined otherwise. For **not**-atoms, a symmetric treatment w.r.t. $\mathscr{N} = \langle N, N_1 \rangle$ is adopted. A three-valued MKNF model (M,N) of an MKNF formula ϕ is a three MKNF structure $(I,(M,N),(M,N))$ that satisfies ϕ and a condition that guarantees minimal knowledge and "minimal undefinedness".

A hybrid MKNF knowledge base $\mathscr{K} = (\mathscr{O}, \mathscr{P})$ consists of a decidable description logic (DL) knowledge base $\mathscr{O}$, translatable into first-order logic and a rule base $\mathscr{P}$, which is a finite set of rules with modal atoms. Following [3], our focus here is on nondisjunctive rules.

A (normal MKNF) rule r is of the form: $\mathbf{K}H \leftarrow \mathbf{K}A_1, \ldots, \mathbf{K}A_m, \mathbf{not}B_1, \ldots, \mathbf{not}B_n$, where H, A_i, and B_j are function-free first-order atoms. Given a rule r, we let $hd(r) = \mathbf{K}H$, $bd^+(r) = \{\mathbf{K}A_i \,|\, i = 1..m\}$, and $bd^-(r) = \{B_i \,|\, i = 1..n\}$. We use the notation $\mathbf{K}(bd^-(r)) = \{\mathbf{K}B_1, \ldots, \mathbf{K}B_n\}$.

For the interpretation of a hybrid MKNF knowledge base $\mathscr{K} = (\mathscr{O}, \mathscr{P})$, a transformation $\pi(\mathscr{K}) = \mathbf{K}\pi(\mathscr{O}) \wedge \pi(\mathscr{P})$ is performed to transform $\mathscr{O}$ into a first-order formula and rules $r \in \mathscr{P}$ into a conjunction of first-order implications to make each of them coincide syntactically with an MKNF formula, i.e.,

$$\pi(r) = \forall \vec{x}\colon (\mathbf{K}H \subset \mathbf{K}A_1 \wedge \ldots \wedge \mathbf{K}A_m \wedge \mathbf{not}B_1 \wedge \ldots \wedge \mathbf{not}B_n)$$
$$\pi(\mathscr{P}) = \textstyle\bigwedge_{r \in \mathscr{P}} \pi(r), \;\; \pi(\mathscr{K}) = \mathbf{K}\pi(\mathscr{O}) \wedge \pi(\mathscr{P})$$

where $\vec{x}$ is the vector of free variables in r.

Under the additional assumption of DL-safety a first-order rule base is semantically equivalent to a finite ground rule base under both two-valued [5] and three-valued semantics [3]; hence decidability is guaranteed. In the following, we assume that a given rule base is DL-safe and already grounded.

---

[1] This is the original definition of stable revision for consistent AFT.

Given a hybrid MKNF knowledge base $\mathcal{K} = (\mathcal{O}, \mathcal{P})$, let $\mathsf{KA}(\mathcal{K})$ be the set of all (ground) **K**-atoms $\mathbf{K}\phi$ such that either $\mathbf{K}\phi$ occurs in $\mathcal{P}$ or $\mathbf{not}\,\phi$ occurs in $\mathcal{P}$. A *partition* of $\mathsf{KA}(\mathcal{K})$ is a pair (T,P) such that $T,P \subseteq \mathsf{KA}(\mathcal{K})$; if $T \subseteq P$, then (T,P) is said to be consistent, otherwise it is inconsistent.

Knorr et al. [3] define an alternating fixpoint construction to compute the well-founded MKNF model. Their construction can be equivalently formulated by stable revisions with the following approximator.

Definition 3 *Let $\mathcal{K} = (\mathcal{O}, \mathcal{P})$ be a normal hybrid MKNF knowledge base. Define an operator $\Phi_{\mathcal{K}}$ on $(2^{\mathsf{KA}(\mathcal{K})})^2$ as follows: $\Phi_{\mathcal{K}}(T,P) = (\Phi_{\mathcal{K}}(T,P)_1, \Phi_{\mathcal{K}}(T,P)_2)$, where*

$$\begin{aligned}\Phi_{\mathcal{K}}(T,P)_1 = \{&\mathbf{K}\,a \in \mathsf{KA}(\mathcal{K}) \mid \mathsf{OB}_{\mathcal{O},T} \models a\} \cup \\ &\{\mathbf{K}a \mid r \in \mathcal{P} : hd(r) = \{\mathbf{K}a\}, bd^+(r) \subseteq T, \mathbf{K}(bd^-(r)) \cap P = \emptyset\} \\ \Phi_{\mathcal{K}}(T,P)_2 = \{&\mathbf{K}\,a \in \mathsf{KA}(KB) \mid \mathsf{OB}_{\mathcal{O},P} \models a\} \cup \\ &\{\mathbf{K}a \mid r \in \mathcal{P} : hd(r) = \{\mathbf{K}a\}, \mathsf{OB}_{\mathcal{O},T} \not\models \neg a, bd^+(r) \subseteq P, \mathbf{K}(bd^-(r)) \cap T = \emptyset\}\end{aligned}$$

Stable revision computes the least fixpoints of the projection operators $\Phi_{\mathcal{K}}(\cdot,P)_1$ and $\Phi_{\mathcal{K}}(T,\cdot)_2$, respectively. Intuitively, given a partition (T,P), the operator $\Phi_{\mathcal{K}}(\cdot,P)_1$, with P fixed, computes the set of true **K**-atoms w.r.t. (T,P) and operator $\Phi_{\mathcal{K}}(T,\cdot)_2$, with T fixed, computes the set o **K**-atoms that are possibly true w.r.t. (T,P). The condition $\mathsf{OB}_{\mathcal{O},T} \not\models \neg a$ attempts to avoid the generation of a contradiction. We can show that $\Phi_{\mathcal{K}}$ is an approximator on the bilattice $(2^{\mathsf{KA}(\mathcal{K})})^2$ and preserves all consistent stable fixpoints (and is thus a strong approximator). Note that operator $\Phi_{\mathcal{K}}$ is not symmetric and it can map a consistent pair to an inconsistent one. A major advantage of AFT is that the stable fixpoints of $\Phi_{\mathcal{K}}$ satisfying a consistency condition characterize all three-valued MKNF models.

Theorem 1 *Let $\mathcal{K} = (\mathcal{O}, \mathcal{P})$ be a normal hybrid MKNF knowledge base and (T,P) be a partition. Also let $(M,N) = (\{I \mid I \models \mathsf{OB}_{\mathcal{O},T}\}, \{I \mid I \models \mathsf{OB}_{\mathcal{O},P}\})$. Then, (M,N) is a three-valued MKNF model of $\mathcal{K}$ iff (T,P) is a consistent stable fixpoint of $\Phi_{\mathcal{K}}$ and $\mathsf{OB}_{\mathcal{O},lfp(\Phi_{\mathcal{K}}(\cdot,T)_1)}$ is satisfiable.*

Thus, given a hybrid MKNF KB $\mathcal{K}$, we can compute the least stable fixpoint of $\Phi_{\mathcal{K}}$ and check if $\mathsf{OB}_{\mathcal{O},lfp(\Phi_{\mathcal{K}}(\cdot,T)_1)}$ is satisfiable to determine the existence of a well-founded MKNF model. The process is polynomial if so is the underlying DL. Since the problem of computing the well-founded MKNF model is in general intractable, even for normal hybrid MKNF KBs [4], a question is whether we can enlarge the class of hybrid MKNF KBs whose well-founded MKNF model can be tractably computed this way. We answer this question positively by formulating a strictly more precise approximator.

References

[1] Marc Denecker, Victor Marek & Mirosław Truszczyński (2000): *Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning*. In: Logic-Based Artificial Intelligence, Springer, pp. 127–144, doi:10.1007/978-1-4615-1567-8_6.

[2] Marc Denecker, Victor W. Marek & Miroslaw Truszczynski (2004): *Ultimate approximation and its application in nonmonotonic knowledge representation systems*. Information and Computation 192(1), pp. 84–121, doi:10.1016/j.ic.2004.02.004.

[3] Matthias Knorr, José Júlio Alferes & Pascal Hitzler (2011): *Local closed world reasoning with description logics under the well-founded semantics*. Artificial Intelligence 175(9-10), pp. 1528–1554, doi:10.1016/j.artint.2011.01.007.

[4] Fangfang Liu & Jia-Huai You (2017): *Three-valued semantics for hybrid MKNF knowledge bases revisited*. Artificial Intelligence 252, pp. 123–138, doi:10.1016/j.artint.2017.08.003.

[5] Boris Motik & Riccardo Rosati (2010): *Reconciling Description Logics and Rules*. Journal of the ACM 57(5), pp. 1–62, doi:10.1145/1754399.1754403.

Summary of Statistical Statements in Probabilistic Logic Programming

Damiano Azzolini Elena Bellodi Fabrizio Riguzzi
University of Ferrara
Ferrara, Italy
`{damiano.azzolini,elena.bellodi.fabrizio.riguzzi}@unife.it`

Statistical statements, also called Type 1 statements by Halpern, have a new representation in terms of Probabilistic Answer Set Programming under the credal semantics. In this extended abstract we summarize that contribution.

1 Contribution

This is a summary of the paper "Statistical Statements in Probabilistic Logic Programming" [1] presented at LPNMR2022. In First-Order knowledge bases, we can adopt statistical statements, also known as Type 1 statements [6], to represent statistical information about the domain, such as "40% of the elements have a particular property". The authors of [7] propose to encode probabilistic conditionals (i.e., statistical statements) with formulas of the form $(C \mid A)[\Pi]$, where C and A are First-Order formulas and $\Pi \in [0,1]$ with the meaning that $100 \cdot \Pi\%$ of elements that satisfy formula A also satisfy C, and compute the probability of a query by adopting the maximum entropy principle [8].

Differently, in [1] the authors propose to encode statistical statements with the syntax

$$(C \mid A)[\Pi_l, \Pi_u]$$

with the meaning that at least $100 \cdot \Pi_l\%$ and at most $100 \cdot \Pi_u\%$ of the elements that satisfy formula A also satisfy C. To assign a semantics to these and to perform inference, the authors propose to adopt the credal semantics of probabilistic answer set programs [3] and encode statements with answer set rules and constraints [2]. Probabilistic answer set programming under the credal semantics represents uncertainty with answer set programming extended with ProbLog probabilistic facts [4] $\Pi :: f$, with the meaning that the probability of f of being true if Π ($\Pi \in [0,1]$). A *world* is a selection of a truth value for every probabilistic fact and is an answer set program with one or more answer sets (the semantics requires that every world has at least one answer set). A world is associated with a probability value computed as

$$P(w) = \prod_{f_i \text{ selected}} \Pi_i \cdot \prod_{f_i \text{ not selected}} (1 - \Pi_i).$$

Since every world has one or more stable models, the probability of a query $P(q)$ is defined by a range: a world contributes to the upper bound if the query is true in at least one answer set and also contributes to the lower bound if it is true in every answer set. It is required that every world has at least one answer set.

The meaning of a statistical statement is given in terms of models for every world. That is, the models of a world where the constraint

$$\Pi_l \leq \frac{\#count\{\mathbf{X} : C(\mathbf{X}), A(\mathbf{X})\}}{\#count\{\mathbf{X} : A(\mathbf{X})\}} \leq \Pi_u$$

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 384–386, doi:10.4204/EPTCS.385.41

is false should be excluded. Practically speaking, consider the following small knowledge base:

```
0.4::bird(1). 0.4::bird(2). 0.4::bird(3). 0.4::bird(4).
(fly(X)|bird(X))[0.6,1].
```

The first line contains four probabilistic facts stating that each of the four elements (indexed with 1, ..., 4) can be birds with probability 0.4. The statistical statement states that at least 60% (and at most 100%) of the birds fly. The statement is converted into two answer set rules:

```
fly(X) ; not_fly(X) :- bird(X).
:- #count{X:bird(X)}=H, #count{X:fly(X),bird(X)}=FH, 100*FH<60*H.
```

Only two rules suffice, since $\Pi_u = 1$. From this, we can compute the probability of a query, such as $fly(1)$. This requires checking its validity in the answer sets of each world. Since we are interested only in knowing if it holds in all the answer sets of a world or in at least one, and not in the number of answer sets, we can adopt projection [5]: every probabilistic fact $\Pi :: f$ is converted into three rules $f(PT) :- f.$, $nf(PF) :- not\ f.$, $\{f\}.$, where $PT = \Pi \cdot 10^n$ and $PF = (1-\Pi)\cdot 10^n$ (where n can be selected and is needed since ASP cannot deal with floating point values), and then we can compute the projective solutions on the query, $f/1$ and, $nf/1$ atoms. For the previous program, we get $P(fly(1)) = [0.2592, 0.4]$.

Acknowledgements

This research was partly supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No. 952215.

References

[1] Damiano Azzolini, Elena Bellodi & Fabrizio Riguzzi (2022): *Statistical Statements in Probabilistic Logic Programming*. In Georg Gottlob, Daniela Inclezan & Marco Maratea, editors: *Logic Programming and Nonmonotonic Reasoning*, Springer International Publishing, Cham, pp. 43–55, doi:10.1007/978-3-031-15707-3_4.

[2] Gerhard Brewka, Thomas Eiter & Mirosław Truszczyński (2011): *Answer Set Programming at a Glance*. *Communications of the ACM* 54(12), pp. 92–103, doi:10.1145/2043174.2043195.

[3] Fábio Gagliardi Cozman & Denis Deratani Mauá (2020): *The joy of Probabilistic Answer Set Programming: Semantics, complexity, expressivity, inference*. *International Journal of Approximate Reasoning* 125, pp. 218–239, doi:10.1016/j.ijar.2020.07.004.

[4] Luc De Raedt, Angelika Kimmig & Hannu Toivonen (2007): *ProbLog: A Probabilistic Prolog and Its Application in Link Discovery*. In Manuela M. Veloso, editor: *20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, 7, AAAI Press, pp. 2462–2467.

[5] Martin Gebser, Benjamin Kaufmann & Torsten Schaub (2009): *Solution Enumeration for Projected Boolean Search Problems*. In Willem-Jan van Hoeve & John Hooker, editors: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Springer Berlin Heidelberg, pp. 71–86, doi:10.1007/978-3-642-01929-6_7.

[6] Joseph Y. Halpern (1990): *An Analysis of First-Order Logics of Probability*. *Artificial Intelligence* 46(3), pp. 311–350, doi:10.1016/0004-3702(90)90019-V.

[7] Gabriele Kern-Isberner & Matthias Thimm (2010): *Novel Semantical Approaches to Relational Probabilistic Conditionals*. In: *Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning*, AAAI Press, pp. 382–392.

[8] Marco Wilhelm, Gabriele Kern-Isberner, Marc Finthammer & Christoph Beierle (2019): *Integrating Typed Model Counting into First-Order Maximum Entropy Computations and the Connection to Markov Logic Networks*. In Roman Barták & Keith W. Brawner, editors: *Proceedings of the Thirty-Second International Florida Artificial Intelligence Research Society Conference*, AAAI Press, pp. 494–499.

Extended Abstract:
Error-free Smart Legal Contracts without Programmers

Kevin Purnell
School of Computing
Macquarie University
Sydney, Australia
kevin.purnell@hdr.mq.edu.au

Rolf Schwitter
School of Computing
Macquarie University
Sydney, Australia
rolf.schwitter@mq.edu.au

1 Introduction

This research investigates the potential of declarative languages to advance what can be achieved by domain experts producing their own automation. We illustrate this potential by exploring the issue of domain experts implementing their own executable legal documents as both human- and machine-readable smart legal contracts deployable on a blockchain [7, 11].

Smart contracts have the potential to enable automation in previously automation-resistant areas (e.g., legal contracts and contract management). While some legal documents can probably never be fully automated (e.g., subjective performance criteria), many legal documents and contracts use forms of logic that are computable, can be agreed by electronic signature, have performance that can be delivered and administered electronically and use money as the consideration. The current reality; however, is that this type of automation is usually impractical because of: an inability to build custom smart contracts economically; the risk of errors and security breaches; and, inflexibility in regard to implementing contract modifications. These hurdles motivate the research question that we answer: Is it possible for **domain experts** to **design, build, test** and deploy their own **custom legal documents** as **smart contracts** which are **free from errors** and **security exploits** and can also support **contract modifications**?

After pure first-order logic was shown to be inadequate for expressing legal concepts [4], non-monotonic declarative languages like Answer Set Programming (ASP), TOAST, SPINdle [1], Turnip [2], and a controlled natural language, Logical English [6] were developed and used to model legal logic. Following this stream of research, we investigate the use of ASP for modelling legal logic by iteratively developing a proof-of-concept smart document editor (SDE editor) which reads an existing legal document (in HTML) and transforms it in five domain expert-guided steps into an error-free executable logic program [8, 9, 10]. Domain experts model the ontology, legal logic, generate the assertional data, and perform all testing tasks for amenable legal documents. The editor communicates visually and verbally in domain familiar concepts; a feature made possible by the formal correspondences achieved between graphics, text and declarative representations, and automation of steps not requiring domain knowledge. Our approach allows automatic construction of ASP code and enables highly automated testing.

2 System Description

The SDE editor uses ASP to represent the ontology and the legal processes derived from a legal document. Firstly, the artefacts in the document and their relationships are captured in representations by a process where three different notations (a term, a graphic and an ASP-like string) are allocated the same

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 387–389, doi:10.4204/EPTCS.385.42

meaning by the user. This formal correspondence and the use of a grammar facilitates the automatic generation of representations, rules and the automation of model validation and program verification. The user sees the ontology being built up as they work through the legal document, and can use ontology elements in rules which are then tested in dynamic visualisations. Other formal correspondences ease translation into structured English sentences that use the correct verb tense and correctly describe both general and instance descriptions of artefacts.

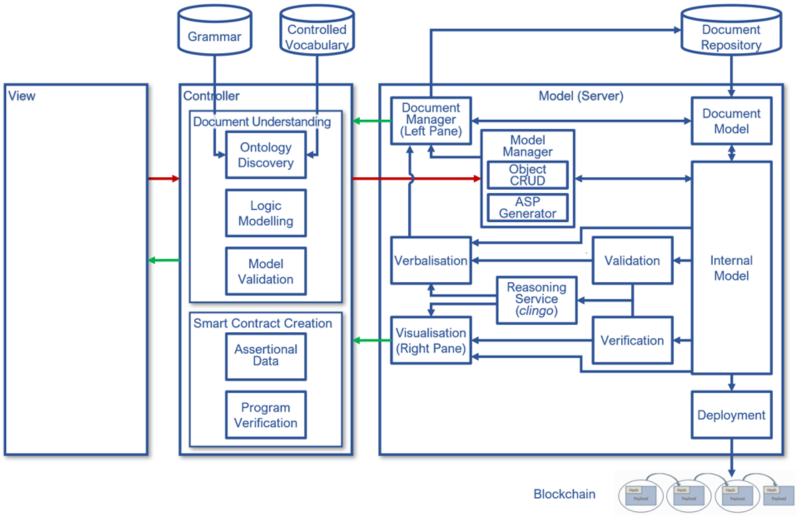


Figure 1: SDE Editor Logical Architecture.

Legal documents have evolved from paper documents filled out and signed by hand, into documents signed digitally and managed by tools like document automation systems [5] which provide support for drafting through to the administration of active contracts. The SDE editor (see Figure 1) adds the ability to first understand the closed world of a legal document, then assist in the modelling and testing of the embedded logic, resulting in an exhaustively tested logic program that can be deployed as a smart contract. This approach offers the potential of broader applicability and more comprehensive automation than current approaches. The workflow of the SDE editor can be split into two phases: 1) document understanding; and, 2) smart contract creation. Document understanding involves: (i) modelling the ontology (ontology discovery); (ii) modelling the legal logic (logic modelling); and, (iii) validating these models against the designer's understanding (model validation). Smart contract creation involves: (i) entering actual data (assertional data); and, (ii) testing that the output is what is expected across all possible inputs (program verification). These processes accumulate knowledge in representations that are consistent across the workflow due to auto-generation from ontology representations. The result is a tested ASP program embedded in the original completed legal document (i.e., a Ricardian Contract [3]).

3 Evaluation and Results

If the components of the research question are used as evaluation criteria then the SDE editor achieves most criteria. The SDE editor enables: 1) **design, build and testing** of suitable **legal documents** as **smart contracts** by **domain experts**; 2) practical **custom** smart legal contracts; 3) smart legal contracts that are **free from errors** from the perspective of the designer; 4) **reduction in security risk** by reducing unproven components in the toolchain; and, 5) practical smart legal **contract modifications**.

By using an executable declarative language to represent and reason over business/legal logic, the SDE editor overcomes most of the hurdles experienced with smart contracts coded in an imperative programming language like Solidity, and extends what practical automation is achievable for legal documents. The approach promises smart contracts that: 1) are designed, built and tested by domain experts; 2) are error-free (designer perspective); 3) explain their automatic decisions in controlled English in an audit log; 4) achieve practical customisation due to the ontology discovery step; 5) achieve practical contract modifications (legal sense) due to the use of an elaboration tolerant declarative language; 6) are flexible in regard to fitting with current practice and partial automation (uses existing legal documents and supports human-in-the-loop processes); 7) achieve user interaction in structured English and via visualisation with optional visual interaction; 8) encode business/legal logic in a declarative language with formal semantics, then verbalises the computed responses back into English; an approach that opens the way to achieving error-free smart legal contracts; 9) use a smart assistant to reduce user workload; and, 10) achieve simplified representations of complex logic and legal reasoning (defeasible logic); a capability beyond current smart contract programming languages, and which extends applicability.

4 Future Work

Recent machine learning advances allow a more fluent user-interface experience than achieved by the SDE editor; however, these approaches struggle to produce explainable solutions and generate error-free programs. Enhancing the current smart assistant with machine learning capabilities could further simplify the user interface and extend maximum practical program size by automating tedious user interactions. We also believe that the proof-of-concept system is ready to be applied to a difficult problem of significance; however, two open questions remain: the degree to which ASP can express the full spectrum of deontic logic; and, the percentage of legal documents that contain logic inexpressible in ASP.

References

[1] Batsakis et al. (2018): *Legal Representation and Reasoning in Practice: A Critical Comparison*. In: *Proceedings of JURIX 2018*, pp. 31–40, doi:10.3233/978-1-61499-935-5-31.

[2] Governatori et al. (2020): *On the Formal Representation of the Australian Spent Conviction Scheme*. In: *Proceedings of RuleML+RR 2020*, pp. 177–185, doi:10.1007/978-3-030-57977-7_14.

[3] Grigg (2004): *The Ricardian Contract*. *Proceedings of IEEE WEC*, doi:10.1109/WEC.2004.1319505.

[4] Herrestad (1991): *Norms and Formalization*. *Proceedings of ICAIL '91*, doi:10.1145/112646.112667.

[5] HighQ (2022): *Automation*. Available at `https://legal.thomsonreuters.com/en/products/highq/`.

[6] Kowalski (2022): *Logical English meets legal English for swaps and derivatives*. *AI Law*, doi:10.1007/s10506-021-09295-3.

[7] Nakamoto (2008): *Bitcoin*. Available at `https://bitcoin.org/bitcoin.pdf`.

[8] Purnell & Schwitter (2021): *User-Defined Smart Contracts Using Answer Set Programming*. In: *LNCS, Advances in AI, AI 2021*, 13151, Springer, pp. 291–303, doi:10.1007/978-3-030-97546-3_24.

[9] Purnell & Schwitter (2021): *User-Guided Machine Understanding of Legal Documents*. In: *LNCS, New Frontiers in AI, JURISIN 2021*, 13856, Springer, pp. 16–32, doi:10.1007/978-3-031-36190-6.

[10] Purnell & Schwitter (2022): *Declarative Smart Contract Testing by Domain Experts*. In: *CEUR Workshop Proceedings, RuleML+RR 2022 DC*, 3229. Available at `https://dblp.org/db/conf/ruleml/ruleml2022c.html`.

[11] Wood (2014): *Ethereum*. Available at `https://gavwood.com/paper.pdf`.

Axiomatization of Aggregates: Extended Abstract

Jorge Fandinno
University of Nebraska Omaha
Nebraska, USA
`jfandinno@unomaha.edu`

Zachary Hansen
University of Nebraska Omaha
Nebraska, USA
`zachhansen@unomaha.edu`

Yuliya Lierler
University of Nebraska Omaha
Nebraska, USA
`ylierler@unomaha.edu`

This paper presents semantics for logic programs with aggregates that refer to neither fixpoints nor grounding, relying instead on transformations into first and second-order theories. This characterization introduces new function symbols to represent aggregates, whose meaning can be fixed by adding appropriate axioms to the result of the proposed transformations. These semantics provide a foundation for reasoning formally about non-ground programs with aggregates in a direct manner. This is useful for the task of formally verifying logic programs, and represents a first step towards using first-order theorem provers to automatically verify properties of programs with aggregates. For programs without positive recursion through aggregates our semantics coincide with the semantics of the answer set solver `clingo`.

Introduction Answer set programming (ASP) is a widely-utilized logic programming paradigm encompassing a number of highly expressive modeling languages and efficient solvers. Aggregates are one of the most useful ASP language constructs, and one of the most well-studied. This is due, in part, to the difficulty of defining a universally accepted semantics for aggregates [6]. Most approaches to defining aggregate semantics rely upon grounding, a process by which programs with variables are transformed into propositional programs. There are also several approaches to describe the semantics of aggregates that bypass the need for grounding. However, these approaches either only allow a restricted class of aggregates [7, 8] or extend the language of classical logic with new constructs to deal with aggregates [1, 2, 3]. Our approach uses only the language of classical logic as in [7, 8], but it is applicable to all programs with aggregates that fit within the restrictions of the ASP-Core-2 semantics, namely, that aggregates are non-recursive. Exclusively using the language of classical logic opens the door for automated verification with tools such as `anthem` [4].

In the following section, we summarize our translation from logic programs with aggregates to second-order logic formulas. We restrict our attention to a simple subset of the `gringo` input language which forbids nested aggregates and positive recursion through aggregates. We then show how the addition of second-order axioms can fix the interpretation of aggregate function symbols in a way that conforms to their behavior under the semantics implemented by the answer set solver `clingo`. Finally, we demonstrate that this second-order characterization can be replaced with a first-order one when we consider only programs with finite aggregates.[1]

Syntax of Logic Programs With Aggregates We consider a basic fragment of the `gringo` input language, with program terms ranging over *symbolic constants*, *program variables*, special terms *inf* and *sup*, and *numerals* (which have a one-to-one correspondence with integers). *Atoms* and *comparisons* are defined as usual in ASP. Basic literals are atoms and comparisons, optionally preceded by one or two occurrences of *not*. An *aggregate element* has the form

$$t_1, \dots, t_k : l_1, \dots, l_m \tag{1}$$

---

[1] It is worth noting that this is not a practical restriction due to the inability of solvers to process infinite sets.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 390–392, doi:10.4204/EPTCS.385.43

where each t_i $(1 \leq i \leq k)$ is a program term and each l_i $(1 \leq i \leq m)$ is a basic literal. An aggregate element can be informally understood as representing the set of tuples of program terms $\langle t_1, \dots, t_k \rangle$ satisfying the list of conditions $l_1, \dots, l_m$. An *aggregate atom* is of form $\#\mathtt{op}\{E\} \prec u$ where `op` is an operation name, E is an aggregate element, $\prec$ is a comparison symbol, and u is a program term, called the *guard*. We consider operation names `count`, `sum`, `sum+`, `min`, and `max`. Our task in this work is to formalize the intuition that these operations characterize functions on the sets of tuples represented by aggregate elements. Finally, programs are defined as a collection of basic rules (containing a single atom in the head) and constraints.

Semantics of Logic Programs With Aggregates For each rule R in a program Π, we obtain its many-sorted first-order formula representation by applying a translation $\tau^*_{\mathbf{Z}}$, where $\mathbf{Z}$ is the list of global variables[2] in R. Broadly speaking, this is a syntactic transformation that maps default negation into classical negation, commas into conjunctions, and the rule operator :- into an implication from the body of the rule to the head. For an aggregate atom A of form $\#\mathtt{op}\{E\} \prec u$, its translation $\tau^*_{\mathbf{Z}}$ is the atom $op(set_{|E/\mathbf{X}|}(\mathbf{X})) \prec u$, where $\mathbf{X}$ is the list of variables in $\mathbf{Z}$ occurring in E. For every rule R, its translation $\tau^* R$ is the universal closure of the implication

$$\tau^*_{\mathbf{Z}} B_1 \wedge \cdots \wedge \tau^*_{\mathbf{Z}} B_n \rightarrow \tau^*_{\mathbf{Z}} H,$$

where $\mathbf{Z}$ is the list of the global variables of R. For example, given an aggregate symbol $Y : r(X,Y,Z)/X$ named $e1$,

$$\tau^*\left(s(X) \text{ :- } q(X), \#\mathtt{sum}\{Y : r(X,Y,Z)\} \geq 1.\right) \quad \text{is} \quad \forall X \left(q(X) \wedge sum(set_{e1}(X)) \geq 1 \ \rightarrow\ s(X)\right).$$

Note that this is a formula over a signature of many sorts. Specifically, we consider separate domains (universes) for program terms, integers, tuples (of program terms), and sets of tuples. The second domain is a subset of the first, and the fourth domain is the powerset of the third. Assumptions about the forms of these domains and the behavior of fundamental functions and predicates such as addition and set membership are defined in our notion of *standard interpretations*. These assumptions do not fix the interpretation of aggregate elements or the function symbols corresponding to aggregate operations. To properly characterize the behavior of aggregates, we require our standard interpretations to additionally satisfy a set of second-order axioms Δ (introduced in the next section). Finally, we define the semantics of logic programs with aggregates using a many-sorted generalization of the SM operator [5]. We say that a standard interpretation I is a *stable model* of program Π if it satisfies the second-order sentence $\mathrm{SM}_{\mathbf{p}}[\tau^*\Pi] \wedge \Delta$ with $\mathbf{p}$ being the list of all predicate symbols occurring in Π.

Axiomatization of Aggregates In this section, we will highlight a few representative axioms in set Δ. In general, this set contains second-order axioms; however, assuming our aggregates are finite allows us to replace the second-order axioms with first-order ones. First, we introduce the class of *set formation axioms*, which formally relate aggregate elements to sets of tuples. An aggregate symbol $E/\mathbf{X}$, where E is an aggregate element, is associated with a unique set characterized by the sentence

$$\forall \mathbf{X}\, T \big(T \in set_{|E/\mathbf{X}|}(\mathbf{X}) \leftrightarrow \exists \mathbf{Y}\ \big(T = tuple_k(t_1, \dots, t_k) \wedge l_1 \wedge \cdots \wedge l_m\big)\big),$$

where $\mathbf{Y}$ is the list of all the variables occurring in E that are not in $\mathbf{X}$. All variables in $\mathbf{Y}$ are of the sort program. T is a variable of the sort tuple. For instance, recall the aggregate symbol $e1$ introduced earlier. The set formation axiom for this symbol has the form

$$\forall X T \big(T \in set_{e1}(X) \leftrightarrow \exists Y Z \big(T = tuple_2(Y,Z) \wedge r(X,Y,Z)\big)\big)$$

---

[2] Variables occurring in basic literals or the guards of aggregates are global.

Similarly, we can characterize the removal of an element from a set with function symbol *rem*:

$$\forall S T S' \big(rem(S,T) = S' \leftrightarrow \forall T' \big(T' \in S' \leftrightarrow (T' \in S \wedge T' \neq T) \big) \big)$$

The `sum` aggregate can be formalized for finite aggregates with the axiom

$$\forall \mathbf{X}\, S \big(\big(\forall T \, \big(T \in S \rightarrow T \in set_{|E/\mathbf{X}|}(\mathbf{X}) \big) \big) \rightarrow FiniteSum(S) \big)$$

where $FiniteSum(S)$ is shorthand for

$$\forall T \big(T \in t_{set} \rightarrow \exists N (sum(rem(t_{set}, T)) = N \wedge sum(t_{set}) = N + weight(T)) \big)$$

Finally, when Π is tight, we can replace the second-order sentence $\mathrm{SM}_{\mathbf{p}}[\tau^{*}\Pi]$ with a first-order sentence using program completion. This gives us a fully first-order characterization.

Final Remarks The proposed semantics characterize logic programs with aggregates in terms of classical logic, allowing us to reason directly about the meaning of non-ground programs. This approach is highly general: the restrictions prohibiting nested aggregates and positive recursion through aggregates are less restrictive than those imposed by existing solvers and the ASP-Core-2 standard. Additionally, the assumptions made by standard interpretations, such as the assumption that the $+$ symbol behaves as addition, are relatively modest. Nevertheless, the task of encoding such assumptions within automated theorem provers is a non-trivial task, and a direction for future work.

References

[1] Vernon Asuncion, Yin Chen, Yan Zhang & Yi Zhou (2015): *Ordered completion for logic programs with aggregates*. *Artificial Intelligence* 224, p. 72–102, doi:10.1016/j.artint.2015.03.007.

[2] Michael Bartholomew, Joohyung Lee & Yunsong Meng (2011): *First-Order Semantics of Aggregates in Answer Set Programming Via Modified Circumscription*. In E. Davis, P. Doherty & E. Erdem, editors: *Proceedings of the AAAI Spring Symposium on Logical Formalizations of Commonsense Reasoning*, AAAI Press, pp. 16–22.

[3] Pedro Cabalar, Jorge Fandinno, Luis Fariñas Del Cerro & David Pearce (2018): *Functional ASP with Intensional Sets: Application to Gelfond-Zhang Aggregates*. *Theory and Practice of Logic Programming* 18(3–4), p. 390–405, doi:10.1017/S1471068418000169.

[4] Jorge Fandinno, Vladimir Lifschitz, Patrick Lühne & Torsten Schaub (2020): *Verifying Tight Logic Programs with anthem and vampire*. *Theory and Practice of Logic Programming* 20(5), p. 735–750, doi:10.1017/S1471068420000344.

[5] Paolo Ferraris, Joohyung Lee & Vladimir Lifschitz (2011): *Stable models and circumscription*. *Artificial Intelligence* 175(1), p. 236–263, doi:10.1016/j.artint.2010.04.011.

[6] Michael Gelfond & Yuanlin Zhang (2019): *Vicious circle principle, aggregates, and formation of sets in ASP based languages*. *Artificial Intelligence* 275, p. 28–77, doi:10.1016/j.artint.2019.04.004.

[7] Joohyung Lee, Vladimir Lifschitz & Ravi Palla (2008): *A Reductive Semantics for Counting and Choice in Answer Set Programming*. In D. Fox & C. Gomes, editors: *Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI'08)*, AAAI Press, pp. 472–479.

[8] Vladimir Lifschitz (2022): *Strong Equivalence of Logic Programs with Counting*. *Theory and Practice of Logic Programming* 22(4), p. 573–588, doi:10.1017/S1471068422000278.

Visual Sensemaking Needs Both Vision and Semantics

On Logic-Based Declarative Neurosymbolism for Reasoning about Space and Motion

Jakob Suchan
German Aerospace Research (DLR)
Germany
jakob.suchan@dlr.de

Mehul Bhatt
Örebro University, Sweden
CoDesign Lab EU
mehul.bhatt@oru.se

Srikrishna Varadarajan
CoDesign Lab EU
Europe
krish@codesign-lab.org

Contemporary artificial vision systems lack abilities for high-level, human-scale mental simulation, e.g., concerning perceived everyday multimodal interactions. Cognitively driven sensemaking functions such as embodied grounding for active vision, visuospatial concept formation, commonsense explanation, diagnostic introspection all remain fertile ground. We posit that developing high-level visual sensemaking capabilities requires a systematic, tight yet modular integration of (neural) visual processing techniques with high-level commonsense knowledge representation and reasoning methods pertaining to space, motion, actions, events, conceptual knowledge etc. As an exemplar of this thinking, we position recent work on deeply semantic, explainable, neurosymbolic visuospatial reasoning driven by an integration of methods in (deep learning based) *vision* and (KR based) *semantics*. The positioned work is general, but its significance is demonstrated and empirically benchmarked in the context of (active, realtime) visual sensemaking for self-driving vehicles.

Visual Sensemaking: A Human-Centred Cognitive Perspective

Artificial visual intelligence [5] denotes the computational capability to semantically process, interpret, and explain diverse forms of visual stimuli (typically) emanating from sensing embodied multimodal interaction of/amongst humans and other artefacts in diverse naturalistic situations of everyday life and profession. Through semantic processing, interpretation, and explanation, alluded here are a wide-spectrum of high-level human-centred *sensemaking* capabilities. These capabilities encompass functions such as visuospatial conception formation, commonsense/qualitative generalisation, hypothetical reasoning, analogical inference, argumentation, event based episodic maintenance & retrieval for perceptual narrativisation, counterfactual reasoning and explanation etc. In essence, in scope are all high-level commonsense visuospatial sensemaking capabilities –be it mundane, analytical, or creative in nature– that humans acquire developmentally or through specialised training, and are routinely adept at performing seamlessly in their everyday life and work.

The Need for Integrated "Vision and Semantics". Computational visual sensemaking requires a systematically developed general and modular integration of high-level techniques concerned with "commonsense and semantics" with low-level neural methods capable of computing primitive features of interest in visual data. Realising computational visual sensemaking —be it realtime or post-hoc— in view of human-centred AI concerns pertaining to explainability, ethics, regulation and requires a systematic (neurosymbolic) integration of Vision and Semantics, i.e., robust commonsense representation & inference about spacetime dynamics on the one hand, and powerful low-level visual computing capabilities, e.g., pertaining to object detection and tracking on the other. It is also critical that the semantics of formal representations, e.g., of space and motion, be rooted to counterparts in natural language [4]. Towards this, the positioned research [11] demonstrates the significance of semantically-driven methods

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 393–395, doi:10.4204/EPTCS.385.44

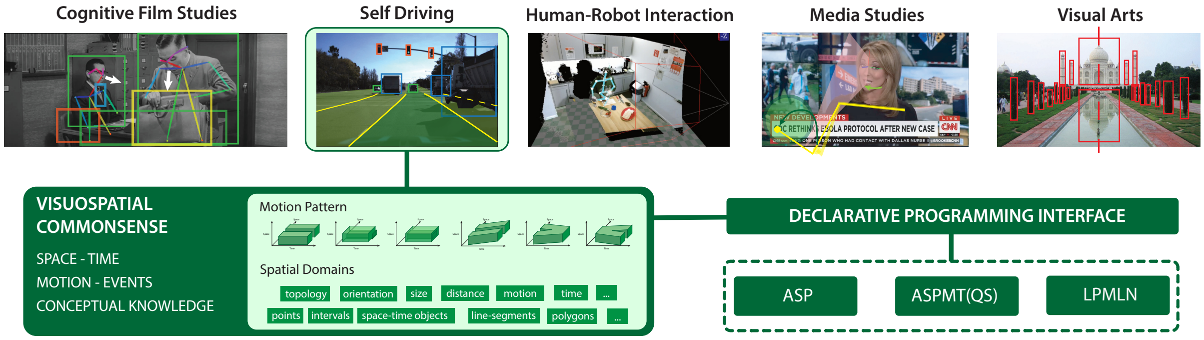


Figure 1: Application Scenarios: Autonomous Systems, Minds and Media Studies, Behavioural Research in Multimodality Interaction [5].

rooted in knowledge representation and reasoning (KR) in addressing research questions pertaining to explainability and human-centred AI particularly from the viewpoint of (perceptual) sensemaking of dynamic visual imagery. This goal is pursued within the larger agenda of *cognitive vision and perception* [5], which is an emerging line of research bringing together a novel & unique combination of methodologies from Artificial Intelligence, Vision and Machine Learning, Cognitive Science and Psychology, Visual Perception, and Spatial Cognition and Computation (also, [9, 10, 11, 12], and [7]). Research in cognitive vision and perception addresses visual, visuospatial and visuo-locomotive perception and interaction from the viewpoints of language, logic, spatial cognition and artificial intelligence [4].

Visuospatial Commonsense, Driven by Answer Set Programming

Answer Set Programming (ASP) is now widely used as a foundational declarative language and robust methodology for a range of (non-monotonic) knowledge representation and reasoning tasks [6, 8]. With ASP as a foundation, and motivated by requirements of semantics, commonsense and explainability, our work bridges the gap between high-level formalisms for logical visual reasoning (e.g., by abduction) and low-level visual processing[1] by tightly integrating semantic abstractions of *space and motion* with their underlying numerical representations within the declarative framework of ASP. Furthermore, directly within ASP, we also address several challenges concerning *epistemological* and *phenomenological* aspects relevant to a wide range of *dynamic spatial systems* [1, 2, 3]: projection and interpolation, object identity maintenance at a semantic level, ability to make default assumptions, maintaining consistent beliefs etc.

Although the work positioned herein [10, 11] selectively focusses on the needs and challenges of active / online sensemaking in autonomous driving, the generality and modularly of the developed ASP-based neurosymbolic framework ensures foundational applicability in diverse applied contexts requiring perceptual interpretation, interaction and control functions. Of at least equal importance are the modularity and elaboration tolerance of the framework, enabling seamless integration and experimentation with advances in (fast evolving) computer vision methods, as well as experimenting with different forms of formal methods for (declarative) *reasoning about space, actions, and change* [1].

---

[1] Visual processing encompasses capabilities including but not limited to motion analysis, object detection, pose estimation, semantic segmentation, image classification. A detailed review is available in [5].

References

[1] Mehul Bhatt (2012): *Reasoning about Space, Actions and Change: A Paradigm for Applications of Spatial Reasoning*. In: *Qualitative Spatial Representation and Reasoning: Trends and Future Directions*, IGI Global, USA, doi:10.4018/978-1-4666-4607-0.ch016.

[2] Mehul Bhatt, Hans W. Guesgen, Stefan Wölfl & Shyamanta M. Hazarika (2011): *Qualitative Spatial and Temporal Reasoning: Emerging Applications, Trends, and Directions*. *Spatial Cognition & Computation* 11(1), pp. 1–14, doi:10.1080/13875868.2010.548568.

[3] Mehul Bhatt & Seng W. Loke (2008): *Modelling Dynamic Spatial Systems in the Situation Calculus*. *Spatial Cognition & Computation* 8(1-2), pp. 86–130, doi:10.1080/13875860801926884.

[4] Mehul Bhatt, Carl Schultz & Christian Freksa (2012): *The 'Space' in Spatial Assistance Systems: Conception, Formalisation and Computation*. In Thora Tenbrink, Jan Wiener & Christophe Claramunt, editors: *Representing space in cognition: Interrelations of behavior, language, and formal models. Series: Explorations in Language and Space*, Oxford University Press, doi:10.1093/acprof:oso/9780199679911.003.0009.

[5] Mehul Bhatt & Jakob Suchan (2023): *Artificial Visual Intelligence: Perceptual Commonsense for Human-Centred Cognitive Technologies*, pp. 216–242. Springer International Publishing, Cham, doi:10.1007/978-3-031-24349-3_12.

[6] Gerhard Brewka, Thomas Eiter & Miroslaw Truszczyński (2011): *Answer Set Programming at a Glance*. *Commun. ACM* 54(12), pp. 92–103, doi:10.1145/2043174.2043195.

[7] David Marr (1982): *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. Henry Holt and Co., Inc., New York, NY, USA.
(Republished 2010, MIT Press, doi: 10.7551/mitpress/9780262514620.001.0001.)

[8] Torsten Schaub & Stefan Woltran (2018): *Special Issue on Answer Set Programming*. *KI* 32(2-3), pp. 101–103, doi:10.1007/s13218-018-0554-8.

[9] Jakob Suchan & Mehul Bhatt (2016): *Semantic Question-Answering with Video and Eye-Tracking Data: AI Foundations for Human Visual Perception Driven Cognitive Film Studies*. In Subbarao Kambhampati, editor: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, IJCAI/AAAI Press, pp. 2633–2639. Available at `http://www.ijcai.org/Abstract/16/374`.

[10] Jakob Suchan, Mehul Bhatt & Srikrishna Varadarajan (2019): *Out of Sight But Not Out of Mind: An Answer Set Programming Based Online Abduction Framework for Visual Sensemaking in Autonomous Driving*. In Sarit Kraus, editor: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, ijcai.org, pp. 1879–1885, doi:10.24963/ijcai.2019/260.

[11] Jakob Suchan, Mehul Bhatt & Srikrishna Varadarajan (2021): *Commonsense visual sensemaking for autonomous driving - On generalised neurosymbolic online abduction integrating vision and semantics*. *Artif. Intell.* 299, p. 103522, doi:10.1016/j.artint.2021.103522.

[12] Jakob Suchan, Mehul Bhatt, Przemyslaw Andrzej Walega & Carl Schultz (2018): *Visual Explanation by High-Level Abduction: On Answer-Set Programming Driven Reasoning About Moving Objects*. In Sheila A. McIlraith & Kilian Q. Weinberger, editors: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, AAAI Press, pp. 1965–1972, doi:10.1609/aaai.v32i1.11569.

Flexible Job-shop Scheduling for Semiconductor Manufacturing with Hybrid Answer Set Programming (Extended Abstract) [*]

Ramsha Ali
University of Klagenfurt, Austria
`ramsha.ali@aau.at`

Mohammed M. S. El-Kholany
University of Klagenfurt, Austria
Cairo University, Egypt
`mohammed.el-kholany@aau.at`

Martin Gebser
University of Klagenfurt, Austria
Graz University of Technology, Austria
`martin.gebser@aau.at`

Modern semiconductor manufacturing involves complex production processes with hundreds of operations from lot release to completion, which often takes several months. In contrast to abstract job-shop scheduling benchmarks, our work addresses the scheduling of realistic semiconductor manufacturing processes. We model their sophisticated requirements by hybrid Answer Set Programming and take advantage of difference logic for expressing machine processing, setup, and maintenance times. Unlike existing approaches that schedule semiconductor manufacturing processes locally by greedy heuristics or isolated optimization of specific machine groups, we explore the potential and limits of large-scale semiconductor fab scheduling.

1 Introduction

Classical and Flexible Job-shop Scheduling Problems (FJSPs) [17, 6], focusing on the allocation of production tasks and resource planning, have been targeted bibliographystyle various local and exact search methods (see [7] for an overview). The importance of optimization in production and supply chain management has grown due to diversified technologies, product customization, quality-of-service targets, and ecological footprint. However, abstract scheduling benchmarks do not adequately represent the complex production processes found in semiconductor manufacturing, facing long production routes, product-specific machine setups, dedicated maintenance procedures, and stochastic factors. These challenges have led to the development of the recent SMT2020 simulation scenario [14], modeling modern semiconductor manufacturing processes and facilities. Currently, production management is performed locally using preconfigured dispatching rules [12] or machine-learned decision making policies [19]. Exact search and optimization frameworks like Constraint, Integer and Answer Set Programming (ASP) [15] at least in principle enable global approaches to resource allocation and operation scheduling that take the entire range of viable schedules into account and guarantee optimality to the extent feasible. For instance, ASP has been successfully applied for scheduling printing devices [4], specialist teams [16], work shifts [2], course timetables [5], medical treatments [8], and aircraft routes [18]. In particular, the hybrid framework of ASP modulo difference logic [13] specifically supports a compact representation and reasoning with quantitative resources like time, which has been exploited in domains such as lab resource [10], train connection [1], and classical job-shop [9] scheduling.

In our work [3], we use ASP modulo difference logic to model the scheduling of semiconductor manufacturing processes, particularly accounting for machine processing, setup, and maintenance times

---

[*] This work, providing an extended abstract of our PADL 2023 paper [3], was partially funded by FFG project 894072 (SwarmIn) as well as KWF project 28472, cms electronics GmbH, FunderMax GmbH, Hirsch Armbänder GmbH, incubed IT GmbH, Infineon Technologies Austria AG, Isovolta AG, Kostwein Holding GmbH, and Privatstiftung Kärntner Sparkasse.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 396–398, doi:10.4204/EPTCS.385.45

present in the SMT2020 scenario. Although long-term, global optimization of production routes with hundreds of operations is prohibitive due to problem size and unpredictable stochastic factors, we lay the ground for a future integration of more informed exact scheduling techniques with simulation and reactive decision making methods, enabling globally optimized (re)scheduling for a certain time in advance.

2 Application Scenario

The recent SMT2020 simulation scenario specifies two kinds of semiconductor fab settings, one denoted High-Volume/Low-Mix (HV/LM) with two types of products and the other Low-Volume/High-Mix (LV/HM) with ten kinds of products. Both have in common that, depending on the product type, the production route of each lot includes many operations, i.e., between 300 and 600 of them. Each operation needs some machine in a given tool group to be performed, which may have to be equipped with a product-specific setup before processing the operation. Moreover, periodic maintenance procedures must be applied after a certain number of operations or accumulating some amount of processing time. The overall objective is to minimize the completion time of lots by making effective use of fab resources.

We represent problem instances in terms of specific facts along with a general ASP modulo difference logic encoding. Our problem encoding supports two assignment strategies: fully flexible or fixed, the latter based on the lexicographic order of operations requiring machines in the same tool group. Setup times and maintenance procedures are also taken into account in scheduling the processing of operations.

3 Experimental Evaluation and Conclusion

We extracted test instances for scheduling from the STM2020 scenario by considering a part of the available tool groups along with their setups and maintenances. Our comparison includes the fixed and flexible machine assignment strategies, for which Table 1 reports the obtained makespan, solving time, conflicts and constraints, where 'TO' stands for more than 600 seconds solving time of clingo[DL] [13].

With the fixed machine assignment, we observe that the transition from trivial runs taking less than a second to aborts is sharp when increasing the number of operations, as the number of search conflicts goes up. A similar behavior is encountered even earlier, i.e., on smaller instances, with the flexible machine assignment, where the up to one order of magnitude greater number of constraints also yields a significantly increased problem size. Our hybrid ASP encoding for semiconductor fab scheduling reflects work in progress. As future work, we target the improvement of space and search efficiency, addition of yet missing capabilities like batch processing and operation pipelining, and integration with simulation.

	Fixed Machine Assignment				**Flexible Machine Assignment**			
OxM	**Makespan**	**Time**	**Conflicts**	**Constraints**	**Makespan**	**Time**	**Conflicts**	**Constraints**
10x2	120	<1	44	2249	120	<1	44	2258
15x3	169	<1	7910	8980	169	<1	16562	12916
20x3	224	TO	6091128	23751	204	TO	5869538	33682
20x4	120	<1	88	4569	105	15	162085	45364
25x4	171	<1	16303	15862	160	TO	2656812	89214
30x4	219	405	2626786	32872	227	TO	2868459	157258
30x5	191	93	1027764	26460	169	TO	2702824	159548
30x6	120	<1	128	6828	153	TO	2250537	158243
35x6	142	<1	925	11526	123	TO	2947938	181867
40x6	192	18	189240	30657	210	TO	1711929	387488
45x6	226	TO	3553153	49191	300	TO	1226970	554625

Table 1: Results for scheduling instances with fixed and flexible machine assignment

References

[1] D. Abels, J. Jordi, M. Ostrowski, T. Schaub, A. Toletti & P. Wanko (2021): *Train scheduling with hybrid answer set programming*. *TPLP* 21(3), pp. 317–347, doi:`10.1017/S1471068420000046`.

[2] M. Abseher, M. Gebser, N. Musliu, T. Schaub & S. Woltran (2016): *Shift design with answer set programming*. *Fundam. Informaticae* 147(1), pp. 1–25, doi:`10.3233/FI-2016-1396`.

[3] R. Ali, M. M. S. El-Kholany & M. Gebser (2023): *Flexible job-shop scheduling for semiconductor manufacturing with hybrid answer set programming (application paper)*. In: *PADL*, Springer, pp. 85–95, doi:`10.1007/978-3-031-24841-2_6`.

[4] M. Balduccini (2011): *Industrial-size scheduling with ASP+CP*. In: *LPNMR*, Springer, pp. 284–296, doi:`10.1007/978-3-642-20895-9_33`.

[5] M. Banbara, K. Inoue, B. Kaufmann, T. Okimoto, T. Schaub, T. Soh, N. Tamura & P. Wanko (2019): *teaspoon: Solving the curriculum-based course timetabling problems with answer set programming*. *Ann. Oper. Res.* 275(1), pp. 3–37, doi:`10.1007/s10479-018-2757-7`.

[6] P. Brucker & R. Schlie (1990): *Job-shop scheduling with multi-purpose machines*. *Computing* 45(4), pp. 369–375, doi:`10.1007/BF02238804`.

[7] I. A. Chaudhry & A. A. Khan (2016): *A research survey: Review of flexible job shop scheduling techniques*. *ITOR* 23(3), pp. 551–591, doi:`10.1111/itor.12199`.

[8] C. Dodaro, G. Galatà, A. Grioni, M. Maratea, M. Mochi & I. Porro (2021): *An ASP-based solution to the chemotherapy treatment scheduling problem*. *TPLP* 21(6), pp. 835–851, doi:`10.1017/S1471068421000363`.

[9] M. M. S. El-Kholany, M. Gebser & K. Schekotihin (2022): *Problem decomposition and multi-shot ASP solving for job-shop scheduling*. *TPLP* 22(4), pp. 623–639, doi:`10.1017/S1471068422000217`.

[10] G. Francescutto, K. Schekotihin & M. M. S. El-Kholany (2021): *Solving a multi-resource partial-ordering flexible variant of the job-shop scheduling problem with hybrid ASP*. In: *JELIA*, Springer, pp. 313–328, doi:`10.1007/978-3-030-75775-5_21`.

[11] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, S. Thiele & P. Wanko (2019): *Potassco user guide*. `https://potassco.org`.

[12] O. Holthaus & C. Rajendran (1997): *Efficient dispatching rules for scheduling in a job shop*. *Int. J. Production Economics* 48(1), pp. 87–105, doi:`10.1016/S0925-5273(96)00068-0`.

[13] T. Janhunen, R. Kaminski, M. Ostrowski, S. Schellhorn, P. Wanko & T. Schaub (2017): *Clingo goes linear constraints over reals and integers*. *TPLP* 17(5-6), pp. 872–888, doi:`10.1017/S1471068417000242`.

[14] D. Kopp, M. Hassoun, A. Kalir & L. Mönch (2020): *SMT2020—A semiconductor manufacturing testbed*. *T-SCM* 33(4), pp. 522–531, doi:`10.1109/TSM.2020.3001933`.

[15] V. Lifschitz (2019): *Answer set programming*. Springer, doi:`10.1007/978-3-030-24658-7`.

[16] F. Ricca, G. Grasso, M. Alviano, M. Manna, V. Lio, S. Iiritano & N. Leone (2012): *Team-building with answer set programming in the Gioia-Tauro seaport*. *TPLP* 12(3), pp. 361–381, doi:`10.1017/S147106841100007X`.

[17] E. Taillard (1993): *Benchmarks for basic scheduling problems*. *Eur. J. Oper. Res.* 64(2), pp. 278–285, doi:`10.1016/0377-2217(93)90182-M`.

[18] P. Tassel & M. Rbaia (2021): *A multi-shot ASP encoding for the aircraft routing and maintenance planning problem*. In: *JELIA*, Springer, pp. 442–457, doi:`10.1007/978-3-030-75775-5_30`.

[19] B. Waschneck, A. Reichstaller, L. Belzner, T. Altenmüller, T. Bauernhansl, A. Knapp & A. Kyek (2018): *Optimization of global production scheduling with deep reinforcement learning*. *Procedia CIRP* 72, pp. 1264–1269, doi:`10.1016/j.procir.2018.03.212`.

Body-Decoupled Grounding via Solving: A Novel Approach on the ASP Bottleneck (Extended Abstract)*

Viktor Besin
TU Wien, Austria
vbesin@dbai.tuwien.ac.at

Markus Hecher
Massachusetts Institute of Technology, USA
hecher@mit.edu

Stefan Woltran
TU Wien, Austria
woltran@dbai.tuwien.ac.at

Answer-Set Programming (ASP) has seen tremendous progress over the last two decades and is nowadays successfully applied in many real-world domains. However, for certain problems, the well-known ASP grounding bottleneck still causes severe problems. This becomes virulent when grounding of rules, where the variables have to be replaced by constants, leads to a ground program that is too huge to be processed by the ASP solver. In this work, we tackle this problem by a novel method that decouples non-ground atoms in rules in order to delegate the evaluation of rule bodies to the solving process. Our procedure translates a non-ground normal program into a ground disjunctive program that is exponential only in the maximum predicate arity, and thus polynomial if this arity is bounded by a constant. We demonstrate the feasibility of this new method experimentally by comparing it to standard ASP technology in terms of grounding size, grounding time and runtime.

1 Body-Decoupled Grounding

Motivated by the ASP *grounding bottleneck* [3, 6], the problem of traditional grounding systems resulting in exponentially large programs when instantiating non-ground rules (even for programs with bounded predicate arities), we briefly tease the concept of *body-decoupled* grounding. The idea of this approach is to reduce the grounding size and grounding time by decoupling dependencies between different predicates of rule bodies. The potential of this is motivated by the following example.

Example 1. *Assume the following non-ground program* Π *that decides in (1) for each edge (e) of a given graph, whether to pick it (p) or not* ($\bar{p}$)*. Then, in (2) it is ensured that the choice of edges does not form triangles.*

$$p(A,B) \vee \bar{p}(A,B) \leftarrow e(A,B) \tag{1}$$

$$\leftarrow p(X,Y), p(Y,Z), p(X,Z), X \neq Y, Y \neq Z, X \neq Z. \tag{2}$$

The typical grounding effort of (2) is in $\mathcal{O}(|\mathrm{dom}(\Pi)|^3)$*. Our approach grounds body predicates of (2) individually, yielding groundings that are linear in the size of the ground atoms. In our example, this corresponds to* $\mathcal{O}(|\mathrm{dom}(\Pi)|^2)$ *due to arity* 2.

	Ground	Non-Ground (bounded arity)
Tight/Normal Programs	NP-c	Σ_2^P-c
Disjunctive Programs	Σ_2^P-c	Σ_3^P-c

Table 1: Known complexity results for some of the program types.

---

[*] This is an extended abstract of a paper [1] that appeared at IJCAI'22. Research was funded by Vienna Science and Technology Fund (WWTF) grant ICT19-065 and Austrian Science Fund (FWF) grants J 4656 and P 32830.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 399–401, doi:10.4204/EPTCS.385.46

Based on earlier complexity results for ground and non-ground logic programs [2, 5, 4], we introduce a reduction-based translation from non-ground, tight (and normal) programs to ground, disjunctive programs (see arrow in Tab. 1), resulting in an alternative grounding procedure. Our encodings translate a non-ground rule by (i) guessing whether the head atom is part of the answer set, (ii) ensuring satisfiabilty of the rule and (iii) preventing unfoundedness of the guessed head atom. To lift this idea to normal programs, one can rely on encoding (iv) orderings. Since every step of the procedure instantiates at most one body predicate at a time, we intuitively deploy body-decoupling, which keeps the grounding size polynomial when assuming bounded predicate arity. Notably, our results imply that body-decoupled grounding blends-in well with existing approaches, enabling us to interleave different grounding approaches.

2 Experimental Results

We implemented a software tool, called `newground`[1], realizing body-decoupled grounding via search as described above. The system `newground` is written in Python3 and uses, among others, the API of `clingo` 5.5 and its ability to efficiently parse logic programs via syntax trees. In our implementation, we opted for partial reducability, allowing users to select program parts that shall be reduced and those being (traditionally) grounded, thereby internally relying on `gringo`.

In order to evaluate `newground`, we design a series of benchmarks. Clearly, we cannot beat highly optimized grounders in all imaginable scenarios. Instead, we discuss potential use cases, where body-decoupled grounding is preferrable, since this approach can be incorporated into every grounder. We consider these (directed) graph *scenarios*: (S1) 3-coloring, (S2) reachable paths, (S3) cliques, (S4) non-partition-removal colorings [8] and (S5) stable marriages (ASP comp. 2014), and compared to `gringo` and `idlv` while measuring grounding size, grounding time as well as overall time.

From our experiments we crafted grounding profiles for each tool. Figure 1 depicts the grounding profile of S1 for `gringo` (left) and `newground` (right), showing grounding times and sizes, depending on the instance size (x-axis) and density (y-axis). Interestingly, for `newground` grounding times and sizes for a fixed instance size (column) of Figure 1 (right) are quite similar, which is in contrast to Figure 1 (left), suggesting that compared to `gringo`, `newground` is not that sensitive to instance density.

In terms of pure grounding time, our experiments show that `newground` in fact outperforms in four of five scenarios. Interestingly, while doing so, `newground` also massively reduces the grounding size (cf., Figure 2 (left), almost all dots above diagonal), while keeping instances solvable where `gringo` and `idlv` output groundings beyond 30GB. For the overall (solving) performance we refer to Figure 2

---

[1] The system (incl. supplemental material) is available at `https://github.com/viktorbesin/newground`.

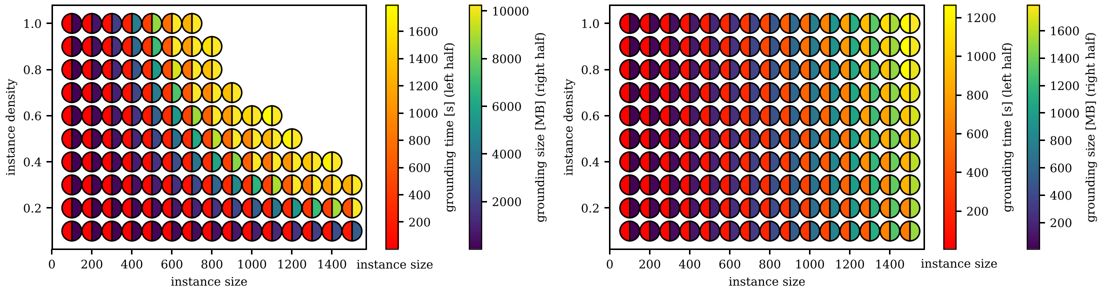


Figure 1: Grounding profile of S1 for `gringo` (left) and `newground` (right). The x-axis refers to the instance size; the y-axis indicates density. Circles mark instances grounded $< 1800s$; the left (right) half depicts grounding time (size), respectively. Mind the different color scales (e.g., 10000MB vs. 1600MB).

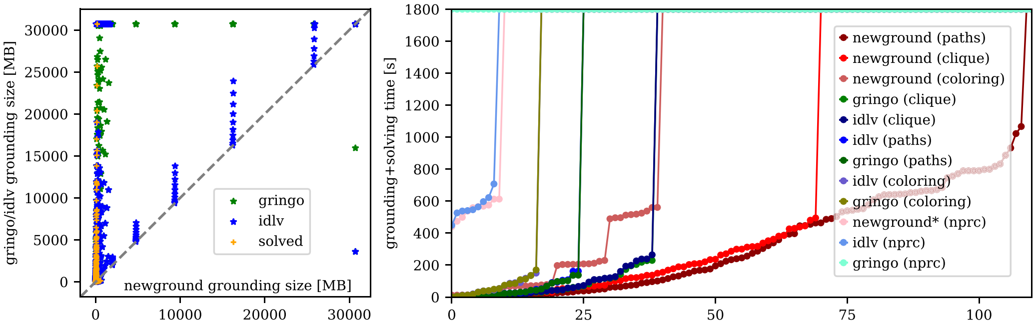


Figure 2: (Left): Scatter plot of grounding size over Scenarios S1–S4 of `newground` (x-axis) compared to both `gringo` (blue) and `idlv` (green) on the y-axis. Those instances that could be solved are highlighted in orange. (Right): Cactus plot of overall (grounding *and solving*) time over Scenarios S1-S4.

(right). While `newground` performs best, we still see a clear difference between solving and grounding performance, which reveals that only a small amount of those grounded instances can then actually be solved by `clingo` within the remaining time. More plots and evaluation can be found in [1].

3 Conclusion

This work introduces a grounding-approach based on a reduction suggesting the body-decoupling of grounding-intense ASP rules. The reduction translates tight (normal) non-ground rules into disjunctive ground rules, thereby being exponential only in the maximum predicate arity. While our evaluation shows that body-decoupled grounding applied on crucial (tight) program parts reduces grounding size compared to state-of-the-art exact grounders, we are currently working on evaluating and tuning of an implementation for normal programs [7].

References

[1] Viktor Besin, Markus Hecher & Stefan Woltran (2022): *Body-Decoupled Grounding via Solving: A Novel Approach on the ASP Bottleneck*. In: *IJCAI'22*, ijcai.org, pp. 2546–2552. Available at `https://doi.org/10.24963/ijcai.2022/353`.

[2] Nicole Bidoít & Christine Froidevaux (1991): *Negation by default and unstratifiable logic programs*. Theoretical Computer Science 78(1), pp. 85–112, doi:10.1016/0304-3975(51)90004-7.

[3] Bernardo Cuteri, Carmine Dodaro, Francesco Ricca & Peter Schüller (2020): *Overcoming the Grounding Bottleneck Due to Constraints in ASP Solving: Constraints Become Propagators*. In: *IJCAI*, ijcai.org, pp. 1688–1694.

[4] Thomas Eiter, Wolfgang Faber, Michael Fink & Stefan Woltran (2007): *Complexity results for answer set programming with bounded predicate arities and implications*. Annals of Mathematics and Artificial Intelligence 51(2-4), pp. 123–165.

[5] Wiktor Marek & Mirosław Truszczyński (1991): *Autoepistemic logic*. Journal of the ACM 38(3), pp. 588–619, doi:10.1145/116825.116836.

[6] Efthymia Tsamoura, Víctor Gutiérrez-Basulto & Angelika Kimmig (2020): *Beyond the Grounding Bottleneck: Datalog Techniques for Inference in Probabilistic Logic Programs*. In: *AAAI'20*, AAAI Press, pp. 10284–10291.

[7] Kaan Unalan (2022): *Body-Decoupled Grounding in Normal Answer Set Programs*. Bachelor's Thesis, TU Wien, Austria.

[8] Antonius Weinzierl, Richard Taupe & Gerhard Friedrich (2020): *Advancing Lazy-Grounding ASP Solving Techniques - Restarts, Phase Saving, Heuristics, and More*. Theory Pract. Log. Program. 20(5), pp. 609–624.

Tackling the DM Challenges with cDMN: A Tight Integration of DMN and Constraint Reasoning (Extended Abstract)

Simon Vandevelde[*], Bram Aerts, and Joost Vennekens
KU Leuven, De Nayer Campus,
Dept of Computer Science,
2560 Sint-Katelijne-Waver, Belgium
Leuven.AI
KU Leuven Institute for AI
B-3000, Leuven, Belgium
Flanders Make@KU Leuven
{s.vandevelde, joost.vennekens}@kuleuven.be

This extended abstract summarizes work published in Theory and Practice of Logic Programming [5].

1 Introduction

The Decision Model and Notation (DMN) [4] standard, maintained by the Object Management Group, allows to represent data and decision logic in an intuitive, table-based way. It is meant to be used directly by business experts, and as such, aims to be user-friendly and low in complexity. Decisions in DMN are modelled in *decision tables*, which contain the in-depth business logic of the model. As an example, Fig 1 shows a table to decide a patient's BMI Level. The meaning of such a decision table is straightforward: the value of the output variable (right, in light blue) is defined by the value of the input variable(s) (left, in green). Each row of the table corresponds to a decision rule. We say that a rule *fires* whenever the actual value of the input variables match the values listed in its cells. The way in which the inputs define the output depends on the *hit policy*, denoted in the top-left corner. In the example, the *U(nique)* hit policy denotes that the rules may not overlap.

BMI Level		
U	BMI	BMI Level
1	< 18.5	Underweight
2	[18.5..25]	Normal
3	(25..30]	Overweight
4	> 30	Obese

Figure 1: Example of decision table

---

[*] This research received funding from the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" programme.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 402–404, doi:10.4204/EPTCS.385.47

2 cDMN: Extending DMN with constraints

While DMN allows modellers to elegantly represent a deterministic decision process, it lacks the ability to specify constraints on the solution space. To address this, we present "Constraint Decision Model and Notation" (cDMN), an extension of DMN with constraints. It also allows for representations that are independent of domain size by supporting types, functions, relations and quantifications. Importantly, it does so in a way that retains the ease of use for domain experts.

To illustrate cDMN, we will now briefly go over each type of table, and illustrate them using the graph colouring example in Fig. 2.

Glossary In logical terms, the "variables" of standard DMN correspond to constants (i.e., 0-ary functions). cDMN extends these by adding n-ary functions and predicates. Modellers must declare their vocabulary by means of a glossary, which consists of at most five glossary tables: *Type*, *Function*, *Relation*, *Boolean* and *Constant*.

Logic in cDMN is *typed*: modellers must define types in the *Type* table together with a set of domain elements. For example, the types *Country* and *Colour* respectively represent the domain of countries and colours. In the *Function* table, symbols can be declared as functions of one or more types to another. All types that appear in the description are interpreted as arguments, and the remainder constitutes the name of the function itself. As such, *colour of Country* in our example corresponds to $colour\_of : Country \rightarrow Colour$. In a similar manner, we can declare n-ary predicates in the *Relation* table, such as *Country borders Country*. This predicate corresponds to the 2-ary predicate $borders : Country \times Country \rightarrow \mathbb{B}$. The last two glossary tables are used to introduce boolean symbols (i.e., propositions) and constants, and are not demonstrated in the example.

Decision tables and constraint tables Where DMN decision tables must precisely define a symbol, our constraint tables do not have this property. Instead, each row of the table expresses a logical implication: if the conditions on the inputs are satisfied, the condition on the output must hold as well. We introduce a new hit policy, denoted as "E*" (Every), because it expresses that every implication must be satisfied.

cDMN also extends the expressions that are allowed in the column headers. Columns of the form "*Type* called *var*" introduce a universally quantified variable *var* over the domain of *Type*. Each subsequent column may use this variable in the header or the cells. In this way, we introduce universal quantification in decision and constraint tables. To illustrate, the constraint table in the example introduces two variables, *c1* and *c2*, both of type *Country*. It can be read as: "For every country c1 and c2, it must hold that if the two countries border, they do not have the same colour". The table corresponds to the following FO formula:

$$\forall c1, c2[Country] : border(c1, c2) \Rightarrow colour\_of(c1) \neq colour\_of(c2).$$

Data tables Typically, problems can be split up into two parts: (1) the general logic of the problem, and (2) the specific problem instance to solve. In the map colouring example, the general logic consists of the constraint that two bordering countries cannot share a colour, whereas the instance of the problem is the specific map (e.g., Western Europe) to colour. cDMN extends DMN with *data tables*, which are used to represent the problem instances, separating them from the general logic. In terms of classical logic, a data table corresponds to the interpretation of a symbol.

Type		
Name	**Type**	**Values**
Country	String	BE, FR, LU, NL, DE, . . .
Colour	String	Red, Blue, Green, Yellow

Function	
Name	**Type**
colour of Country	Colour

Relation
Name
Country borders Country

Colour constraint

E*	Country called c1	Country called c2	c1 borders c2	Colour of c1
1	—	—	Yes	not(Colour of c2)

Data table: bordering countries

	Country called c1	Country called c2	c1 borders c2
1	BE	FR, LU, NL, DE	Yes
2	DE	FR, DK, LU, BE, NL	Yes
3	. . .	. . .	. . .

Goal
Get 3 models

Figure 2: Example of a cDMN model for the map colouring problem

Goal table A standard DMN model defines a deterministic decision procedure. In cDMN, this is no longer the case: cDMN models instead have a solution space. By including a *goal* table, modellers can state what the specification is to be used for: generating x solutions, generating the solution with an optimal value for a variable, or deriving the consequences.

cDMN models can be executed using the cDMN solver [1]. It translates the model to First Order Logic, and then runs the IDP-Z3 reasoning engine [3] to generate solutions.

3 Results

We evaluated cDMN by implementing DM Community challenges [2], which are challenges set forth by the decision management community to test the boundaries of DMN. Of the 24 challenges we considered, cDMN could model 22, whereas state-of-the-art DMN engines could model 12. We find that our models are more robust to domain changes due to the introduction of types, quantification and data tables. Moreover, cDMN implementations are typically more concise, and do not need to rely on unconventional workarounds (as is the case in some of the DMN implementations). We feel that these benefits make cDMN models more readable and maintainable compared to the other solutions due to its increased expressiveness.

References

[1] *cDMN documentation*. `http://www.cdmn.be`.

[2] *DMCommunity Challenge*. `https://dmcommunity.org/challenge/`.

[3] Pierre Carbonnelle, Simon Vandevelde, Joost Vennekens & Marc Denecker (2023): *Interactive Configurator with FO(.) and IDP-Z3*. arXiv:`2202.00343`.

[4] Object Management Group (2021-02, 2021): *Decision Model and Notation v1.3*.

[5] Simon Vandevelde, Bram Aerts & Joost Vennekens (2021): *Tackling the DM Challenges with cDMN: A Tight Integration of DMN and Constraint Reasoning*. Theory and Practice of Logic Programming, pp. 1–24, doi:`10.1017/S1471068421000491`.

FOLL-E: Teaching First Order Logic to Children (Extended Abstract)

Simon Vandevelde* and Joost Vennekens

KU Leuven, De Nayer Campus,
Dept of Computer Science,
2560 Sint-Katelijne-Waver, Belgium

Leuven.AI
KU Leuven Institute for AI
B-3000, Leuven, Belgium

Flanders Make@KU Leuven

`{s.vandevelde, joost.vennekens}@kuleuven.be`

This abstract summarizes work presented at the Thirteenth AAAI Symposium on Educational Advances in Artificial Intelligence [6].

1 Introduction

First-order logic (FO) is an important foundation of many domains, such as mathematics, philosophy, computer science and artificial intelligence. Currently, there is a growing interest in teaching computer science concepts and computational thinking to children, such as through Scratch [5], Blockly [3] or CS Unplugged [1]. One topic which appears to be curiously understudied, however, is that of teaching of FO. Indeed, FO is not only important in CS and AI, but learning FO can also help children in a variety of other disciplines by sharpening their reasoning skills. While Kowalski and colleagues already saw the potential of teaching Prolog in 1980 [4], not many tools or applications have been published for teaching any form of logic since then. This is due to a number of issues that may make teaching FO difficult: (1) logic has a steep learning curve, (2) it does not "do" anything, and (3) it is not fun due to the lack of graphics, animations and sounds. In our work, we overcome these objections by designing an appropriate learning environment. The focus of this environment is twofold: a child-friendly representation for FO, combined with a fun and engaging task to solve.

2 Goals and Approach

Representation of FO: The traditional way of writing logic is hard for children to understand. Our goal is therefore to introduce an alternative notation that meets four goals: (G1) *avoid discouraging children with finicky syntax*, (G2) *make the structure of formulas clear*, (G3) *encourage experimentation*, and (G4) *encourage collaboration*.

To meet these goals, we designed a block-based representation for FO, inspired by languages such as Scratch [5] and Blockly [3]. It was designed according to a *pegs-and-slots* approach, in which blocks physically fit together if the connection is syntactically correct and the formulas are correctly typed. This representation, of which an example is shown in Fig. 1a, effectively eliminates syntax errors (G1), and provides a clear overview of the structure of the formulas (G2).

---

*This research received funding from the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 405–407, doi:10.4204/EPTCS.385.48

(a) Blocks expressing "$\forall c : Color(c) = red$"

(b) Laser-cut blocks (Engraving in Dutch)

Figure 1: Block-based notation for FO

Instead of implementing these blocks in a GUI, we laser cut physical blocks out of 3mm plywood, engraved with their intended meaning and, when possible, a relevant icon (Fig. 1b). Blocks representing a colour were also painted in that colour. In this way, the bocks are low on text, with much visual stimulation. This results in a tactile, playful and almost puzzle-like experience, which stimulates experimentation (G3) and encourages experimentation between children (G4).

Task to solve: Our system asks children to construct formulas in order to solve certain tasks. In designing these tasks, we focus on *teaching the* $\models$*-relation* (R1). In addition, we also aim to: (R2) *provide clear and immediate feedback*, (R3) *offer a gradual increase in difficulty*, and (R4) *make learning fun and engaging*.

As an application domain, we have chosen the design of simple robots, as shown on the monitor in Fig. 2a. A robot consists of six body parts, called *components*, which can be individually coloured in one of three colours. Additionally, a robot may also wear a hard hat. In the application, children are shown three "good" robots and three "bad" robots, and must devise a rule that distinguishes between them. In other words, they must create a formula for which the three good robots are models and the three bad ones are not (R1).

By puzzling together blocks, the children can express sentences of increasing difficulty, such as "The left leg is blue", "If the robot wears a helmet, it has a green body" and "Every component that is a limb must be coloured green" (R3). A Raspberry Pi mini-computer continually scans the playing area, and, as soon as it detects a complete sentence, will run the IDP-Z3 reasoning system [2] to verify which robots satisfy the formula, which is shown visually on the screen. FOLL-E is "always live": in this way, it gives immediate feedback (R2). Moreover, the feedback is entirely visual, which makes the application more intuitive, enticing and engaging (R4).

(AI) Vlaanderen" programme.

(a) (b)

Figure 2: Photos taken at a FOLL-E workshop

3 Results

As a preliminary evaluation, we tested FOLL-E in workshops aimed at 8-13 year old children. During the workshop, the children are given a 5 minute introduction on FOLL-E, after which they are free to play through the levels in pairs. All students were able to complete all ten levels, but the time required varied with age: children near the upper age limit (13) finished about 15 minutes ahead of the others.

The children reported that in general (a) they enjoyed their time with FOLL-E, (b) found the puzzle-like nature inviting and (c) liked the assignment. One of the older children remarked "It's satisfying to puzzle the pieces together and see the result, especially if it's correct."

In future work, we will be teaming up with cognitive psychology researchers to measure the effectiveness of the tool. Furthermore, we will be looking at what changes can be made to make the tool more effective, such as introducing more levels, experimenting with applications besides robot design, and adding automated error explanation.

References

[1] Tim Bell & Jan Vahrenhold (2018): *CS Unplugged—How Is It Used, and Does It Work?* In Hans-Joachim Böckenhauer, Dennis Komm & Walter Unger, editors: *Adventures Between Lower Bounds and Higher Altitudes: Essays Dedicated to Juraj Hromkovič on the Occasion of His 60th Birthday*, Springer International Publishing, Cham, pp. 497–521, doi:10.1007/978-3-319-98355-4_29.

[2] Pierre Carbonnelle, Simon Vandevelde, Joost Vennekens & Marc Denecker (2023): *Interactive Configurator with FO(.) and IDP-Z3*. arXiv:2202.00343.

[3] Neil Fraser (2015): *Ten Things We've Learned from Blockly*. In: *2015 IEEE Blocks and beyond Workshop (Blocks and Beyond)*, pp. 49–50, doi:10.1109/BLOCKS.2015.7369000.

[4] Robert A Kowalski (1982): *Logic as a Computer Language for Children*. In: *ECAI*, pp. 2–10.

[5] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman & Yasmin Kafai (2009): *Scratch: Programming for All*. *Communications of The Acm* 52(11), pp. 60–67, doi:10.1145/1592761.1592779.

[6] Simon Vandevelde & Joost Vennekens (2023): *FOLL-E: Teaching First Order Logic to Children*. In: *Proceedings of the Thirteenth AAAI Symposium on Educational Advances in Artificial Intelligence*, Washington, USA.

Learning to Break Symmetries for Efficient Optimization in Answer Set Programming (Extended Abstract)

Alice Tarzariol
Martin Gebser*
Konstantin Schekotihin
University of Klagenfurt
Klagenfurt, Austria
{firstname.lastname}@aau.at

Mark Law
ILASP Limited
Grantham, UK
mark@ilasp.com

The ability to efficiently solve hard combinatorial optimization problems is a key prerequisite to various applications of declarative programming paradigms. Symmetries in solution candidates pose a significant challenge to modern optimization algorithms since the enumeration of such candidates might substantially reduce their optimization performance. The full version of this paper proposes a novel approach using Inductive Logic Programming (ILP) to lift symmetry-breaking constraints for optimization problems modeled in Answer Set Programming (ASP). Given an ASP encoding with optimization statements and a set of representative instances, our method augments ground ASP programs with auxiliary normal rules enabling the identification of symmetries by existing tools, like SBASS. Then, the obtained symmetries are lifted to first-order constraints with ILP. We show the correctness of our method and evaluate it on optimization problems from the domain of automated configuration. Our experiments show significant improvements of optimization performance due to the learned first-order constraints. *The full version of this paper was presented at AAAI 2023.*

1 Introduction

Combinatorial optimization problems appear in various practical applications, including transportation, manufacturing, healthcare, or power generation and distribution. An efficient and elegant approach to tackle these problems is to model them in a declarative programming paradigm such as Answer Set Programming (ASP) [6]. However, finding optimal solutions to the vast majority of real-world problems is challenging since their search spaces include many symmetric solution candidates, i.e., an isomorphic candidate can be obtained by permuting elements of a known solution. To be effective, the problem encoding must thus include Symmetry-Breaking Constraints (SBCs) that prune the search space of solution candidates by removing symmetric ones. Hence, several techniques to automatically identify and discard redundant solutions have been designed in the recent decades [7, 8, 13], based on embedding symmetry breaking in search algorithms or adding SBCs to a given problem encoding or instance, respectively.

In the context of ASP, the system SBASS [5] was designed to provide an instance-specific approach to symmetry breaking. To this end, SBASS identifies symmetries of a ground ASP program by *(i)* representing the input program as a colored graph, *(ii)* finding symmetric vertex permutations in the graph using SAUCY [1, 3], and *(iii)* constructing ground SBCs from the permutations. In previous work [9], we introduced an approach that learns first-order constraints by lifting ground SBCs of SBASS by means of Inductive Logic Programming (ILP) [2]. Our method applies SBASS to a set of representative problem instances and then uses the obtained permutations to define an ILP task. The subsequent learning

---

[*] also affiliated with Graz University of Technology, Austria

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 408–410, doi:10.4204/EPTCS.385.49

phase determines first-order constraints that remove symmetric solutions (classified as negative examples) while preserving the representative solutions (positive examples).

The main shortcoming of previously introduced methods [10, 12] to learn constraints from SBCs of ASP programs is their inability to deal with optimization statements. That is, if a suboptimal solution happens to be lexicographically smallest and is taken as representative to discard symmetric solutions, the learned constraints might eliminate (all) optimal solutions from the search space.

Example. Let us consider the (ground) ASP program P_1:

```
1 {a;b;c} 1.
:~ a. [3]   :~ b. [2]   :~ c. [3]
```

The choice rule in the first line gives rise to three answer sets: $AS(P_1) = \{\{\mathtt{a}\}, \{\mathtt{b}\}, \{\mathtt{c}\}\}$. Three weak constraints in the second line attribute the weight 3 to answer sets including `a` or `c`, and 2 to the answer set comprising `b`, so that the optimal answer set is $\{\mathtt{b}\}$. When SBASS is run to analyze P_1, i.e., the choice rule in the first line, it classifies the answer sets in $AS(P_1)$ as symmetric. Thus, applying the previous approach [12] yields the lexicographically smallest answer set $\{\mathtt{a}\}$ as representative solution to preserve, while $\{\mathtt{b}\}$ and $\{\mathtt{c}\}$ are considered redundant. As a consequence, the framework will learn some constraint(s), e.g., `:- not a.`, eliminating the optimal answer set $\{\mathtt{b}\}$ with the smallest weight 2.

2 Contributions

In our AAAI 2023 paper [11], we extend the learning approaches of [10, 12] to optimization problems and make the following contributions:

- We propose a method that augments a ground ASP program P with auxiliary rules, O_P, that do not change the solutions but tighten the applicable symmetries for optimization problems. We show that SBASS on $P \cup O_P$ partitions solutions of P in a way that respects their quality determined by optimization statements. That is, each cell of the partition of answer sets computed by SBASS is guaranteed to consist of symmetric solutions sharing the same optimization value.
- We enhance the previous learning approaches to support the learning of effective first-order constraints for optimization problems.
- Lastly, we test the proposed framework on combinatorial configuration problems incorporating optimization statements. Our results show that the learning performance is comparable between ILP tasks defined with or without taking optimization statements into account. Moreover, the learned first-order constraints prune the symmetries of optimization problems and thus speed up the computation of (representative) optimal solutions.

3 Conclusion

Our AAAI 2023 paper extends the previous framework [9, 10, 12] to support the learning of effective first-order constraints for optimization problems. The introduced auxiliary rules are crucial for SBASS to identify symmetries applicable to ASP programs with optimization statements. We have shown the correctness of this approach and devised suitable methods to generate ILP tasks for optimization problems. Experiments with the new framework on optimization versions of the combinatorial problems addressed in [10, 12] as well as on the fastfood problem [4] show that the learned constraints can help to significantly speed up the computation of (representative) optimal solutions. The implementation, benchmarks as well as settings for reproducing our experiments can be found at `https://github.com/prosysscience/Symmetry_Breaking_with_ILP/tree/optimization`.

Acknowledgment This work was funded by KWF project 28472, cms electronics GmbH, FunderMax GmbH, Hirsch Armbänder GmbH, incubed IT GmbH, Infineon Technologies Austria AG, Isovolta AG, Kostwein Holding GmbH, and Privatstiftung Kärntner Sparkasse.

References

[1] P. Codenotti, H. Katebi, K. Sakallah & I. Markov (2013): *Conflict Analysis and Branching Heuristics in the Search for Graph Automorphisms*. In: *IEEE 25th International Conference on Tools with Artificial Intelligence*, IEEE Computer Society, pp. 907–914, doi:`10.1109/ICTAI.2013.139`.

[2] A. Cropper, S. Dumančić & S. Muggleton (2020): *Turning 30: New ideas in Inductive Logic Programming*. In: *International Joint Conference on Artificial Intelligence*, ijcai.org, pp. 4833–4839, doi:`10.24963/ijcai.2020/673`.

[3] P. Darga, H. Katebi, M. Liffiton, I. Markov & K. Sakallah (2004): *Saucy*. `http://vlsicad.eecs.umich.edu/BK/SAUCY/`. Accessed: 2021-05-21.

[4] M. Denecker, J. Vennekens, S. Bond, M. Gebser & M. Truszczyński (2009): *The Second Answer Set Programming Competition*. In E. Erdem, F. Lin & T. Schaub, editors: *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, *Lecture Notes in Artificial Intelligence* 5753, Springer, pp. 637–654, doi:`10.1007/978-3-642-04238-6_75`.

[5] C. Drescher, O. Tifrea & T. Walsh (2011): *Symmetry-breaking answer set solving*. *AI Communications* 24(2), pp. 177–194, doi:`10.3233/AIC-2011-0495`.

[6] V. Lifschitz (2019): *Answer Set Programming*. Springer, doi:`10.1007/978-3-030-24658-7`.

[7] F. Margot (2010): *Symmetry in integer linear programming*. In: *50 Years of Integer Programming 1958-2008*, Springer, pp. 647–686, doi:`10.1007/978-3-540-68279-0_17`.

[8] K. Sakallah (2009): *Symmetry and Satisfiability*. In A. Biere, M. Heule, H. van Maaren & T. Walsh, editors: *Handbook of Satisfiability*, chapter 10, *Frontiers in Artificial Intelligence and Applications* 185, IOS Press, pp. 289–338, doi:`10.3233/978-1-58603-929-5-289`.

[9] A. Tarzariol, M. Gebser & K. Schekotihin (2021): *Lifting Symmetry Breaking Constraints with Inductive Logic Programming*. In: *International Joint Conference on Artificial Intelligence*, ijcai.org, pp. 2062–2068, doi:`10.24963/ijcai.2021/284`.

[10] A. Tarzariol, M. Gebser & K. Schekotihin (2022): *Lifting Symmetry Breaking Constraints with Inductive Logic Programming*. *Machine Learning* 111(4), pp. 1303 – 1326, doi:`10.1007/s10994-022-06146-3`.

[11] A. Tarzariol, M. Gebser, K. Schekotihin & M. Law (2023): *Learning to Break Symmetries for Efficient Optimization in Answer Set Programming*. In: *Proceedings of the Thirty-seventh National Conference on Artificial Intelligence (AAAI'23)*, AAAI Press. To appear.

[12] A. Tarzariol, K. Schekotihin, M. Gebser & M. Law (2022): *Efficient Lifting of Symmetry Breaking Constraints for Complex Combinatorial Problems*. *Theory and Practice of Logic Programming*, doi:`10.1017/S1471068422000151`.

[13] T. Walsh (2012): *Symmetry breaking constraints: Recent results*. In: *26th AAAI Conference on Artificial Intelligence*, AAAI Press, pp. 2192–2198, doi:`10.1609/aaai.v26i1.8437`.

Penalization Framework For Autonomous Agents Using Answer Set Programming

Vineel S. K. Tummala
Computer Science and Software Engineering
Miami University
Oxford, USA
`tummalvs@miamioh.edu`

This paper presents a framework for enforcing penalties on intelligent agents that do not comply with authorization or obligation policies in a changing environment. A framework is proposed to represent and reason about penalties in plans, and an algorithm is proposed to penalize an agent's actions based on their level of compliance with respect to authorization and obligation policies. Being aware of penalties an agent can choose a plan with a minimal total penalty, unless there is an emergency goal like saving a human's life. The paper concludes that this framework can reprimand insubordinate agents.

1 Introduction and Motivation

A framework will be presented that can enforce penalties on intelligent agents in a changing environment if autonomous agents don't obey authorization or obligation policies. Policies for an agent A acting in a changing domain/environment T are encoded as a set of conditions that describes whether an agent's actions are permitted or not, and whether the agent is obligated to perform an action or not perform it. In some cases, an autonomous agent can choose to perform an unauthorized action. In this case, the agent is forced to pay a penalty, where it can be reprimanded for its actions or lose its job for insubordination [1]. To make this possible, an architecture will be created to represent and reason about penalties in plans.

As autonomous agents continue to develop, it is necessary that proper policy representation and comprehension exist. It is essential to monitor the issues within the environment to ensure that agents act appropriately. Autonomous agents are intended to perform certain actions without any instructions or interference from a controller (pilot or driver) of the agent. The activities of an autonomous agent are often modeled after human behavior so that the agent can perform tasks that humans would. In order to do this, human behavior needs to be accurately represented. This can be successfully achieved by a logical approach, using declarative programming languages. Declarative programming languages are able to define human behavior correctly and in an understandable way. One such declarative programming language is Answer Set Programming [2],

2 Background and overview of the existing literature

In this section, background work related to the penalization framework for autonomous agents will be described. I assume that the reader is familiar with Answer Set Programming (ASP) [8, 7], a declarative programming language with roots in Logic Programming and Nonmonotonic reasoning [4], and the ASP solver CLINGO[1]. Full details about CLINGO can be found in the papers by Calimeri et al. [3] and Gebser

---

[1] `https://potassco.org/Clingo/`

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 411–415, doi:10.4204/EPTCS.385.50

et al. [5]. I also assume familiarity of the reader with autonomous intelligent agents.

2.1 Transition Systems

In order for intelligent agents to be functional and autonomous, they require a proper description of the changing environment in which they act. Transition systems are a tool that is used to describe these domains. Transition systems are fundamentally directed graphs that represent a changing domain, in which the nodes represent the state of the domain, and actions that might occur in the domain are represented by arcs [1]. As a result, the domain is altered from one state to another. Transition diagrams can become huge if many fluents are being tracked, thus they are not used in practice. They can be represented in a more concise way by an action language [2] which is a high-level declarative language.

2.2 Policies

The proposed research will be focusing on agents which are aware of policies. Policies are sets of rules to follow in a given domain or environment and are also described using declarative programming languages. Given a policy, P, with respect to a given environment T, all actions performed in the said environment can be judged based on whether or not they follow the rules defined by the policy. In the case of an intelligent agent, it is important for it to be aware of all the policies in advance in order to comply with all the necessities when coming up with plans for action. The following section will define compliance to policy, look into both authorization and obligation policies.

2.2.1 Authorization

Authorization policies are policies in which an agent is either permitted (authorized) or not permitted (not authorized) to perform an action. Gelfond and Lobo [9] define Authorization Policy Language (*APL*) to describe authorization policies in a dynamic system $\Sigma = \langle A, T \rangle$, with an agent A and a transition diagram/environment T. This language is implemented through Answer Set Programming (ASP). The signature of *APL* includes a predicate for authorization policies *permitted(e)* where e is an elementary action of A. [9].

2.2.2 Obligation

An obligation policy is one in which an agent is obligated to carry out an action or not to carry it out. *AOPL* is a policy language to describe systems with an agent A and a transition diagram T over signature Σ, similar to *APL* [9]. Known as *Authorization and Obligation Policy Language*, Gelfond and Lobo [9] designed this language, to extend *APL* with obligation policies by adding a new symbol *obl(h)* where h is an elementary action of A or negation of any such action [9]. This language is implemented through ASP.

2.3 Compliance

Compliance denotes whether an action or set of actions should be taken or not in accordance with a given policy P. A set of actions is defined as strongly compliant, weakly compliant, or non-compliant. Strongly compliant means that no rules were broken and it allows/supports their execution. If there is

---

[2] See `https://www.diva-portal.org/smash/get/diva2:1716299/FULLTEXT01.pdf`

no sufficient information then it is called weakly compliant, and if the actions have broken one or more policies then it is called as non-compliant.

2.3.1 Authorization Compliance

Compliance in relation to authorization policies, as defined by *APL(Σ)*, can be divided into strongly compliant, weakly compliant, or not compliant [9].

2.3.2 Obligation Compliance

For obligation policies defined through *AOPL*(Σ), events are classified into *compliant* or *non-compliant* [9]. Unlike authorization policies, there are no weakly compliant events with respect to obligations.

2.4 Planning

A plan is a sequence of actions taken by an agent in order to achieve a given goal [6]. *Answer Set Planning* [10] refers to the use of Answer Set Programming to solve different types of planning problems by adding a planning module to an ASP description of a changing environment. Answer sets of the resulting program correspond to possible plans.

3 Goal of the research

The goal of the proposed research is to create a penalization framework for autonomous agents using ASP where autonomous agents are penalized if they do any job/task that they are not intended to do according to the policy.

3.1 Constructing the Framework

In this section, we present our ideas for constructing a penalization framework for autonomous agents.

3.1.1 Encoding the penalties

In order for these penalties to be imposed on the agents we will create a statement to penalize the agents. we will choose a specific scale to impose penalty points for agents; as of now the scale is 1-3, in the actual framework we will explore different possibilities.

We keep track of the values as low, medium, and high to help the agent calculate the value of the penalty while choosing the plans.

3.1.2 Calculating the total penalty

We calculate the total penalty, in the end, using the aggregate sum in CLINGO[3] (solver for ASP). Aggregates are functions that combine the values of a set of terms to produce a single value. Using this function we add up all the penalties given to an agent in an environment and we can evaluate an agent based on these penalties.

---

[3] `https://potassco.org/Clingo/run/`

4 Current status of the research

Collected all background and related work, and created two domains to test the penalization framework.

4.1 Collecting policies and scenarios

When we speak about penalizing autonomous agents, we need to create domains and scenarios where we evaluate autonomous agents for their actions and penalize them. Also, to achieve this we need to specify authorization and obligation policies for the domain beforehand. Conflicts may arise when Authorization and obligation policies interact or come across each other. Therefore we need to specify authorization and obligation policies precisely. First definitions and rules of both the authorization and obligation policies developed by Gelfond and Lobo [9] must be understood. After fully understanding the rules that exist, different scenarios can be tested. Scenarios will be tested to see what an agent selects if it has two non-compliant plans, and what plan it chooses. Some of the scenarios are:

4.1.1 Domain 1: Drone Delivery

Goal: Its goal is to deliver a package to a customer's doorstep, located in a suburban area with many houses.

Policy Rules: Consider an autonomous agent in this scenario as a delivery drone with the following rules to ensure safe and efficient deliveries:

- Always fly at a safe height to avoid obstacles and people on the ground.
- only land on designated landing pads or open spaces to avoid collisions and ensure safe deliveries.

4.1.2 Domain 2: Self-Driving Car

The autonomous agent in this scenario is a self-driving car, programmed with a set of rules to ensure safe and efficient transportation.

Goal: Its goal is to transport a passenger from their current location to a specified destination, which involves driving through a busy city street with heavy traffic.

Policy Rules: Consider an autonomous agent in this scenario as a self-driving car with the following rules to ensure safe and efficient deliveries:

- Always obey traffic laws and signals, including speed limits, stop signs, and traffic lights.
- Prioritize the safety of passengers and other drivers on the road.

5 Open issues and expected achievements

Open Issues

- We can't create a penalty for each and every possible action, so should we set a default value if the action is not encoded?
- Setting a maximum penalty limit after which the agent would be automatically disabled from its operations for not following the compliance. This would ensure the agents that they do not perform any operations that are not compliant.

The expected achievement is to create a framework where autonomous agent chooses plans with fewer penalties and also according to the strength of the goal.

References

[1] Justin Blount (2013): *An architecture for intentional agents.* Ph.D. thesis, doi:`10.4204/EPTCS.345.23`.

[2] Gerhard Brewka, Thomas Eiter & Miroslaw Truszczynski (2011): *Answer set programming at a glance.* *Commun. ACM* 54(12), pp. 92–103, doi:`10.1145/2043174.2043195`.

[3] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca & Torsten Schaub (2020): *ASP-Core-2 input language format.* *Theory and Practice of Logic Programming* 20(2), pp. 294–309, doi:`10.1017/S1471068419000450`.

[4] Thomas Eiter, Giovambattista Ianni & Thomas Krennwallner (2009): *Answer Set Programming: A Primer.* In Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutierrez, Siegfried Handschuh, Marie-Christine Rousset & Renate A. Schmidt, editors: *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30 - September 4, 2009, Tutorial Lectures*, *Lecture Notes in Computer Science* 5689, Springer, pp. 40–110, doi:`10.1007/978-3-642-03754-2_2`.

[5] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub & Marius Schneider (2011): *Potassco: The Potsdam Answer Set Solving Collection.* *AI Commun.* 24(2), pp. 107–124, doi:`10.3233/AIC-2011-0491`.

[6] Michael Gelfond & Yulia Kahl (2014): *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach.* Cambridge University Press, doi:`10.1017/CBO9781139342124`.

[7] Michael Gelfond & Vladimir Lifschitz (1988): *The Stable Model Semantics for Logic Programming.* In Robert A. Kowalski & Kenneth A. Bowen, editors: *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*, MIT Press, pp. 1070–1080, doi:`10.2307/2275201`.

[8] Michael Gelfond & Vladimir Lifschitz (1990): *Logic Programs with Classical Negation.* In David H. D. Warren & Péter Szeredi, editors: *Logic Programming, Proceedings of the Seventh International Conference, Jerusalem, Israel, June 18-20, 1990*, MIT Press, pp. 579–597, doi:`10.1007/BF03037169`.

[9] Michael Gelfond & Jorge Lobo (2008): *Authorization and Obligation Policies in Dynamic Systems.* In Maria Garcia de la Banda & Enrico Pontelli, editors: *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, *Lecture Notes in Computer Science* 5366, Springer, pp. 22–36, doi:`10.1007/978-3-540-89982-2_7`.

[10] Vladimir Lifschitz (1999): *Answer Set Planning (Abstract).* In Michael Gelfond, Nicola Leone & Gerald Pfeifer, editors: *Logic Programming and Nonmonotonic Reasoning, 5th International Conference, LPNMR'99, El Paso, Texas, USA, December 2-4, 1999, Proceedings*, *Lecture Notes in Computer Science* 1730, Springer, pp. 373–374, doi:`10.1007/3-540-46767-X_28`.

Beyond Traditional Neural Networks: Toward adding Reasoning and Learning Capabilities through Computational Logic Techniques

Andrea Rafanelli
Department of Computer Science
University of Pisa, Pisa, Italy
Department of Information Engineering, Computer Science and Mathematics
University of L'Aquila, Italy
`andrea.rafanelli@phd.unipi.it`

Deep Learning (DL) models have become popular for solving complex problems, but they have limitations such as the need for high-quality training data, lack of transparency, and robustness issues. Neuro-Symbolic AI has emerged as a promising approach combining the strengths of neural networks and symbolic reasoning. Symbolic Knowledge Injection (SKI) techniques are a popular method to incorporate symbolic knowledge into sub-symbolic systems. This work proposes solutions to improve the knowledge injection process and integrate elements of ML and logic into multi-agent systems (MAS).

1 Introduction and Problem Description

Deep Learning (DL) models have gained popularity for addressing complex problems in various domains. However, they face challenges. Insufficient or biased training data can result in poor model performance and limited generalization. DL models heavily rely on statistical models which can restrict their ability to reason about concepts not explicitly represented in the data. This limitation hampers their capacity to understand the world and draw accurate conclusions from available information. To address these issues, Neuro-Symbolic AI has emerged, which seeks to combine the strengths of neural networks and symbolic reasoning to overcome the limitations of purely statistical models. There are various techniques for integrating neural and symbolic systems (for a better understanding c.f., e.g., [6, 12, 21]). This work focuses mainly on methods that aim to combine these two systems by injecting symbolic knowledge into neural networks, known as Symbolic Knowledge Injection (SKI) techniques. Additionally, we propose the creation of collaborative networks that combine the strengths of logic-based agents and neural agents within a Neuro-Symbolic architecture, creating a more comprehensive and efficient system. This approach offers a unique synergy, paving the way for more advanced and intelligent systems.

2 Background and Existing Literature

In this section, we will provide a quick overview of the concepts and methods considered in our work.

Symbolic Knowledge Injection SKI is a strategy used to improve the performance of sub-symbolic predictors, such as neural networks, by integrating useful symbolic knowledge into them. It has several benefits, such as alleviating the problem of insufficient training data, reducing the required time

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 416–422, doi:10.4204/EPTCS.385.51

and computational resources for learning, increasing the predictive accuracy of sub-symbolic predictors, and providing human-interpretable frameworks. There are several ways the knowledge can be injected into predictors, mainly: *(i) loss constraining* [3, 13, 28], i.e., the inclusion of penalty terms in the loss function. Penalty terms are indicative of the limitations imposed by prior knowledge; *(ii) structure constraining* [4, 19, 29], i.e., the alteration of the structure of the sub-symbolic model to reflect symbolic knowledge expressed as constraints, e.g. by changing the number and size of hidden layers or by setting the connections between neurons; *(iii) knowledge embedding* [5, 7, 27], i.e., incorporating additional domain-specific knowledge by generating numeric data from the symbolic ones.

Abductive Logic Programming ALP is a programming paradigm [16] that combines logic programming with abductive reasoning [18] to generate hypotheses (or abducibles) that explain certain observations. Abduction is formally described [20] as a framework where, given a knowledge base KB and observations O, one seeks to find a hypothesis H, s.t. $KB \cup H \models O$.

ALP is a version of traditional logic programming in which the reasoner generates abductive hypotheses by assuming the truth of abducibles (a set of open predicates) given some integrity constraints. Integrity constraints are logical formulas of the form :-*Body*., where *Body* is a conjunction of logical literals. Intuitively, an integrity constraint specifies a set of conditions that must not hold for any valid solution to the problem. According to [17], an abductive framework is known as a triple $< P, A, I_c >$, where P is a collection of clauses, A is a set of predicate symbols, called *abducibles*, and I_c is a set of closed formulae. Clauses in P are of the form $H \leftarrow L_1, L_2, .., L_k,\ \ k \geq 0$, where H is an atom with a predicate symbol not present in A and L_i is a single literal. Therefore, P is a general logic program with the limitation that predicates present in A do not have definitions in P. Here, the idea is to link to an abductive framework a collection of models–commonly called generalized stable models, or GSMs [14]– that are assumed to reflect the framework itself and may be used to characterise abduction within this framework. Hence, any observation has an abductive explanations if it is consistent with at least one of these GSM models. Let $< P, A, I_c >$ represent an abductive framework, whereas O represents an observation. Then O is explained abductively given a set of hypotheses Δ iff exists a GSM, $M(\Delta)$, s.t. $M(\Delta) \models O$. In general, the main aim of abduction, given an observation O, is to find a set of *abductive explanation* $\Delta \subseteq A$, s.t.: *(i)* $M \models O$., and *(ii)* $M \models I_c$.

DALI DALI [8, 9] is a programming language developed to construct logical agents. It is an extension of the Prolog language[1], and any Prolog program can be considered a DALI agent. The agent orientation is provided through events as first-class objects in the language, as well as two types of rules: *reactive rules* and *proactive rules*. Reactive rules permit agents to interact with and respond to their surroundings. These rules enable the agents to react to external events. *External events* are triggered by the environment of the agent and are denoted by the postfix **E**. An agent may choose to react to an external event by employing a reactive rule containing the event in its head. Proactive rules, on the other hand, are used by DALI agents to take initiative and start activities when they deem it suitable. Using *internal events*, that are denoted by the postfix **I**, agents can act independently of their environment and other agents. DALI agents use *actions* to influence their surroundings, potentially in response to external or internal events. Actions may have pre-conditions, which are specified in the action rule. Otherwise, they are just an action atom denoted with the postfix **A**. Agents can communicate with and affect their environment through actions.

---

[1] `https://en.wikipedia.org/wiki/Prolog`

3 Goal of the Research

This work aims to address challenges in the field of DL models by exploring the integration of neural models and computational logic. It will focus on two main tools: SKI and MAS. The main goals are: *(i)* **RG 1**: use abduction as a way of integrating targeted knowledge into the injection process. *(ii)* **RG 2**: develop metrics to assess the goodness-of-fit of injection mechanisms from several perspectives, both qualitative and efficiency-related. *(iii)* **RG3**: formalise an integrated MAS in which one or more agents have purely perceptual tasks and other purely logical tasks.

RG 1 aims to use abduction to improve the SKI process. In supervised learning, the observation comprises input-output pairs encoded as a model $f(\mathscr{X}) \rightarrow \mathscr{Y}$. When using SKI, we want to modify $f(\cdot)$ to align it with symbolic knowledge. However, selecting the right rules for each output becomes challenging with increasing rule count, causing possible scalability issues. One solution is to leverage abduction, which helps identify plausible rules for a given output. Here, abduction can be seen as a rule $Y : -A.$, where Y is the observation (ground-truth) and A are abducibles generated by the reasoner. The injection procedure can then be modified to incorporate abducibles, making the predictions of $f(\cdot)$ consistent with the provided abducibles.

RG 2 aims to investigate different metrics to gain a deeper understanding of the potential benefits and limitations of SKI from multiple perspectives. In the field of SKI, the effectiveness of the injection procedure is often measured by comparing the performance of the SKI predictor with its counterpart (non-injected predictor), using metrics such as accuracy, F1 score or MSE. However, these metrics may not fully capture all aspects of knowledge injection, such as whether such injection resulted in a more sustainable predictor in terms of allocated resources or a more robust predictor, and so on. Therefore, **RG 2** seeks to study the injection process through a broader lens, including factors such as the sustainability and robustness of the injection procedure.

Finally, the aim of **RG 3** is to create a MAS, in which ML and logic-based elements can collaborate and interact with each other. In practice, given a neuro-symbolic architecture, the main objective is to allocate the different modules of the architecture to different collaborating agents. The integration of ML and logic-based elements into MAS could potentially lead to more efficient and effective decision-making processes.

Results We provided a rough idea of **RG 1** in a conference position paper [24], in which we proposed the incorporation of abductive reasoning through two possible frameworks. The idea is applied to image segmentation to address issues of data scarcity and low robustness. In this proposal, we suggest the use of either ABL [11] or a combination of ABL and knowledge injection. The article aims to indicate possible uses of abduction within the realm of Neuro-Symbolic integration.

We presented a preliminary idea of **RG 2** in [2], where we provide the first - to the best of our knowledge - set of Quality-of-Service (QoS) metrics for SKI, with a focus on quantifying robustness and efficiency advantages owing to injection. In this paper, we offer an initial formulation for the following metrics: *(i) memory footprint efficiency*, i.e., the gain in model complexity; *(ii) energy efficiency*, i.e., the gain in total energy required to train and deploy a sub-symbolic model; *(iii) latency efficiency*, i.e., improvements in terms of the time required for inference; *(iv) data efficiency*, i.e., the improvement in terms of the amount of data required to optimise a sub-symbolic model; *(v) robustness*, i.e., the capability of the injection mechanism to adapt to variations of input data and knowledge; *(vi) comprehensibility*, i.e., the capability of the injection mechanism to produce more intelligible models. In addition, this work discusses the potential impact of these metrics within the field of MAS, considering that a probable inefficiency of the sub-symbolic models incorporated in the agents affects their functioning in multiple ways (e.g. energy inefficiency, data inefficiency, computational inefficiency, and so on).

In a subsequent paper, [1], we extended and enhanced **RG 2** by providing a more rigorous mathematical definition and implementation of the following metrics: memory footprint, energy efficiency, latency efficiency, and data efficiency. The software tools necessary for the practical use of these metrics were also provided, and the metrics were evaluated on three different datasets using three different injection models[2]. The results provide valuable insights into the performance of different injection predictors on various datasets, highlighting the importance of adopting specific metrics when evaluating them. The paper emphasises that the use of these metrics in the context of MAS could be a crucial tool for comparing different predictors and selecting the most suitable one for a given task.

In [22], an initial approach to symbolic and sub-symbolic integration within a MAS was proposed. This paper was later extended in [23]. These two works are a first step towards **RG 3**. In [22], we present the potential capabilities of an integrated system consisting of logical agents and a neural network specialized in monitoring flood events for civil protection purposes. The paper describes a framework composed of a group of intelligent agents performing various tasks and communicating with each other to efficiently generate alerts during flood crisis events. In [23], the work is extended, and an initial implementation of the framework is provided. The paper presents a preliminary prototype of a MAS that autonomously collects weather warnings, categorises related images using a DL module, filters the results, and alerts human operators only if there is a reasonable certainty that a risk situation is occurring. To this end, we implemented a system[3] using a combination of logical agents and a DL component. The neural network is trained on eight classes of topographic entities to segment the images. Once the images have been segmented, a *Logical Image Descriptor* (LID) is used to generate a logical description of the segmented mask predicates. This description is then submitted to a logical agent that performs the reasoning. In our system, the reasoning is performed using a perception-fusion approach, where the MAS agents use their perceptions to reason about the environment and make decisions collectively. The paper emphasises the adoption of agents based on computational logic with a logical basis to provide verifiability, explainability, and reliability.

4 Future Works

In the future, potential research directions include: *(i)* for **RG 1**: explore abductive reasoning technique to enhance knowledge injection efficiency and effectiveness. Evaluate the framework's performance through extensive experiments in various domains; *(ii)* for **RG 2**: refine and expand evaluation metrics, including comprehensibility metrics, for better assessment of injected models. Validate metrics across different scenarios and datasets; *(iii)* for **RG 3**: explore collaborative frameworks and hybrid models combining logical and neural reasoning. Possibly evaluate their performance in real-world applications.

Overall, these future works aim to improve knowledge injection, evaluation methodologies, and the integration of logical and neural agents, contributing to more efficient and interpretable machine learning models.

---

[2] The repository containing the experiments can be found at the following link: `https://github.com/pikalab-unibo/ski-qos-jaamas-experiments-2022`

[3] The repository containing the experiments can be found at the following link: `https://github.com/AAAI-DISIM-UnivAQ/MAS_Py_FLOOD`

Acknowledgements

This PhD work is conducted under the supervision of professors Stefania Costantini (University of L'Aquila, Italy), Fosca Giannotti (Scuola Normale Superiore, Pisa, Italy), and Andrea Omicini (Alma Mater Studiorum–University of Bologna, Italy).

References

[1] Andrea Agiollo, Andrea Rafanelli, Matteo Magnini, Giovanni Ciatto & Andrea Omicini (2023): *Symbolic knowledge injection meets intelligent agents: QoS metrics and experiments*. *Autonomous Agents and Multi-Agent Systems* 37(2), p. 27, doi:`10.1007/s10458-023-09609-6`.

[2] Andrea Agiollo, Andrea Rafanelli & Andrea Omicini (2022): *Towards Quality-of-Service Metrics for Symbolic Knowledge Injection*. In Angelo Ferrando & Viviana Mascardi, editors: *WOA 2022 – 23rd Workshop "From Objects to Agents"*, *CEUR Workshop Proceedings* 3261, Sun SITE Central Europe, RWTH Aachen University, pp. 30–47. Available at `http://ceur-ws.org/Vol-3261/paper3.pdf`.

[3] Sebastian Bader, Steffen Hölldobler & Nuno C. Marques (2008): *Guiding Backprop by Inserting Rules*. In Artur S. d'Avila Garcez & Pascal Hitzler, editors: *Proceedings of the Fourth International Workshop on Neural-Symbolic Learning and Reasoning, NeSy 2008, Patras, Greece, July 21, 2008*, *CEUR Workshop Proceedings* 366, CEUR-WS.org. Available at `https://ceur-ws.org/Vol-366/paper-5.pdf`.

[4] Samy Badreddine, Artur S. d'Avila Garcez, Luciano Serafini & Michael Spranger (2022): *Logic Tensor Networks*. *Artif. Intell.* 303, p. 103649, doi:`10.1016/j.artint.2021.103649`.

[5] Antoine Bordes, Nicolas Usunier, Alberto García-Durán, Jason Weston & Oksana Yakhnenko (2013): *Translating Embeddings for Modeling Multi-relational Data*. In Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani & Kilian Q. Weinberger, editors: *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pp. 2787–2795. Available at `https://proceedings.neurips.cc/paper/2013/hash/1cecc7a77928ca8133fa24680a88d2f9-Abstract.html`.

[6] Roberta Calegari, Giovanni Ciatto & Andrea Omicini (2020): *On the integration of symbolic and sub-symbolic techniques for XAI: A survey*. 14, pp. 7–32, doi:`10.3233/IA-190036`.

[7] Edward Choi, Mohammad Taha Bahadori, Le Song, Walter F. Stewart & Jimeng Sun (2017): *GRAM: Graph-based Attention Model for Healthcare Representation Learning*. In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*, ACM, pp. 787–795, doi:`10.1145/3097983.3098126`.

[8] Stefania Costantini & Arianna Tocchio (2002): *A Logic Programming Language for Multi-agent Systems*. In Sergio Flesca, Sergio Greco, Nicola Leone & Giovambattista Ianni, editors: *Logics in Artificial Intelligence, European Conference, JELIA 2002, Cosenza, Italy, September, 23-26, Proceedings*, *Lecture Notes in Computer Science* 2424, Springer, pp. 1–13, doi:`10.1007/3-540-45757-7_1`.

[9] Stefania Costantini & Arianna Tocchio (2004): *The DALI Logic Programming Agent-Oriented Language*. In José Júlio Alferes & João Alexandre Leite, editors: *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings*, *Lecture Notes in Computer Science* 3229, Springer, pp. 685–688, doi:`10.1007/978-3-540-30227-8_57`.

[10] Wang-Zhou Dai & Stephen H. Muggleton (2021): *Abductive Knowledge Induction from Raw Data*. In Zhi-Hua Zhou, editor: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, ijcai.org, pp. 1845–1851, doi:`10.24963/ijcai.2021/254`.

[11] Wang-Zhou Dai, Qiu-Ling Xu, Yang Yu & Zhi-Hua Zhou (2019): *Bridging Machine Learning and Logical Reasoning by Abductive Learning*. In Hanna M. Wallach et. al., editor: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019,*

December 8-14, 2019, Vancouver, BC, Canada, pp. 2811–2822. Available at `https://proceedings.neurips.cc/paper/2019/hash/9c19a2aa1d84e04b0bd4bc888792bd1e-Abstract.html`.

[12] Tirtharaj Dash, Sharad Chitlangia, Aditya Ahuja & Ashwin Srinivasan (2022): *A review of some techniques for inclusion of domain-knowledge into deep neural networks*. *Scientific Reports* 12(1), p. 1040, doi:`10.1038/s41598-021-04590-0`.

[13] Michelangelo Diligenti, Soumali Roychowdhury & Marco Gori (2017): *Integrating Prior Knowledge into Deep Learning*. In Xuewen Chen, Bo Luo, Feng Luo, Vasile Palade & M. Arif Wani, editors: *16th IEEE International Conference on Machine Learning and Applications, ICMLA 2017, Cancun, Mexico, December 18-21, 2017*, IEEE, pp. 920–923, doi:`10.1109/ICMLA.2017.00-37`.

[14] Michael Gelfond & Vladimir Lifschitz (1988): *The Stable Model Semantics for Logic Programming*. In Robert A. Kowalski & Kenneth A. Bowen, editors: *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*, MIT Press, pp. 1070–1080.

[15] Hao & Basura Fernando (2023): *Fine-Grained Regional Prompt Tuning for Visual Abductive Reasoning*. *CoRR* abs/2303.10428, doi:`10.48550/arXiv.2303.10428`. arXiv:`2303.10428`.

[16] Antonis C. Kakas, Robert A. Kowalski & Francesca Toni (1992): *Abductive Logic Programming*. *J. Log. Comput.* 2(6), pp. 719–770, doi:`10.1093/logcom/2.6.719`.

[17] Antonis C. Kakas & Paolo Mancarella (1990): *Generalized Stable Models: A Semantics for Abduction*. In: *9th European Conference on Artificial Intelligence, ECAI 1990, Stockholm, Sweden, 1990*, pp. 385–391.

[18] Tomis Kapitan (1992): *Peirce and the Autonomy of Abductive Reasoning*. *Erkenntnis (1975-)* 37(1), pp. 1–26, doi:`10.1007/BF00220630`.

[19] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester & Luc De Raedt (2021): *Neural probabilistic logic programming in DeepProbLog*. *Artif. Intell.* 298, p. 103504, doi:`10.1016/j.artint.2021.103504`.

[20] Raymond J Mooney (2000): *Integrating abduction and induction in machine learning*. *Abduction and Induction: essays on their relation and integration*, pp. 181–191, doi:`10.1007/978-94-017-0606-3`.

[21] Luc De Raedt, Sebastijan Dumancic, Robin Manhaeve & Giuseppe Marra (2020): *From Statistical Relational to Neuro-Symbolic Artificial Intelligence*. In Christian Bessiere, editor: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, ijcai.org, pp. 4943–4950, doi:`10.24963/ijcai.2020/688`.

[22] Andrea Rafanelli, Stefania Costantini & Giovanni De Gasperis (2022): *A Multi-Agent-System framework for flooding events*. In Angelo Ferrando & Viviana Mascardi, editors: *WOA 2022 – 23rd Workshop "From Objects to Agents"*, *CEUR Workshop Proceedings* 3261, Sun SITE Central Europe, RWTH Aachen University, pp. 30–47. Available at `http://ceur-ws.org/Vol-3261/paper11.pdf`.

[23] Andrea Rafanelli, Stefania Costantini & Giovanni De Gasperis (2023): *Neural-logic multi-agent system for flood event detection*. *Intelligenza Artificiale* 17, pp. 19–35, doi:`10.3233/IA-230004`.

[24] Andrea Rafanelli, Stefania Costantini & Andrea Omicini (2022): *Position paper: On the role of abductive reasoning in semantic image segmentation*. In Agostino Dovier, Angelo Montanari & Andrea Orlandini, editors: *22nd International Conference of the Italian Association for Artificial Intelligence (AIxIA 2022), Udine, Italy, November 28-December 2, 2022*, *CEUR Workshop Proceedings* 3419, CEUR-WS.org. Available at `https://ceur-ws.org/Vol-3419/paper9.pdf`.

[25] Andrea Rafanelli, Matteo Magnini, Andrea Agiollo, Giovanni Ciatto & Andrea Omicini (under revision): *Are Symbolic Knowledge Injection Techniques Robust Against Data Quality Degradation?*

[26] Efthymia Tsamoura, Timothy M. Hospedales & Loizos Michael (2021): *Neural-Symbolic Integration: A Compositional Perspective*. In: *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, AAAI Press, pp. 5051–5060, doi:`10.1609/aaai.v35i6.16639`.

[27] Quan Wang, Bin Wang & Li Guo (2015): *Knowledge Base Completion Using Embeddings and Rules*. In Qiang Yang & Michael J. Wooldridge, editors: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, AAAI Press, pp. 1859–1866. Available at `http://ijcai.org/Abstract/15/264`.

[28] Jingyi Xu, Zilu Zhang, Tal Friedman, Yitao Liang & Guy Van den Broeck (2018): *A Semantic Loss Function for Deep Learning with Symbolic Knowledge*. In Jennifer G. Dy & Andreas Krause, editors: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, *Proceedings of Machine Learning Research* 80, PMLR, pp. 5498–5507. Available at `http://proceedings.mlr.press/v80/xu18h.html`.

[29] Jianbo Yu & Guoliang Liu (2021): *Extracting and inserting knowledge into stacked denoising auto-encoders*. *Neural Networks* 137, pp. 31–42, doi:`10.1016/j.neunet.2021.01.010`.

[30] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht & Oriol Vinyals (2021): *Understanding deep learning (still) requires rethinking generalization*. *Commun. ACM* 64(3), pp. 107–115, doi:`10.1145/3446776`.

Explainable Answer-set Programming

Tobias Geibinger
Knowledge-based Systems Group, Institute for Logic and Computation, TU Wien, Austria
`tobias.geibinger@tuwien.ac.at`

The interest in explainability in artificial intelligence (AI) is growing vastly due to the near ubiquitous state of AI in our lives and the increasing complexity of AI systems. Answer-set Programming (ASP) is used in many areas, among them are industrial optimisation, knowledge management or life sciences, and thus of great interest in the context of explainability. To ensure the successful application of ASP as a problem-solving paradigm in the future, it is thus crucial to investigate explanations for ASP solutions. Such an explanation generally tries to give an answer to the question of why something is, respectively is not, part of the decision produced or solution to the formulated problem. Although several explanation approaches for ASP exist, almost all of them lack support for certain language features that are used in practice. Most notably, this encompasses the various ASP extensions that have been developed in the recent years to enable reasoning over theories, external computations, or neural networks. This project aims to fill some of these gaps and contribute to the state of the art in explainable ASP. We tackle this by extending the language support of existing approaches but also by the development of novel explanation formalisms, like contrastive explanations.

1 Introduction and Problem Definition

The topic of "explainability" in *artificial intelligence* (AI) has become increasingly prominent over the recent years and regulation is being discussed.

Nowadays, there is a multitude of AI systems in use for decision-making and problem-solving. In this project we concern ourselves with *Answer-set Programming* (ASP) [12] which is a popular declarative problem-solving paradigm used in many domains [13, 15]. ASP allows for the encoding of problems in a succinct way. Such a problem encoding usually consists of a set of *rules* representing the underlying constraints of the problem and sets of *facts* describing concrete scenarios. The solutions are given in terms of *answer-sets*, which capture the evaluation of the rules.

Extensions of ASP have been considered to model and solve problems in practice for a variety of reasons. Sometimes language extensions ease the modelling of the problem and make the representation more concise. However, ASP extensions can also enable new forms of reasoning and problem-solving, like for example, over custom theories [18], external computations [8, 20], or neural networks [38].

The same reasons that make ASP a popular problem-solving paradigm, also make it attractive in the context of explainable AI. ASP is a symbolic and rule-based approach and thus has good prerequisites for humanly readable and intuitive explanations. In ASP, they usually amount to justifications as to why certain elements are (or are not) contained in an answer-set or why there is no answer-set at all.

While there is a rich body of work on explainability for standard ASP [17], the same is not the case for its numerous extensions. In fact, most current explanation approaches do not directly support basic language features like variables or disjunction. Furthermore, the notion of *contrastive explanations* has recently been brought to attention to the AI community by Miller [26], who argues in favour of them due to their established history in the psycho- and sociological communities.

The goal of this project is thus to close those gaps by focusing on (contrastive) notions of explainability for ASP extensions.

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 423–429, doi:10.4204/EPTCS.385.52

2 Background

Answer-set Programming We consider disjunctive *Answer-set Programming* (ASP) [12]. An ASP program is a (finite) set of (disjunctive) rules of the form $a_1 \vee \cdots \vee a_n \leftarrow b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_m$, where all $a_1, \ldots a_n$ and $b_1, \ldots, b_m$ are function-free first-order atoms. The head is the set of atoms $a_1, \ldots a_n$ before the implication symbol $\leftarrow$, and the body are the atoms and negated atoms $b_1, \ldots, b_m$. The intuitive meaning of a rule is that if all atoms $b_1, \ldots, b_k$ can be derived, and there is no evidence for any of the atoms $b_{k+1}, \ldots, b_m$ (i.e., the rule fires), then at least one of $a_1, \ldots, a_n$ has to be derived. A rule with an empty body (i.e., $m = n = 0$) is called a *fact*, with $\leftarrow$ usually omitted and a rule with empty head (i.e., $n = 0$) is a *constraint*. Furthermore, whenever the head of a rule is a singleton, we call the rule *normal*. The semantics is defined as usual in terms of particular *(Herbrand) models* of the *grounding* of the program and an *answer-set* I is a $\subseteq$-minimal model of the Gelfond-Lifschitz reduct $P^I = \{a_1 \vee \cdots \vee a_n \leftarrow b_1, \ldots, b_k \mid r \in P,\ \{b_{k+1}, \ldots b_m\} \cap I = \emptyset\}$.

Several ASP language extensions exist, some are mainly syntactic sugar enabling more concise representation of problems, like *aggregates* [14] or *choice rules* [19]. However, others either extend the semantics of ASP or generalise them to full propositional theories as *equilibrium logic* [29] does. Extensions of the semantics include reasoning over linear constraints [2] or arbitrary theories [20], but they also facilitate neuro-symbolic computation, like for example, *NeurASP* [38] or LP^{MLN} programs [22].

Explanations for ASP Several approaches for explaining consistent ASP programs exist. The goal is usually to give a justification as to why an atom is contained respectively not contained in an answer-set. *Off-line justifications* [32, 34, 35] are labelled directed graphs providing reasons for why an atom is (or is not) contained in a given answer-set. Intuitively, this can be seen as a derivation of the truth value of the respective atom using the rules of the program. *Causal justifications* [16] provide explanations for literals in a given answer-set by means of an alternative semantics for logic programs with disjunction. The basis for causal justifications are *causal terms* which are algebraic terms representing joint causation through multiplication and alternative causes through addition. *Why-not provenance* [36] provides justifications for the truth values of atoms w.r.t answer-set semantics for normal logic programs. In particular, it explains literals in a general fashion for a given program without the need of a specific answer-set. *Witnesses* [37] give reason as to why an atom is contained in a particular answer-set by giving a resolution proof. Similary, the system `xclingo` [5] prints out a rule derivation for the atom to be explained.

Explanations for inconsistent programs are usually generated to help with debugging programs and make them consistent. For example, the systems `spock` [21] and `Ouroboros` [27] explain why an interpretation is not an answer-set via program transformations which encode the given ASP program into a meta-program. Answer-sets of this program contain an interpretation which is not an answer-set and the reasons why. Saribatur et al. [33] employed `Ouroboros` in their *abstraction* method and used it to generate explanations for inconsistent instances in certain problem domains. Further debugging approaches are DWASP [7] and `stepping` [28]. DWASP was introduced for the ASP solver WASP [1] and works by amending *debug atoms* to programs. The inconsistency of the resulting *debugging program* can then be explained in terms of sets of debug atoms that cannot jointly hold. In difference, `stepping` takes lets the user perform the solving process by applying one applicable rule after the other.

Contrastive Explanation To answer questions like "Why P and rather than Q?" is the basis of contrastive explanations [23]. It has been argued that such explanations are intuitive for humans to understand and to produce and also that standard why questions contain a hidden *contrast case*, e.g., "Why P?" represents "Why P rather than not P?" [26]. Lipton [23] defines an answer to such a question as the *difference condition*, stating that the answer contains a cause for P that is missing for not Q.

Example 1. *Consider an algorithm that classifies bugs and suppose the algorithm has classified a particular bug as a beetle. A potential non-contrastive explanation for this classification is to present the values of the features, i.e., the bug is a beetle because it has 8 legs, 2 eyes and 2 wings. If we interpret the question "Why is the bug a beetle?" as "Why is the bug a beetle instead of another bug?", then an adequate explanation is to highlight that the bug has 2 eyes, if it had 5 eyes, then it would be a fly.*

3 Research Goals

Explaining ASP Extensions and Advanced Features One of our main research directions will be the study of explainability in the context of the ASP extensions, as the mentioned existing explanation approaches often do not support them. Besides existing approaches, we intend to investigate explanations based on *abduction*, i.e. the process of determining which hypothesis need to be added to a theory to explain given observations. This reasoning method has been studied for logic programs [11] and – depending on the use-case – can be also be employed as an explanation approach. This seems interesting in the context of contrastive explanations [23], which can be expressed through abductive reasoning. [25]

Furthermore, it is worthwhile to investigate the different ways of how the "inner workings" of extensions can be considered in the explanations. We intend to consider the following settings. In a black-box setting the explanation approach does not take into account how the ASP extension determines the truth values of its atoms. For HEX programs, this would mean that external atoms are simply considered true or false depending on the context and their truth values. The direct opposite to the previously mentioned approach would be the white-box setting, where the semantics of the ASP extension is fully known and considered in the explanation. In the case of NeurASP, the explanation would need to incorporate information about the neural network. The middle way between the black-box and the white-box approach would be a gray-box setting, where we do not have full access to the inner workings of the ASP extension but we do have some information. For external computations or neural atoms, the explanation approach could take, for example, certain input/output relationships into account.

Explaining Instead of Debugging Inconsistency The explanation approaches for inconsistent programs mentioned in Section 2 are all intended to facilitate debugging of answer-set programs. While those explanations focused on debugging are of course all valid, they may not be useful in all contexts, for example when the explanation is needed by an end-user and not an ASP engineer. Our aim is thus to investigate explanation approaches for inconsistency that are more akin to the ones for consistent programs. Related work in this direction was done by Damásio et al. [6] who extended the debugging system `spock` with providing justifications for atoms in the case that the program is actually consistent.

Explainability and Equilibrium Logic Certainly, a formal proof can be seen as an explanation as to why some proposition has to follow from the given premises. In equilibrium logic [29] this is not different.While a proof system for equilibrium logic based on tableaux methods exists [30], there are two main issues that make it problematic for usage in explainability. The first is that the tableaux approach contains multiple steps and it is difficult to obtain humanly readable explanations from the tableaux. The other is that the axiomatized entailment relation is similar to what is also called *skeptical inference*, i.e. the inference holds if the proposition is contained in each equilibrium model (answer-set). However, as we have seen in Section 2, explanation approaches generally concern themselves with the question of why something is true in *some* answer-set and this case is arguably also more important in practice. The corresponding entailment relation is often called *brave* or *credulous inference*. We thus intend to develop proof systems for this type of inference that are more akin to the more understandable sequent calculi, similar to what was done for related nonmonotonic formalisms [4]. Furthermore, abduction has also been investigated for equilibrium logic [31] and we intend to investigate abductive explanations.

Towards Practical Algorithms A natural step in our study of explainability for ASP is to produce implementations for the methods we introduce. An approach that is often employed in generating inconsistent explanations, like for `spock` [21] and `Ouroboros` [27], is to encode this in ASP itself. We also aim to provide implementations of the approaches we introduce in certain problem domains, like scheduling or planning, where ASP has been shown to produce good solutions and benchmark data is available. Specific questions may arise in those areas and whenever possible we aim to investigate tailored enhancements for said domains. Any developed explanation system should also be interactive, as it has been argued that the act of explanation is not static and more of a dialogue [26]. Hence, we aim to build systems, where the user can guide the explanation process.

4 Research Status & Outlook

The project was officially started in the summer of 2022 and is expected to last for 3 years. Currently, we are focused on two research directions. The first is investigating formal notions of justification, akin to off-line justifications [32] and witnesses [37], for disjunctive programs comprised of Abstract Constraint Atoms [24]. Programs defined over such atoms neatly generalise several language extensions.

Example 2. *Consider the aggregate* $\#\mathtt{sum}\{\mathtt{2:a,1:b,1:c}\} > 1$*. Suppose we have a model* $I_1 = \{a,b,c\}$*. Then, an intuitive justification as to why* I_1 *is a model of the aggregate, is that it is because* $a \in I_1$*. Of course, it is also valid to say that* I_1 *is a model because* $b,c \in I_1$*. Consider* $I_2 = \{b\}$*. Since* I_2 *is not a model of the aggregate, we would like to know why. The justification in this case is that neither a nor c are satisfied by* I_2*, which, since b cannot be false in* I_2*, would be a requirement.*

As the example shows, this approach is purely based on semantics and does not take rule application into account. Therefore, we are also working on more syntactic notions that can be used together with the above concept is to use them both in an interactive explanation system that supports choice, aggregates and disjunctive rules. A conference paper on this topic has recently been accepted [9].

The other direction we are pursuing is that of contrastive explanations with and for ASP.

Example 3. *Consider the bug classification from Example 1 and suppose it is expressed by an ASP program P. So, given the instance* $F = \{legs(6), eyes(2), wings(2)\}$*, we obtain the answer-set* $I = \{class(beetle)\}$*. The question "Why is the bug a beetle instead of another bug?" can then be formulated as the problem of finding some* F' *such that I is not the answer-set of* $P \cup F'$ *and* F' *differs minimally from F. In this case,* $F' = \{legs(6), eyes(5), wings(2)\}$*,* $F \setminus F' = \{eyes(2)\}$ *and* $F' \setminus F = \{eyes(5)\}$*.*

In the example we said that the contrastive set of facts should be minimally different. A natural choice would be minimal symmetric difference, but in general this difference is problem-dependent. In the example the rules remained fixed, which is not always suited. For example, Bogatarkan et al. [3] use similar explanations for their path finding system, but they focus on relaxing constraints. On this subject, a paper, focused on a specific application, has just been accepted [10] and another is under review.

Besides the numerous research goals we have not tackled yet, there are some bigger issues with the overall topic of explainability. When Miller [26] surveyed the work on explainability, he noticed that formal notions of explainability often differ from what is accepted as explanation in the social sciences. This includes explanation approaches that are purely static mathematical objects and Miller thus argues for interactive approaches that may or may not utilise contrastiveness.

Regarding expected achievements, we plan to investigate the research topics presented in Section 3 and implement them in an explanation system to offer interactive explanations.

References

[1] Mario Alviano, Carmine Dodaro, Wolfgang Faber, Nicola Leone & Francesco Ricca (2013): *WASP: A Native ASP Solver Based on Constraint Learning*. In: *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013)*, *LNCS* 8148, Springer, pp. 54–66, doi:10.1007/978-3-642-40564-8_6.

[2] Mutsunori Banbara, Benjamin Kaufmann, Max Ostrowski & Torsten Schaub (2017): *Clingcon: The next generation*. *Theory and Practice of Logic Programming* 17(4), pp. 408–461, doi:10.1017/S1471068417000138.

[3] Aysu Bogatarkan & Esra Erdem (2020): *Explanation Generation for Multi-Modal Multi-Agent Path Finding with Optimal Resource Utilization using Answer Set Programming*. *Theory and Practice of Logic Programming* 20(6), pp. 974–989, doi:10.1017/S1471068420000320.

[4] Piero A. Bonatti (1996): *Sequent Calculi for Default and Autoepistemic Logics*. In: *Proceedings of the 5th International Conference on Theorem Proving with Analytic Tableaux and Related Methods (TABLEAUX 1996)*, *LNCS* 1071, Springer, pp. 127–142, doi:10.1007/3-540-61208-4_9.

[5] Pedro Cabalar, Jorge Fandinno & Brais Muñiz (2020): *A System for Explainable Answer Set Programming*. In: *Technical Communications of the 36th International Conference on Logic Programming (ICLP 2020)*, *EPTCS* 325, pp. 124–136, doi:10.4204/EPTCS.325.19.

[6] Carlos Viegas Damásio, João Moura Pires & Anastasia Analyti (2015): *Unifying Justifications and Debugging for Answer-Set Programs*. In: *Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015)*, *CEUR Workshop Proceedings* 1433, CEUR-WS.org. Available at `ceur-ws.org/Vol-1433/tc_84.pdf`.

[7] Carmine Dodaro, Philip Gasteiger, Benjamin Musitsch, Francesco Ricca & Kostyantyn M. Shchekotykhin (2015): *Interactive Debugging of Non-ground ASP Programs*. In: *Proceedings of 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2015)*, *LNCS* 9345, Springer, pp. 279–293, doi:10.1007/978-3-319-23264-5_24.

[8] Thomas Eiter, Michael Fink, Giovambattista Ianni, Thomas Krennwallner, Christoph Redl & Peter Schüller (2016): *A model building framework for answer set programming with external computations*. *Theory and Practice of Logic Programming* 16(4), pp. 418–464, doi:10.1017/S1471068415000113.

[9] Thomas Eiter & Tobias Geibinger (2023): *Explaining Answer-Set Programs with Abstract Constraint Atoms*. In: *Proceedings of the 32nd International Joint Conference on Artificial Intelligence (IJCAI 2023, to appear)*.

[10] Thomas Eiter, Tobias Geibinger, Nelson Higuera Ruiz & Johannes Oetsch (2023): *A Logic-based Approach to Contrastive Explainability for Neurosymbolic Visual Question Answering*. In: *Proceedings of the 32rd International Joint Conference on Artificial Intelligence (IJCAI 2023, to appear)*.

[11] Thomas Eiter, Georg Gottlob & Nicola Leone (1997): *Abduction from Logic Programs: Semantics and Complexity*. *Theoretical Computer Science* 189(1-2), pp. 129–177, doi:10.1016/S0304-3975(96)00179-X.

[12] Thomas Eiter, Giovambattista Ianni & Thomas Krennwallner (2009): *Answer Set Programming: A Primer*. In: *Reasoning Web*, Springer, pp. 40–110, doi:10.1007/978-3-642-03754-2_2.

[13] Esra Erdem, Michael Gelfond & Nicola Leone (2016): *Applications of Answer Set Programming*. *AI Magazine* 37(3), pp. 53–68, doi:10.1609/aimag.v37i3.2678.

[14] Wolfgang Faber, Nicola Leone & Gerald Pfeifer (2004): *Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity*. In: *Proceedings of the 9th European Conference on Logics in Artificial Intelligence (JELIA 2004)*, *LNCS* 3229, Springer, pp. 200–212, doi:10.1007/978-3-540-30227-8_19.

[15] Andreas Falkner, Gerhard Friedrich, Konstantin Schekotihin, Richard Taupe & Erich C. Teppan (2018): *Industrial Applications of Answer Set Programming*. *KI* 32(2), pp. 165–176, doi:10.1007/s13218-018-0548-6.

[16] Jorge Fandinno (2016): *Deriving conclusions from non-monotonic cause-effect relations*. *Theory and Practice of Logic Programming* 16(5-6), p. 670–687, doi:10.1017/S1471068416000466.

[17] Jorge Fandinno & Claudia Schulz (2019): *Answering the "why" in answer set programming – A survey of explanation approaches.* Theory and Practice of Logic Programming 19(2), pp. 114–203, doi:10.1017/S1471068418000534.

[18] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub & Philipp Wanko (2016): *Theory Solving Made Easy with Clingo 5.* In: Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016), OASIcs 52, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 2:1–2:15, doi:10.4230/OASIcs.ICLP.2016.2.

[19] Martin Gebser, Roland Kaminski, Benjamin Kaufmann & Torsten Schaub (2013): *Answer Set Solving in Practice.* Springer, doi:10.1007/978-3-031-01561-8.

[20] Martin Gebser, Roland Kaminski, Benjamin Kaufmann & Torsten Schaub (2014): *Clingo = ASP + Control: Preliminary Report.* CoRR abs/1405.3694. Available at `arxiv.org/abs/1405.3694`.

[21] Martin Gebser, Jörg Pührer, Torsten Schaub & Hans Tompits (2008): *A Meta-Programming Technique for Debugging Answer-Set Programs.* In: Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI 2008), AAAI Press, pp. 448–453. Available at `www.aaai.org/Library/AAAI/2008/aaai08-071.php`.

[22] Joohyung Lee & Yi Wang (2016): *Weighted Rules under the Stable Model Semantics.* In: Proceedings of the 15th International Conference on Principles of Knowledge Representation and Reasoning (KR 2016), AAAI Press, pp. 145–154. Available at `www.aaai.org/ocs/index.php/KR/KR16/paper/view/12901`.

[23] Peter Lipton (1990): *Contrastive Explanation.* Royal Institute of Philosophy Supplement 27, p. 247–266, doi:10.1017/S1358246100005130.

[24] Victor W. Marek & Miroslaw Truszczynski (2004): *Logic Programs with Abstract Constraint Atoms.* In: Proceedings of the 19th National Conference on Artificial Intelligence (AAAI 2004), AAAI Press / The MIT Press, pp. 86–91. Available at `www.aaai.org/Library/AAAI/2004/aaai04-014.php`.

[25] Joao Marques-Silva (2023): *Logic-Based Explainability in Machine Learning.* In Leopoldo Bertossi & Guohui Xiao, editors: Reasoning Web. Causality, Explanations and Declarative Knowledge, Springer Nature Switzerland, Cham, pp. 24–104, doi:10.1007/978-3-031-31414-8_2.

[26] Tim Miller (2019): *Explanation in artificial intelligence: Insights from the social sciences.* Artificial Intelligence 267, pp. 1–38, doi:10.1016/j.artint.2018.07.007.

[27] Johannes Oetsch, Jörg Pührer & Hans Tompits (2010): *Catching the Ouroboros: On debugging non-ground answer-set programs.* Theory and Practice of Logic Programming 10(4-6), pp. 513–529, doi:10.1017/S1471068410000256.

[28] Johannes Oetsch, Jörg Pührer & Hans Tompits (2018): *Stepwise debugging of answer-set programs.* Theory and Practice of Logic Programming 18(1), pp. 30–80, doi:10.1017/S1471068417000217.

[29] David Pearce (2006): *Equilibrium logic.* Annals of Mathematics and Artificial Intelligence 47, pp. 3–41, doi:10.1007/s10472-006-9028-z.

[30] David Pearce, Inman P. de Guzmán & Agustín Valverde (2000): *A Tableau Calculus for Equilibrium Entailment.* In: Proceedings of the 9th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2000), LNCS 1847, Springer, pp. 352–367, doi:10.1007/10722086_28.

[31] David Pearce & Agustín Valverde (2001): *Abduction in equilibrium logic.* In: Proceeding of the 1st International Answer Set Programming Workshop (ASP 2001). Available at `http://www.cs.nmsu.edu/%7Etson/ASP2001/23.ps`.

[32] Enrico Pontelli, Tran Cao Son & Omar Elkhatib (2009): *Justifications for logic programs under answer set semantics.* Theory and Practice of Logic Programming 9(1), pp. 1–56, doi:10.1017/S1471068408003633.

[33] Zeynep G. Saribatur, Thomas Eiter & Peter Schüller (2021): *Abstraction for non-ground answer set programs.* Artificial Intelligence 300, p. 103563, doi:10.1016/j.artint.2021.103563.

[34] Ly Ly T. Trieu, Tran Cao Son & Marcello Balduccini (2021): *exp(ASPc) : Explaining ASP Programs with Choice Atoms and Constraint Rules*. In: *Technical Communications of the 37th International Conference on Logic Programming (ICLP 2021)*, *EPTCS* 345, pp. 155–161, doi:10.4204/EPTCS.345.28.

[35] Ly Ly T. Trieu, Tran Cao Son & Marcello Balduccini (2022): *xASP: An Explanation Generation System for Answer Set Programming*. In: *Proceddings of the 16th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2022)*, *LNCS* 13416, Springer, pp. 363–369, doi:10.1007/978-3-031-15707-3_28.

[36] Carlos Viegas Damásio, Anastasia Analyti & Grigoris Antoniou (2013): *Justifications for Logic Programming*. In: *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013)*, *LNCS* 8148, Springer, pp. 530–542, doi:10.1007/978-3-642-40564-8_53.

[37] Yisong Wang, Thomas Eiter, Yuanlin Zhang & Fangzhen Lin (2022): *Witnesses for Answer Sets of Logic Programs*. *ACM Transactions on Computational Logic*, doi:10.1145/3568955.

[38] Zhun Yang, Adam Ishay & Joohyung Lee (2020): *NeurASP: Embracing neural networks into answer set programming*. In: *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI 2020)*, pp. 1755–1762, doi:10.24963/ijcai.2020/243.