EPTCS 402

Proceedings of the

18th International Workshop on

Logical and Semantic Frameworks, with Applications

and 10th Workshop on

Horn Clauses for Verification and Synthesis

Rome, Italy & Paris, France, 1-2 July, 2023 & 23rd April 2023

Edited by: Temur Kutsia, Daniel Ventura, David Monniaux and José F. Morales

Published: 23rd April 2024
DOI: 10.4204/EPTCS.402
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

This volume contains the post-proceedings of two events: the 18th Workshop on Logical and Semantic Frameworks with Applications (LSFA 2023) and the 10th Workshop on Horn Clauses for Verification and Synthesis (HCVS 2023).

1 LSFA 2023

LSFA 2023 was held on July 1-2, 2023 in Rome, Italy, organized by the Sapienza Università di Roma and co-located with FSCD 2023, the 8th International Conference on Formal Structures for Computation and Deduction. The aim of the LSFA series of workshops is to bring together researchers and students interested in theoretical and practical aspects of logical and semantic frameworks and their applications. The covered topics include proof theory, type theory and rewriting theory, specification and deduction languages, and formal semantics of languages and systems. For LSFA 2023, six regular papers were accepted for presentation out of ten submissions, with three reviewers per submission. After the meeting, revised versions of the papers were reviewed again, from which five regular papers were finally included in this volume. In addition, the workshop program included three talks by distinguished invited speakers Pablo Barenbaum (Universidad de Buenos Aires), Cynthia Kop (Radboud University Nijmegen), and Brigitte Pientka (McGill University). We express our sincere gratitude to them.

We want to thank the PC members and the additional reviewers for doing a great job providing high-quality reviews. Many thanks to the LSFA 2023 organizers, Daniele Nantes Sobrinho and David Cerna, the FSCD 2023 General Chair Daniele Gorla, and the FSCD Workshop Chair Ivano Salvo. All their valuable time spent was indispensable in guaranteeing the success of the workshop.

Temur Kutsia
Daniel Ventura
(LSFA 2023 PC co-chairs)

Program committee

Sandra Alves	Universidade de Porto	Portugal
Carlos Areces	Universidad Nacional de Cordoba	Argentina
Mauricio Ayala-Rincón	Universidade de Brasília	Brazil
Haniel Barbosa	Universidade Federal de Minas Gerais	Brazil
Eduardo Bonelli	Stevens Institute of Technology	USA
David Cerna	Inst. Computer Science, Czech Academy of Sciences	Czechia
Alejandro Diaz-Caro	UNQ & ICC CONICET-UBA	Argentina
Marcelo Finger	Universidade de São Paulo	Brazil
Pascal Fontaine	University of Liege	Belgium
Lourdes del Carmen González Huesca	UNAM	Mexico
Giulio Guerrieri	Aix-Marseille Université	France
Fairouz Kamareddine	Heriot-Watt University	UK
Delia Kesner	Université Paris Cité	France
Marina Lenisa	Università di Udine	Italy
Mircea Marin	West University of Timisoara	Romania
Mariano Moscato	National Institute of Aerospace	USA
Daniele Nantes-Sobrinho	Imperial College London	UK
Miguel Pagano	Universidad Nacional de Córdoba	Argentina
Valeria de Paiva	Topos Institute, Berkeley	USA
Cleo Pau	Johannes Kepler University Linz	Austria
Elaine Pimentel	University College London	UK
Paolo Pistone	Università Roma Tre	Italy
Femke van Raamsdonk	Vrije Universiteit Amsterdam	Netherlands
Andrew Reynolds	University of Iowa	USA
Wolfgang Schreiner	Johannes Kepler University Linz	Austria

External Reviewers

André Luiz Galdino, Jonathan Laurent

Organization

Daniele Nantes-Sobrinho	Imperial College London	UK
David Cerna	Inst. Computer Science, Czech Academy of Sciences	Czechia

2 HCVS 2023

The *workshop on Horn clauses for verification and synthesis* (HCVS) series aims to bring together researchers working in the two communities of constraint/ logic programming (e.g., ICLP and CP), program verification (e.g., CAV, TACAS, and VMCAI), and automated deduction (e.g., CADE, IJCAR), on the topics of Horn clause based analysis, verification, and synthesis.

The 10th edition took place on Sunday 23, April 2023 at the *Institut Henri Poincaré* in Paris, France, as part of ETAPS (European joint conferences on theory and practice of software). The workshop had received 7 submissions, 6 of which had been selected for presentation. Participants could opt for presentation-only or publication in proceedings. The program was supplemented by invited presentations.

We want to thank the PC members and additional reviewers for doing a great job providing high-quality reviews. The ETAPS local organization was splendid. We also thank Institut Henri Poincaré for providing the venue.

David Monniaux
Jose F. Morales
(HCVS 2023 PC co-chairs)

Program committee

Nikolaj Bjørner	Microsoft Research	USA
Evelyne Contejean	CNRS, LMF	France
Stefania Dumbrava	ENSIIE, Samovar	France
Fabio Fioravanti	Università di Chieti-Pescara	Italy
Pierre-Loïc Garoche	ENAC, LII	France
Ekaterina Komendantskaya	Heriot-Watt Univ. & Univ. of Southampton	UK
David Monniaux	CNRS, Verimag	France
José Francisco Morales	IMDEA Software	Spain
Jorge A. Navas	Certora	
Philipp Rümmer	Univ. Regensburg	Germany
Hiroshi Unno	Univ. Tsukuba	Japan

Proof Terms for Higher-Order Rewriting and Their Equivalence (Invited Talk)

Pablo Barenbaum
Universidad de Buenos Aires, CONICET/UNQ/ICC
Buenos Aires, Argentina
pbarenbaum@dc.uba.ar

Proof terms are syntactic expressions that represent computations in term rewriting. They were introduced by Meseguer and exploited by van Oostrom and de Vrijer to study equivalence of reductions in (left-linear) first-order term rewriting systems. In this joint work with Eduardo Bonelli, we study the problem of extending the notion of proof term to higher-order rewriting, which generalizes the first-order setting by allowing terms with binders and higher-order substitution. In previous works that devise proof terms for higher-order rewriting, such as Bruggink's, it has been noted that the challenge lies in reconciling composition of proof terms and higher-order substitution (β-equivalence). This led Bruggink to reject "nested" composition, other than at the outermost level. We propose a notion of higher-order proof term we dub rewrites that supports nested composition. We then define two notions of equivalence on rewrites, namely permutation equivalence and projection equivalence, and show that they coincide.

Cutting a Proof into Bite-Sized Chunks (Incrementally Proving Termination in Higher-Order Term Rewriting) (Invited Talk)

Cynthia Kop
Radboud University
Nijmegen, the Netherlands
C.Kop@cs.ru.nl

In this talk, I will discuss a number of methods to prove termination of higher-order term rewriting systems, with a particular focus on large systems. In first-order term rewriting, the dependency pair framework can be used to split up a large termination problem into multiple (much) smaller components that can be solved individually. This is important because a large problem may take exponentially longer to solve in one go than solving each of its components.

Unfortunately, while there are higher-order versions of several of these methods, they often fail to simplify a problem enough. Here, we will explore some of these techniques and their limitations, and discuss what else can be done to incrementally build a termination proof for higher-order systems.

Mechanizing Session-Types: Enforcing Linearity without Linearity (Invited Talk)

Brigitte Pientka
School of Computer Science
McGill University
Montreal, Canada
bpientka@cs.mcgill.ca

Process calculi provide a tool for the high-level description of interactions, communications, and synchronizations between a collection of independent processes. Session types allow us to statically verify that processes communicate according to prescribed protocols. Hence, they rule out a wide class of communication-related bugs before executing a given process. They also statically guarantee safety properties such as session fidelity and deadlock freedom, analogous to preservation and progress in the simply typed lambda-calculus.

Although there have been many efforts to mechanize process calculi such as the pi-calculi in proof assistants, mechanizing these systems remains an art. Process calculi use channel or action names to specify process interactions, and they often feature rich binding structures and semantics such as channel mobility. Both of these features can be challenging to mechanize, for we must track names to avoid conflicts, ensure that alpha-equivalence and renaming are well-defined, etc. Moreover, session types employ a linear type system, where variables cannot be implicitly copied or dropped, and therefore, many mechanizations of these systems require modeling the context and carefully ensuring that its variables are handled linearly.

In this talk, I demonstrate a technique to localize linearity conditions as additional predicates embedded within type judgments, which allows us to use unrestricted typing contexts instead of linear ones. This technique is especially relevant when leveraging (weak) higher-order abstract syntax to defer the intricate channel mobility and bindings that arise in a session typed system. In particular, I discuss the mechanization of a session typed system based on classical linear logic in the proof assistant Beluga, which uses the logical framework LF as its encoding language.

This is joint work with Chuta Sano and Ryan Kavanagh and is based on the [1].

References

[1] Chuta Sano, Ryan Kavanagh & Brigitte Pientka (2023): *Mechanizing Session-Types using a Structural View: Enforcing Linearity without Linearity*. *Proc. ACM Program. Lang.* 7(OOPSLA2), pp. 374–399, doi:10.1145/3622810.

Stalnaker's Epistemic Logic in Isabelle/HOL.[*]

Laura P. Gamboa Guzman
Iowa State University
Ames, IA 50011, USA
lpgamboa@iastate.edu

Kristin Y. Rozier
Iowa State University
Ames, IA 50011, USA
kyrozier@iastate.edu

The foundations of formal models for epistemic and doxastic logics often rely on certain logical aspects of modal logics such as S4 and S4.2 and their semantics; however, the corresponding mathematical results are often stated in papers or books without including a detailed proof, or a reference to it, that allows the reader to convince themselves about them. We reinforce the foundations of the epistemic logic S4.2 for countably many agents by formalizing its soundness and completeness results for the class of all weakly-directed pre-orders in the proof assistant Isabelle/HOL. This logic corresponds to the knowledge fragment, i.e., the logic for formulas that may only include knowledge modalities in Stalnaker's system for knowledge and belief. Additionally, we formalize the equivalence between two axiomatizations for S4, which are used depending on the type of semantics given to the modal operators, as one is commonly used for the relational semantics, and the other one arises naturally from the topological semantics.

1 Introduction

Epistemic logics are a family of logics that allow us to reason about knowledge among a group of agents, as well as their knowledge about other's knowledge[12]. Reasoning about knowledge is useful for detecting and identifying faults during the operation of complex critical systems [7, 27], where important safety properties are formalized using a modal language that combines temporal, in particular, LTL (Linear Temporal Logic), and epistemic modal operators, so to verify the correctness of the system using model checking and related formal fault-detection techniques [9, 21, 24].

When it comes to modal logics for knowledge, most of these logics correspond to normal logics between S4 and S5 [11, 25]. In particular, we consider Stalnaker's epistemic logic, which coincides with the logic S4.2. It is known that this logic strictly stronger than S4, but weaker than S5 [8]. This logic is known to be sound and complete with respect to all weakly directed S4-frames, that is, all frames consisting of reflexive and transitive binary relations that are confluent [23], but this proof is often omitted in textbooks where most extensions to system K (the weakest normal modal logic) are usually treated informally.

Additionally, we encode in Isabelle/HOL the axiomatization of S4 obtained from the study of the topological interpretation for modal languages, which was introduced prior to the relational one that is more commonly found in the literature. This topological interpretation is done by reading the modal necessity operator as an interior operator on a topological space, for which is known that the modal logic S4 is complete with respect to all topological spaces [1]. The preferred axiomatization for the logic of topological spaces differs from the one presented in [15], not only from the set of axioms, but also the deductive rules, since it captures the axioms for an interior operator instead of a reflexive and transitive binary relation. As a consequence, this makes the topological axiomatization not directly recognizable as a normal modal logic, since the deduction rules seem to be weaker at first glance. Since several authors

[*]Supported by NSF CAREER Award CNS-1552934 and NSF:CCRI Award CNS-2016592.

have been recently developing topological semantics for notions of knowledge and belief [2, 4, 3, 16], we provide a formalization for this result, which often gets briefly mentioned and applied without being proved in detail.

Contributions

We formalize Stalnaker's epistemic logic, which is expressively equivalent to S4.2 [25], as well as some intermediate results for the underlying propositional logic and the modal logics K, .2, and S4 mainly regarding rewriting rules, properties for maximal consistent sets of formulas, and frame properties that are induced by the chosen set of axioms in the proof assistant Isabelle/HOL [17]. Our main result is a formalization of the soundness and completeness of Stalnaker's epistemic logic (restricted to countably many agents) with respect to all weakly directed(also refered to as *confluent* or *convergent* in the literature [22, 23]) S4 frames, this is, all frames consisting of a non-empty set W and a binary relation R_i on W, one for each agent label i, that is reflexive, transitive, and that satisfies the property described by the following condition

$$\forall x \forall y \forall z (xR_iy \land xR_iz \implies \exists w\, yR_iw \land zR_iw).$$

The proof uses a Henkin-style completeness method, which is commonly used for these kinds of logics [5] and was already available on Isabelle's Archive of Formal Proofs [14].

As far as we know, all systems corresponding to some multi-agent epistemic logic already formalized in Isabelle/HOL, which are all contained in [14] (the ground base for our formalization), were complete with respect to a class of frames characterized by a universal formula, i.e., a property of frames given by a first order formula of the form $\forall \overline{x}\, \phi(\overline{x})$, where $\phi(\overline{x})$ is a quantifier-free formula with variable symbols in $\overline{x} = (x_1, \ldots, x_n)$. However, the logic S4.2 is complete with respect to a class of frames that cannot be characterized using a universal formula; instead it is characterized by a universal-existential formula. This universal-existential characterization makes it harder to formalize its completeness result, since one has to show the existence of an object in the universe of the canonical model satisfying a condition on a union of consistent sets of formulas. For this, we followed an argument given by Stalnaker in [26] that includes a set of theorems that are consequences of the axiom (.2) in K, which imply the consistency of a set obtained by taking the union of all known facts for an agent in two different worlds that were accessible from a third one.

Nonetheless, our formalization also includes some intermediate results that are well-known for all normal modal logics, and that are commonly used when dealing with formal proofs in Hilbert-style systems. Finally, we formalized the equivalence between two of the most used axiomatizations of S4, the one presented in [15] which is commonly used when dealing with the relational semantics [5], and the one introduced by McKinsey and Tarski for the topological semantics [1].

Related work

Our ground base is the Isabelle/HOL theory `EpistemicLogic.thy` [14], which contains not only the formalization of other epistemic logics such as S4 and S5, but also formalizes the definition of an abstract canonical model, as well as a simple and convenient way to work with any desired normal modal logic by adding necessary the axioms to the basic system K [15]. A related paper to this formalization is [28], which contains a broad and updated summary on formalizations of logical systems and correspondent important results using different theorem provers. Other ways to formalize logical systems and their completeness results on Isabelle have also been studied in [10] and [6], which include (but are not limited to) the use of natural deduction rules, sequent-style rules, and tableau rules for the formal systems, and coinductive methods for soundness and completeness results.

However, given the already existent formalization of LTL in Isabelle/HOL [24], as well as the prevalence of Isabelle as a tool for formal verification of safety requirements for critical systems, it becomes important to provide this formalization for this particular proof assistant. In addition to this, in [19] the authors defined and investigated the notion of KB_R-structures, which are used to represent a description of the epistemic status of a rational agent that is not necessarily aware of their ignorance, and provided a result that matches them with models of the epistemic logic S4.2. Modal logics between S4.2 and S5 are of special interest for applications in epistemic logic, since they allow formalizations of several degrees of ignorance for each one of the agents [22].

The paper is organized as follows: Section 2 introduces the necessary background on epistemic logic, including its relational and topological semantics, and how the syntax and the relational semantics were formalized in Isabelle/HOL in [14]. Section 3 explains our formalization of Stalnaker's epistemic logic, including the intermediate results necessary to prove the main results, and the limitations of these to only countably many agents. Section 4 explains our formalization of the equivalence between the two most common axiomatizations for S4, the one that arises from the topological semantics, and the one commonly used when working with the relational semantics. Finally, in Section 5, we conclude with a discussion about the results, limitations, and future work.

2 Background

2.1 Stalnaker's Epistemic logic

We briefly present the axiomatic system developed by R. Stalnaker for both notions of knowledge and belief, as well as the main result for the "knowledge formulas" (i.e., for those formulas that do not contain any belief modal operators), which correspond to the multimodal system S4.2 [25]. We omit the proof for this result, as the Isabelle theory "Epistemic logic: Completeness of Modal Logics" [14] does not support belief formulas.

Consider the well-formed formulas obtained from the following grammar, where x ranges over the set of propositional symbols and i ranges over the set of agent labels:

$$\phi, \psi ::= \bot \,|\, x \,|\, \phi \vee \psi \,|\, \phi \wedge \psi \,|\, \phi \to \psi \,|\, K_i\phi \,|\, B_i\phi.$$

The operators K_i and B_i mean "agent i knows" and "agent i believes," respectively. Although Stalnaker does not present his logic of knowledge and belief using this exact set of propositional connectives, but a proper subset of these, we added the remaining ones given that From's formalization includes all of them [15].

Stalnaker's principles (axioms) for knowledge and belief appear in Table 1, along with their interpretations in natural language. *Stalnaker's logic for knowledge and belief* corresponds to the formal system obtained by adding these axioms to the axioms and rules of the multi-modal logic S4, that is, the smallest logic containing the following axioms:

- all propositional tautologies,
- axiom K: $(K_i(\phi \to \psi) \wedge K_i\phi) \to K_i\psi$,
- axiom T: $K_i\phi \to \phi$, and
- axiom 4: $K_i\phi \to K_iK_i\phi$;

and that is closed under Modus Ponens and the Necessitation rule, "from ϕ infer $K_i\phi$", where i ranges over the set of agents.

Table 1: Axioms for knowledge and belief.

$B_i\phi \to K_iB_i\phi$	Positive introspection
$\neg B_i\phi \to K_i\neg B_i\phi$	Negative introspection
$K_i\phi \to B_i\phi$	Knowledge implies belief
$B_i\phi \to \neg B_i\neg\phi$	Consistency of belief
$B_i\phi \to B_iK_i\phi$	Strong belief

The following proposition summarizes some relevant properties of this logic.

Proposition 1. *The following are some key properties of Stalnaker's logic for knowledge and belief [25].*

1. *The following equivalences, one for each agent label i and formula ϕ, are theorems in this logic:*

$$B_i\phi \longleftrightarrow \neg K_i\neg K_i\phi.$$

2. *As a consequence of the previous property, by replacing 'B_i' with '$\neg K_i\neg K_i$' in the* Consistency of belief *axiom, we get that $\neg K_i\neg K_i\phi \to K_i\neg K_i\neg\phi$ (also known as axiom .2) is a theorem in this logic. This implies that the knowledge formulas of this logic correspond exactly to the logic given by the system S4.2, i.e., those that can be obtained from the rules and axioms of the multi-modal logic S4 in presence of the axiom .2.*

The above proposition allows us to interpret this logic by giving a semantics only for the propositional variables, Boolean connectives and knowledge operators. Formally, we use structures $\mathfrak{M} = (\mathscr{F}, \pi)$ known as Kripke models, where the frame $\mathscr{F} = (W, (R_i)_i)$ is a pair consisting of a non-empty set of worlds W, a set of binary accessibility relations $R_i \subseteq W \times W$, one for each agent i, and $\pi : \mathrm{Var} \to 2^W$ is a valuation of propositional symbols. Formula satisfiability at a given world $w \in W$ is defined as follows:

$\mathfrak{M}, w \not\models \bot$		
$\mathfrak{M}, w \models x$	iff	$w \in \pi(x)$
$\mathfrak{M}, w \models \phi \vee \psi$	iff	$\mathfrak{M}, w \models \phi$ or $\mathfrak{M}, w \models \psi$
$\mathfrak{M}, w \models \phi \wedge \psi$	iff	$\mathfrak{M}, w \models \phi$ and $\mathfrak{M}, w \models \psi$
$\mathfrak{M}, w \models \phi \to \psi$	iff	$\mathfrak{M}, w \not\models \phi$ or $\mathfrak{M}, w \models \psi$
$\mathfrak{M}, w \models K_i\phi$	iff	$\forall v \in W\,(wR_iv \to \mathfrak{M}, v \models \phi)$
$\mathfrak{M}, w \models B_i\phi$	iff	$\mathfrak{M}, w \models \neg K_i\neg K_i\phi$.

One can use functions $\mathscr{K} : W \to 2^W$ instead of sets of ordered pairs $R \subseteq W \times W$, as there is a correspondence between these objects by setting

$$wRv \iff v \in \mathscr{K}(w),$$

for all $w, v \in W$.

2.2 Epistemic Logic: Completeness of Modal Logics

The "Epistemic Logic: Completeness of Modal Logics" entry on Isabelle's AFP [14] contains not only a formalization for the completeness results for some epistemic logics, but also a formalization of the general strategy for Henkin-style proofs for completeness. This is what enabled us to formalize a proof for the completeness result for S4.2. We show here the formalization of the Kripke models from [14], which are structures consisting of a set of worlds of type $'w$, a truth assignment for each propositional variable on each world given by the function denoted π, and a set of accessible worlds from each possible world for each agent $'i$.

```
datatype ('i, 'w) kripke =
  Kripke (𝒲: ‹'w set›) (π: ‹'w ⇒ id ⇒ bool›) (𝒦: ‹'i ⇒ 'w ⇒ 'w set›)
```

Consequently, given a Kripke model $\mathfrak{M} = (W, (\mathscr{K}_i)_{i\in I}, \pi)$ with accessibility functions $\mathscr{K}_i$ for each agent i, formula satisfiability is defined by setting

$$\mathfrak{M}, w \models K_i\phi \quad \text{iff} \quad \forall v \in \mathscr{K}_i(w)\,(\mathfrak{M}, v \models \phi).$$

Additionally, the *dual* operator for each knowledge operator K_i is denoted in this formalization as L_i and is defined as a short hand for "agent i does not know if something is false." In other words, $L_i\phi := \neg K_i(\neg\phi)$, for all formulas ϕ. The Kripke semantics for this operator corresponds to [5, 8]

$$\mathfrak{M}, w \models L_i\phi \quad \text{iff} \quad \exists v \in \mathscr{K}_i(w)\,(\mathfrak{M}, v \models \phi).$$

We show here the corresponding formalization presented in [15] for the Kripke semantics, which is defined inductively on formulas for each world.

```
primrec semantics :: ‹('i, 'w) kripke ⇒ 'w ⇒ 'i fm ⇒ bool›
  (‹_, _ ⊨ _› [50, 50] 50) where
  ‹(M, w ⊨ ⊥) = False›
| ‹(M, w ⊨ Pro x) = π M w x›
| ‹(M, w ⊨ (p ∨ q)) = ((M, w ⊨ p) ∨ (M, w ⊨ q))›
| ‹(M, w ⊨ (p ∧ q)) = ((M, w ⊨ p) ∧ (M, w ⊨ q))›
| ‹(M, w ⊨ (p ⟶ q)) = ((M, w ⊨ p) ⟶ (M, w ⊨ q))›
| ‹(M, w ⊨ K i p) = (∀v ∈ 𝒲 M ∩ 𝒦 M i w. M, v ⊨ p)›
```

From's formalization then focuses on proving the soundness and completeness results for each of the most commonly found *normal modal logics* in the literature concerning certain *classes of frames* [5, 8, 15]. We now summarize the relevant ones for our formalization.

1. The basic logic, K, whose corresponding axiomatic system consists of all propositional tautologies and the axiom K, and is closed under Modus Ponens and the Necessitation Rule, is sound and complete with respect to the class of all frames.
2. The logic S4 is sound and complete with respect to the class of all transitive and reflexive frames.

Notice that From's formalization does not include modal operators for belief, this restricts us to the knowledge fragment of the language. However, Proposition 1 tells us that belief is equivalent to knowledge, thus we do not lose any information by restricting to the knowledge fragment.

2.3 Topological semantics and its axioms

The topological semantics for modal logics was introduced before the relational semantics that presently dominate the field [1], and the first semantics completeness proof for S4 was derives from there. Recall the notion of this topological semantics for a language with a single modal operator $\Box$. Let $\mathscr{L}$ be the language composed of all formulas given by the following grammar:

$$\phi, \psi ::= x \,|\, \neg\phi \,|\, \phi \wedge \psi \,|\, \phi \vee \psi \,|\, \phi \to \psi \,|\, \Box\phi,$$

where x ranges over the set of propositional symbols Var. Formulas in $\mathscr{L}$ are interpreted in a topological model $M = \langle W, \tau, v \rangle$, consisting of a non-empty set W, a topology τ over W, and a valuation $v : \mathrm{Var} \to 2^W$ in the following way:

- $M, w \models x$ iff $x \in v(x)$;
- $M, w \models \neg\phi$ iff $M, x \not\models \phi$;
- $M, w \models \phi \wedge \psi$ iff $M, x \models \phi$ and $M, x \models \psi$;
- $M, w \models \phi \vee \psi$ iff $M, x \models \phi$ or $M, x \models \psi$;
- $M, w \models \phi \to \psi$ iff $M, x \not\models \phi$ or $M, x \models \psi$;
- $M, w \models \Box\phi$ iff there exists $U \in \tau$ such that $w \in U$ and $M, y \models \phi$ for all $y \in U$.

Although there is nothing inherently wrong with using the deductive system presented in [15] for the logic S4, the following axiomatization is often preferable when working with the topological semantics, as the meaning of the axioms and rules under this semantics resembles some well-known properties of topological spaces [1].

Table 2: Topological S4 axioms and rules.

Axiom	Formula	Rule	Formula
N	$\Box\top$	MP	$\dfrac{\phi \to \psi \quad \phi}{\psi}$
R	$\Box(\phi \wedge \psi) \leftrightarrow (\Box\phi \wedge \Box\psi)$		
T	$\Box\phi \to \psi$	M	$\dfrac{\phi \to \psi}{\Box\phi \to \Box\psi}$
4	$\Box\phi \to \Box\Box\phi$		

Notice that at first it is not obvious weather or not the logic obtained from this axiomatization is a normal modal logic, often defined as a logic that *extends* system K [5], as neither axiom K nor the Necessitation Rule are present in the list of axioms and rules. However, we formalized a proof for the equivalence between both axiomatizations in the context of a multi-agent epistemic logic, as in recent years several authors have been developing topological semantics for notions of knowledge and belief [2, 4, 3], where this result is often briefly mentioned and applied, but not proved in detail. Nonetheless, it is also worth noticing here that the relational semantics of S4 is no more than a particular case for the topological semantics, as one can assign a binary relation to each topological space by defining what is known as the *specialization order* [1].

3 Formalization

We now consider the epistemic logic based on the axioms in Table 1 and the results in Proposition 1 for the knowledge fragment of the language. We prove the soundness and completeness of the pure epistemic logic obtained from this system with respect to all frames consisting of weakly directed pre-orders by combining and applying the results formalized in [15] with some auxiliary results provided in the *Utility* section of our Isabelle theory. This allows us to utilize the canonical model strategy to prove completeness of the obtained system. We do not formalize a logic for both knowledge and belief, since we aimed to work on top of the formalization in [14], which considers modalities only for knowledge. Formalizing the whole logic for both knowledge and belief will then require developing a new theory almost from scratch that includes modalities for both notions.

In order to do this, we prove first some intermediate results towards the completeness of the system obtained by adding axiom .2 to the system K, also known as system G in the literature [8], including some results about the underlying propositional logic. This system is known to be complete with respect to the class of weakly directed frames, and, although we do not formalize this result completely, we do formalize a version of the Truth lemma for this system, which is needed so that we can combine it with the results for system S4 formalized in [15] to achieve our goal of formalizing the completeness result for the logic S4.2 with respect to all weakly directed pre-orders.

3.1 Rewriting propositional and modal formulas

In the deductive system formalized in [14], deduction from a set of premises is defined as follows: given a set of formulas $\Gamma \cup \{\phi\}$, we say that "ϕ is derived from Γ" (denoted $\Gamma \vdash \phi$) if there are formulas $\psi_1, \ldots, \psi_k$ in Γ such that the formula $\psi_1 \to (\psi_2 \to \ldots (\psi_k \to \phi))$ is a theorem in the system, where k is a non-negative integer. It is well-known that this formula is logically equivalent to $(\psi_1 \wedge \ldots \wedge \psi_k) \to \phi$ in classical propositional logic, thus the notion can be defined by requiring the latter to be a theorem in the system instead. Being able to translate between these two equivalent formulas in our formal deductive system plays an important role for the proof of our main result, thus we provided a formalization of several results of this kind in the *Utility* section of our Isabelle theory, which includes some results that were not used later but that might become handy for the development of the formalizations of other related theories in the future.

Similarly to the **imply** function in [14], which produces, from a list of formulas $[\psi_1, \ldots, \psi_k]$ and a formula ϕ, the formula $\psi_1 \to (\psi_2 \to \ldots (\psi_k \to \phi))$, we introduce the function **conjunct**, which takes a list of formulas $[\psi_1, \ldots, \psi_k]$ and produces the formula $\psi_1 \wedge \ldots \wedge \psi_k \wedge \top$. (Notice that the input may be an empty list, in which case the output is $\top$.) We formalized some results about logical equivalences, and derived rules and maximal consistent sets regarding the **imply** and **conjunct** functions that are well-known for the logic K, some of which are presented in the following lemmas. These are required to prove in section 3.2 that the axiom .2 induces the weakly directed property on all frames that satisfy it, following [26]. We include the proofs for those lemmas that require elaborated arguments.

Lemma 2 (Derived rules)**.** *For all formulas $\psi_1, \ldots, \psi_k, \phi$, it is the case that $\vdash (\psi_1 \wedge \ldots \wedge \psi_k) \to \phi$ if and only if $\vdash \psi_1 \to (\psi_2 \to \ldots (\psi_k \to \phi))$.*

Lemma 3 (Logical equivalences)**.** *The following two lemmas capture the fact that in system K, hence in any normal modal logic, the formulas $(K_i\psi_1 \wedge \ldots \wedge K_i\psi_k)$ and $K_i(\psi_1 \wedge \ldots \wedge \psi_k)$ are equivalent, for any finite set of formulas $\psi_1, \ldots, \psi_k$ and any agent i.*

Lemma 4 (Closure under conjunctions for MCSs)**.** *The following lemma proves that maximal consistent sets of formulas are closed under conjunctions, that is, if Γ is a maximal consistent set of formulas and $\psi_1, \ldots, \psi_k$ are some formulas in Γ, then $\psi_1 \wedge \ldots \wedge \psi_k \in \Gamma$.*

Lemma 5. *For all formulas ϕ, ψ, θ, it is the case that*

$$\vdash ((K_i\phi \wedge K_i\psi) \to \theta) \to (K_i(\phi \wedge \psi) \to \theta)$$

for any agent label i, as long as the type of i is countable.

```
lemma K-conj-imply-factor:
  fixes A :: ‹(('i :: countable) fm ⇒ bool)›
  shows ‹A ⊢ ((((K i p) ∧ (K i q)) ⟶ r) ⟶((K i (p ∧ q)) ⟶ r))›
```

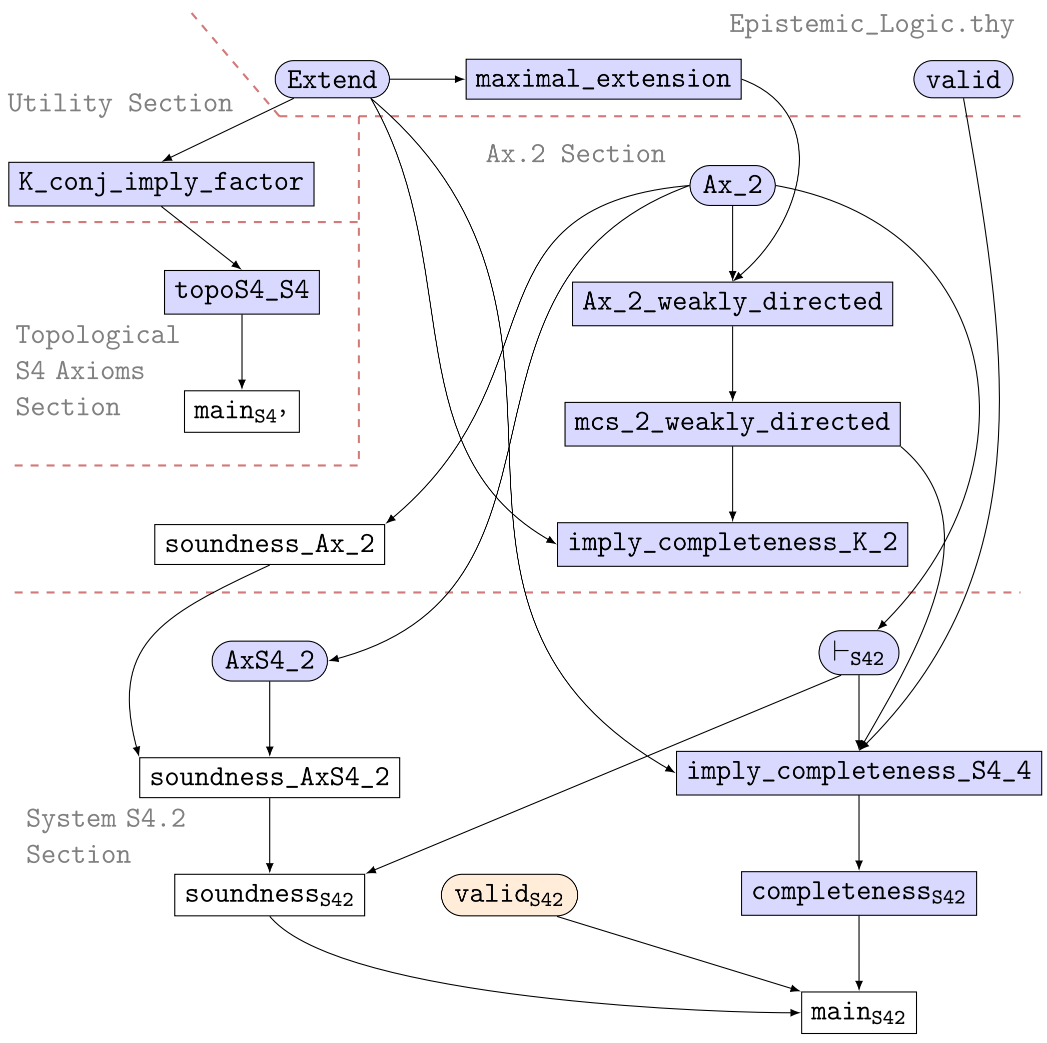


Figure 1: Dependency graph showing the main results and the definitions, abbreviations, and intermediate results from their proofs that require the countable type condition. The dotted lines and the gray text show the files or sections of the Isabelle theory corresponding to our formalization where these can be found. Definitions and abbreviations appear in rounded rectangles, whereas lemmas and theorems appear in rectangles. Those that explicitly mention the countability condition are colored in blue, and the color orange means that this result is applied using the set of natural numbers to label the agents.

The assumption over the set of agent labels for the previous lemma is imposed by the proof that was formalized for it, as it relies on the proof for the completeness of K in [14], which requires this condition, as depicted in Figure 1.

Additionally, we formalize the following lemma, which plays a significant role in the proof of the completeness result for Stalnaker's epistemic logic that follows [26].

Lemma 6. *Given any pair of formulas ϕ, ψ, $(K_i\phi \wedge L_i\psi) \rightarrow L_i(\phi \wedge \psi)$ is a theorem in system K, for every agent label i.*

Proof. Notice that, for any formulas ϕ and ψ, $\vdash \phi \to (\neg(\phi \wedge \psi) \to \neg\psi)$, hence

$$\vdash K_i\phi \to K_i(\neg(\phi \wedge \psi) \to \neg\psi).$$

On the other hand, we have that $\vdash K_i(\neg(\phi \wedge \psi) \to \neg\psi) \to ((K_i\neg(\phi \wedge \psi)) \to K_i\neg\psi)$, thus

$$\vdash K_i\phi \to ((K_i\neg(\phi \wedge \psi)) \to K_i\neg\psi).$$

This implies that $\vdash K_i\phi \to (L_i\psi \to L_i(\phi \wedge \psi))$, which is equivalent to

$$\vdash (K_i\phi \wedge L_i\psi) \to L_i(\phi \wedge \psi).$$

□

3.2 Axiom .2

We formalize axiom schema .2, which when added to the axioms and rules of system K imposes a structure on the canonical model, namely, we obtain a weakly directed frame. For this, the **inductive** command lets us define the axiom schema in such a way that *i* and *p* can be instantiated at will, as long as the type for the agents labels is countable.

inductive *Ax-2* :: ‹(*'i* :: *countable*) *fm* ⇒ *bool*› **where**
 ‹*Ax-2* (¬ *K i* (¬ *K i p*) ⟶ *K i* (¬ *K i* (¬ *p*)))›

A frame $\mathscr{F} = (W, (R)_{i\in I})$ is said to be *weakly directed* if whenever vR_iw and vR_iu, there exists $x \in W$ such that wR_ix and uR_ix, for each $i \in I$. Accordingly, we formalize this property for Kripke frames as follows:

definition *weakly-directed* :: ‹(*'i*, *'s*) *kripke* ⇒ *bool*› **where**
 ‹*weakly-directed M* ≡ ∀ *i*. ∀ *s* ∈ 𝒲 *M*. ∀ *t* ∈ 𝒲 *M*. ∀ *r* ∈ 𝒲 *M*.
  (*r* ∈ 𝒦 *M i s* ∧ *t* ∈ 𝒦 *M i s*)⟶(∃ *u* ∈ 𝒲 *M*. (*u* ∈ 𝒦 *M i r* ∧ *u* ∈ 𝒦 *M i t*))›

The soundness of axiom schema .2 with respect to weakly directed frames is formalized in our Isabelle theory, and it follows from the definitions for the semantics and the weakly directed property. However, proving that the property holds for the canonical model when adding the axiom to a normal modal logic is non-trivial. Unlike the frame properties imposed by the axioms considered in the Epistemic Logics formalized in [15], which are all universal properties, this property is universal-existential, so to prove that the canonical model has this property means that one has to show the existence of a possible world satisfying a property under some assumptions.

Recall that the *canonical frame*, $\mathscr{F}^{\text{can}} = (W^{\text{can}}, (R_i^{\text{can}})_{i\in I})$, consists of the set all maximal consistent sets of formulas (with respect to K+.2) as the set of possible worlds, W^{can}, and the accessibility relations R_i^{can} are defined as follows:

$$\Gamma R_i^{\text{can}} \Delta \quad \text{iff} \quad \{\phi : K_i\phi \in \Gamma\} \subseteq \Delta,$$

for each agent *i*. Thus, showing that the canonical frame for a system including axiom .2 is weakly directed involves verifying that if we have $\{\phi : K_i\phi \in \Gamma\} \subseteq \Delta_1$ and $\{\phi : K_i\phi \in \Gamma\} \subseteq \Delta_2$ for some maximal consistent sets Γ, Δ_1 and Δ_2, then there exists a maximal consistent set Θ such that $\{\phi : K_i\phi \in \Delta_1\} \cup \{\phi : K_i\phi \in \Delta_2\} \subseteq \Theta$. We capture this in our formalization by the following lemma.

Lemma 7 (Weakly directed property and the axiom .2)**.** *Suppose that V, U, W are three maximal consistent sets with respect to a normal modal logic containing the axiom .2. If $VR_i^{\text{can}}U$ and $VR_i^{\text{can}}W$, then there exists a maximal consistent set X such that $UR_i^{\text{can}}X$ and $WR_i^{\text{can}}X$.*

Proof. First, fix a set of formulas A and three maximal consistent sets of formulas V, U, W (with respect to A) satisfying the lemma assumptions for some agent label i of a countable type. Assume towards contradiction that such a set X does not exist, then

$$S := \{\phi : K_i\phi \in W\} \cup \{\phi : K_i\phi \in U\}$$

has to be inconsistent with respect to A, hence there are formulas $\theta_1, \ldots, \theta_k \in \{\phi : K_i\phi \in U\}$ and $\psi_1, \ldots, \psi_m \in \{\phi : K_i\phi \in W\}$, for some $k, m \in \mathbb{N}$, such that

$$A \vdash (\alpha \wedge \beta) \to \perp,$$

where $\alpha = \theta_1 \wedge \ldots \wedge \theta_k$ and $\beta = \psi_1 \wedge \ldots \wedge \psi_m$. This implies that $A \vdash K_iK_i(\neg(\alpha \wedge \beta))$, since $\phi \to \perp$ is equivalent to $\neg\phi$ for every formula ϕ, by applying the Necessitation rule twice. By definition, we have that $K_i\theta_1, \ldots, K_i\theta_k \in U$ and $K_i\psi_1, \ldots, K_i\psi_m \in W$, thus

$$K_i\theta_1 \wedge \ldots \wedge K_i\theta_k \in U \quad \text{and} \quad K_i\psi_1 \wedge \ldots \wedge K_i\psi_m \in W,$$

since these sets are closed under logical consequences. We then use the logical equivalences and properties for maximal consistent sets from the *Utility section* (see section 3.1) to obtain that $K_i\alpha \in U$ and $K_i\beta \in W$. This implies that the formulas $L_iK_i\alpha$ and $L_iK_i\beta$ are elements of V, and so is the formula $K_iL_i\alpha$, since V contains every instance of axiom .2 and is closed under logical consequences. This implies that $(L_iK_i\beta) \wedge (K_iL_i\alpha) \in V$, thus $L_i(K_i\beta \wedge L_i\alpha) \in V$, so there exists a maximal consistent set Z such that

$$VR_i^{\text{can}}Z \qquad \text{and} \qquad K_i\beta \wedge L_i\alpha \in Z.$$

Applying the lemma *K-thm* we get that $L_i(\beta \wedge \alpha) \in Z$, but notice that $K_i\neg(\alpha \wedge \beta) \in Z$, thus $K_i\neg(\beta \wedge \alpha) \in Z$, which is a contradiction because we have found a formula ϕ such that $\phi, \neg\phi \in Z$. □

Note that we have restricted ourselves to countable types for the set of agent labels in formalization of the previous two lemmas, as we are only allowed to extend a consistent set into a maximal one when the language is countable, because of a dependency shown in Figure 1. Unlike in the respective result for each normal modal logic formalized in [15], this restriction to a countable type was necessary as we were dealing with a universal-existence property and not with a purely universal one. We then prove a version of the Truth lemma for the minimal normal modal logic that includes axiom .2, which is the relevant result about this system that will allow us to prove the completeness result for system S4.2 in the next section.

Lemma 8 (Imply completeness for Axiom .2)**.** *Let $\Gamma \cup \{\phi\}$ be a set of formulas. Suppose that, for all weakly directed Kripke structures M, $M, w \models \Gamma$ implies $M, w \models \phi$, for each world $w \in M$. Then there are formulas $\gamma_1, \gamma_2, \ldots, \gamma_m \in \Gamma$ such that*

$$\vdash_{.2} \gamma_1 \to (\gamma_2 \to \ldots (\gamma_m \to \phi) \ldots).$$

We omit the proof for this lemma, since it follows the same strategy as the correspondent ones for the systems formalized in [14].

3.3 System S4.2

We define system S4.2 as the one obtained by adding to system K the axioms T, 4 and .2, by making use of the abbreviation $\oplus$ introduced in [14] that allows combining axiom predicates:

abbreviation *SystemS4-2* :: ‹(*'i* :: *countable*) *fm* ⇒ *bool*› ($\vdash_{S42}$ - [*50*] *50*) **where**
  ‹$\vdash_{S42}$ *p* ≡ *AxT* ⊕ *Ax4* ⊕ *Ax-2* ⊢ *p*›

Recall that axioms T and 4 impose reflexivity and transitivity on the canonical frame, respectively [5], which was formalized in [14]. This implies that the composition of these two with axiom .2 imposes all three conditions on the canonical frame, which leads to the soundness and completeness of S4.2 with respect to all weakly directed pre-orders. To prove the completeness result, we prove first the analog results to Lemmas 7 and 8 but for S4.2 and Kripke models based on weakly directed preorders.

Lemma 9 (S4.2 and Weakly directed preorders). *Let $\Gamma \cup \{\phi\}$ be a set of formulas. Suppose that, for all countable Kripke structures M based on weakly directed preorders, $M, w \models \Gamma$ implies $M, w \models \phi$, for each world $w \in M$. Then, there are formulas $\gamma_1, \ldots, \gamma_m \in \Gamma$ such that*

$$\vdash_{S42} \gamma_1 \to (\ldots \to (\gamma_m \to \phi)\ldots).$$

lemma *imply-completeness-S4-2*:
  assumes *valid*: ‹∀ (*M* :: (*'i* :: *countable*, *'i fm set*) *kripke*). ∀ *w* ∈ 𝒲 *M*.
   w-directed-preorder M ⟶ (∀ *q* ∈ *G*. *M*, *w* ⊨ *q*) ⟶ *M*, *w* ⊨ *p*›
  shows ‹∃ *qs*. *set qs* ⊆ *G* ∧ (*AxS4-2* ⊢ *imply qs p*)›

This implies that if a formula is valid in all countable Kripke structures based on weakly directed preorders, then it is a theorem in S4.2.

lemma *completeness*$_{S42}$:
  assumes ‹∀ (*M* :: (*'i* :: *countable*, *'i fm set*) *kripke*). ∀ *w* ∈ 𝒲 *M*. *w-directed-preorder M* ⟶ *M*, *w* ⊨ *p*›
  shows ‹$\vdash_{S42}$ *p*›

Our main result follows: the completeness of S4.2 with respect to all frames consisting of weakly directed pre-orders.

Theorem 10 (Completeness of S4.2). *A formula is valid in all countable Kripke structures based on weakly directed preorders if and only if it is a theorem in S4.2.*

theorem *main*$_{S42}$: ‹*valid*$_{S42}$ *p* ⟷ $\vdash_{S42}$ *p*›

4 An alternative axiomatization for System S4

Inspired by the last section of [14], we formalize an alternative axiomatization for System S4 that is often used when dealing with the *topological semantics* [1] for modal operators and we show its equivalence to the one considered in [15]. We formalize the system corresponding to the axioms and rules in Table 2 in Isabelle, which we call topoS4, and, if a formula ϕ is a theorem in this system, we denote this by $\vdash_{Top} \phi$.

inductive *System-topoS4* :: ‹*'i fm* ⇒ *bool*› ($\vdash_{Top}$ - [*50*] *50*) **where**
  A1': ‹*tautology p* ⟹ $\vdash_{Top}$ *p*›
  | *AR*: ‹$\vdash_{Top}$ ((*K i* (*p* ∧ *q*)) ⟷ ((*K i p*) ∧ *K i q*))›
  | *AT'*: ‹$\vdash_{Top}$ (*K i p* ⟶ *p*)›
  | *A4'*: ‹$\vdash_{Top}$ (*K i p* ⟶ *K i* (*K i p*))›
  | *AN*: ‹$\vdash_{Top}$ *K i* ⊤›
  | *R1'*: ‹$\vdash_{Top}$ *p* ⟹ $\vdash_{Top}$ (*p* ⟶ *q*) ⟹ $\vdash_{Top}$ *q*›
  | *RM*: ‹$\vdash_{Top}$ (*p* ⟶ *q*) ⟹ $\vdash_{Top}$ ((*K i p*) ⟶ *K i q*)›

To show that this formulation is equivalent to the one in [15] (which is based on [5]), we provide derivations of axiom K and the Necessitation Rule (from ϕ deduct $\Box\phi$). This is enough as our system already includes axioms T and 4 in the same fashion as in [15], and is based on the same propositional logic.

Lemma 11. *For all formulas ϕ and ψ, $\vdash_{top} (K_i\phi \wedge K_i(\phi \to \psi)) \to K_i\psi$ and, if $\vdash_{top} \phi$, then $\vdash_{top} K_i\phi$.*

Proof. For the first part, notice that $\vdash_{top} (\phi \wedge (\phi \to \psi)) \to \psi$, since it is an instance of a propositional tautology. Then, we apply RM to obtain that $\vdash_{top} K_i(\phi \wedge (\phi \to \psi)) \to K_i\psi$, which implies that $\vdash_{top} (K_i\phi \wedge K_i(\phi \to \psi)) \to K_i\psi$. For the second one, suppose that $\vdash_{top} \phi$ and notice that $\vdash_{top} \phi \to (\top \to \phi)$, hence $\vdash_{top} \top \to \phi$. Applying RM we get that $\vdash_{top} K_i\top \to K_i\phi$, thus $\vdash_{top} K_i\phi$. □

From this it then follows that any formula derivable in the classical S4 system (denoted $\vdash_{S4}$) can be derived in our system as well.

Lemma 12. *All S4 theorems are theorems in topoS4.*

lemma *S4-topoS4*: ‹$\vdash_{S4}$ *p* $\Longrightarrow$ $\vdash_{Top}$ *p*›

The converse follows by a similar argument, we show that axioms and rules from our system are all derivable in $\vdash_{S4}$, under the condition that there are only countably many agents.

Lemma 13. *All theorems in topoS4 are theorems in S4, assuming that there are only countably many agents.*

lemma *topoS4-S4*:
  fixes *p* :: ‹(*'i* :: *countable*) *fm*›
  shows ‹$\vdash_{Top}$ *p* $\Longrightarrow$ $\vdash_{S4}$ *p*›

By combining the last two results with the main result for S4 in [15], we obtain formalized soundness and completeness for this alternative axiomatization of S4 over the class of S4 frames, namely, all reflexive and transitive frames.

Theorem 14 (Soundness and Completeness of topoS4)**.** *A formula is valid in all S4 Kripke models if and only if it is a theorem in topoS4.*

theorem $main_{S4}'$: ‹$valid_{S4}$ *p* $\longleftrightarrow$ ($\vdash_{Top}$ *p*)›

5 Results, Discussion, and Future work

We have formalized the soundness and completeness for Stalnaker's Epistemic Logic S4.2 with respect to the class of Kripke frames consisting of weakly-directed pre-orders for countably many agents, which has not been formalized before neither in Isabelle, nor in any other publicly available proof assistant. Additionally, the equivalence between the topological axiomatization of S4 and the one in [14] is also described in this document. The proofs for the main result, as well as for many of the intermediate results, have been sketched before in multiple sources, but we were not able to find a unique source, making this the first work of its kind. Additionally, given the recent interest in applications of the topological semantics for epistemic modal operators [2, 4, 3], some of which coincide with Stalnaker's epistemic logic, this provides a reinforcement for the foundations of these works.

We emphasize on the assumption of the cardinality of the set of agent labels, as it was necessary to impose such restriction even in some definitions in our formalization, thus creating a discrepancy with the definitions commonly found in the literature. We present in Figure 1 a summary of the definitions and results of our formalization and the one in [14] that rely on this condition, since the formalization for the general strategy applied to obtain the completeness results in [14] requires it to be able to obtain maximal consistent sets. Although, in theory, it is possible to provide an argument for the general case using Zorn's lemma (this was also later noted in [15]), which is available in [13].

Further work in formalizing in Isabelle/HOL of different formal aspects of modal logics that include S4 operators, as is the case with many temporal logics like LTL with its *always* operator, for which a complete axiomatization is already known [18]; as well as concrete examples of epistemic scenarios based on Stalnaker's principles, like the example detailed in [20]. We hope that this work will facilitate further work in formalizing different logical systems in Isabelle/HOL.

References

[1] Marco Aiello, Johan van Benthem & Guram Bezhanishvili (2003): *Reasoning About Space: The Modal Way*. *Journal of Logic and Computation* 13(6), pp. 889–920, doi:10.1093/logcom/13.6.889. arXiv:https://academic.oup.com/logcom/article-pdf/13/6/889/2936128/130889.pdf.

[2] Alexandru Baltag, Nick Bezhanishvili & Saúl Fernández González (2022): *Topological Evidence Logics: Multi-agent Setting*. In Aybüke Özgün & Yulia Zinova, editors: *Language, Logic, and Computation*, Springer International Publishing, Cham, pp. 237–257, doi:10.1007/978-3-030-98479-3_12.

[3] Alexandru Baltag, Nick Bezhanishvili, Aybüke Özgün & Sonja Smets (2013): *The Topology of Belief, Belief Revision and Defeasible Knowledge*. In Davide Grossi, Olivier Roy & Huaxin Huang, editors: *Logic, Rationality, and Interaction*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 27–40, doi:10.1007/978-3-642-40948-6_3.

[4] Alexandru Baltag, Nick Bezhanishvili, Aybüke Özgün & Sonja Smets (2016): *Justified Belief and the Topology of Evidence*. In Jouko Väänänen, Åsa Hirvonen & Ruy de Queiroz, editors: *Logic, Language, Information, and Computation*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 83–103, doi:10.1007/978-3-662-52921-8_6.

[5] Robert Blackburn, Maarten de Rijke & Yde Venema (2001): *Modal Logic*. *Modal Logic*, doi:10.1017/CBO9781107050884.

[6] Jasmin Christian Blanchette, Andrei Popescu & Dmitriy Traytel (2017): *Soundness and Completeness Proofs by Coinductive Methods*. *Journal of Automated Reasoning* 58(1), pp. 149 – 179, doi:10.1007/s10817-016-9391-3. Available at `https://hal.inria.fr/hal-01643157`.

[7] Marco Bozzano, Alessandro Cimatti, Marco Gario & Stefano Tonetta (2014): *Formal design of fault detection and identification components using temporal epistemic logic*. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 326–340, doi:10.1007/978-3-642-54862-8_22.

[8] Alexander Chagrov (1997): *Modal Logic*. Oxford logic guides, Clarendon Press, doi:10.1093/oso/9780198537793.001.0001. Available at `https://books.google.com/books?id=dhgi5NF4RtcC`.

[9] Alessandro Cimatti, Marco Gario & Stefano Tonetta (2016): *A Lazy Approach to Temporal Epistemic Logic Model Checking*. In: *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, AAMAS '16, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, p. 1218–1226.

[10] F Miguel Dionísio, Paula Gouveia & Joao Marcos (2005): *Defining and using deductive systems with Isabelle*. *Computing, Philosophy, and Cognition*, pp. 271–293. Available at `http://temporallogic.org/courses/AppliedFormalMethods/DefiningAndUsingDeductiveSystemsWithIsabelle.pdf`.

[11] Hans van Ditmarsch, Wieve van der Hoek & Barteld Kooi (2008): *Dynamic Epistemic Logic*. Springer Netherlands, doi:10.1007/978-1-4020-5839-4.

[12] Ronald Fagin, Joseph Y. Halpern, Yoram Moses & Moshe Vardi (1995): *Reasoning About Knowledge*. MIT Press, London, England.

[13] Jacques D. Fleuriot, Tobias Nipkow & Christian Sternagel: *Zorn's Lemma (ported from Larry Paulson's Zorn.thy in ZF)*. Available at `https://isabelle.in.tum.de/dist/library/HOL/HOL/Zorn.html`.

[14] Asta Halkjær From (2018): *Epistemic Logic: Completeness of Modal Logics*. Archive of Formal Proofs. `https://isa-afp.org/entries/Epistemic_Logic.html`, Formal proof development.

[15] Asta Halkjær From (2021): *Formalized soundness and completeness of epistemic logic*. In: International Workshop on Logic, Language, Information, and Computation, Springer, pp. 1–15, doi:10.1007/978-3-030-88853-4_1.

[16] Laura P. Gamboa Guzman (2021): *Dynamical operators on models with evidence*. Master's thesis, Universidad de los Andes, Bogota, Colombia. Available at `https://repositorio.uniandes.edu.co/handle/1992/55112`.

[17] Laura P. Gamboa Guzman (2022): *Stalnaker's Epistemic Logic*. Archive of Formal Proofs. `https://isa-afp.org/entries/Stalnaker_Logic.html`, Formal proof development.

[18] Robert Goldblatt (1992): *Logics of Time and Computation*. CSLI Publications.

[19] Costas D. Koutras & Yorgos Zikos (2011): *Relating Truth, Knowledge and Belief in Epistemic States*. In Weiru Liu, editor: Symbolic and Quantitative Approaches to Reasoning with Uncertainty, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 374–385, doi:10.1007/978-3-642-22152-1_32.

[20] Costas D. Koutras & Yorgos Zikos (2015): *Relating Truth, Knowledge and Belief in epistemic states*. Online: `http://users.uop.gr/~ckoutras/KZ-KBstructures-Dec2015.pdf`.

[21] Stephan Merz (2005): *TLA: Lamport's Temporal Logic of Actions*. Online: `https://isabelle.in.tum.de/library/HOL/HOL-TLA/README.html`. Directory in the Isabelle/HOL Library.

[22] Leonardo Pacheco & Kazuyuki Tanaka (2022): *On the Degrees of Ignorance: via Epistemic Logic and μ-Calculus*. In: Proceedings of SOCREAL2022 6th International Workshop on Philosophy and Logic of Social Reality, pp. 74–78. Available at `http://hdl.handle.net/2115/84806`.

[23] Rasmus Rendsvig & John Symons (2021): *Epistemic Logic*. In Edward N. Zalta, editor: The Stanford Encyclopedia of Philosophy, Summer 2021 edition, Metaphysics Research Lab, Stanford University.

[24] Salomon Sickert (2016): *Linear Temporal Logic*. Archive of Formal Proofs. `https://isa-afp.org/entries/LTL.html`, Formal proof development.

[25] Robert Stalnaker (2006): *On logics of knowledge and belief*. Philosophical Studies 128, pp. 169–199, doi:10.1007/S11098-005-4062-Y.

[26] Robert Stalnaker (2009): *Lecture Notes | Modal Logic | Linguistics and Philosophy | MIT OpenCourseWare*. Available at `https://dspace.mit.edu/bitstream/handle/1721.1/100157/24-244-fall-2009/contents/lecture-notes/index.htm`.

[27] Stefano Tonetta, Marco Gario, Alessandro Cimatti & Marco Bozzano (2015): *Formal design of asynchronous fault detection and identification components using temporal epistemic logic*. Logical Methods in Computer Science 11, doi:10.2168/LMCS-11(4:4)2015.

[28] Jørgen Villadsen, Asta Halkjær From, Alexander Birch Jensen & Anders Schlichtkrull (2022): *Interactive Theorem Proving for Logic and Information*, pp. 25–48. Springer International Publishing, Cham, doi:10.1007/978-3-030-90138-7_2.

Formalizing Factorization on Euclidean Domains and Abstract Euclidean Algorithms[*]

Thaynara Arielly de Lima
Universidade Federal de Goiás, Brasil
thaynaradelima@ufg.br

Andréia Borges Avelar
Universidade de Brasília, Brasil
andreiaavelar@unb.br

André Luiz Galdino
Universidade Federal de Catalão, Brasil
andregaldino@ufcat.edu.br

Mauricio Ayala-Rincón
Universidade de Brasília, Brasil
ayala@unb.br

This paper discusses the extension of the Prototype Verification System (PVS) sub-theory for rings, part of the PVS `algebra` theory, with theorems related to the division algorithm for Euclidean rings and Unique Factorization Domains that are general structures where an analog of the Fundamental Theorem of Arithmetic holds. First, we formalize the general abstract notions of divisibility, prime, and irreducible elements in commutative rings, essential to deal with unique factorization domains. Then, we formalize the landmark theorem, establishing that every principal ideal domain is a unique factorization domain. Finally, we specify the theory of Euclidean domains and formally verify that the rings of integers, the Gaussian integers, and arbitrary fields are Euclidean domains. To highlight the benefits of such a general abstract discipline of formalization, we specify a Euclidean gcd algorithm for Euclidean domains and formalize its correctness. Also, we show how this correctness is inherited under adequate parameterizations for the structures of integers and Gaussian integers.

1 Introduction

The NASA PVS `algebra` library ([5]) was recently enriched with a series of theorems related to the theory of rings. The extension includes complete formalizations of the isomorphism theorems for rings, principal, prime, and maximal ideals, and a general abstract version of the Chinese Remainder Theorem (CRT), which holds for abstract rings, including non-commutative rings. The benefit of formalizing algebraic results from this abstract theoretical perspective was made evident by showing how, from the abstract version of CRT, the well-known numerical version of CRT for the ring of integers $\mathbb{Z}$ was formalized [22].

In this work, we give another substantial step towards enriching the PVS abstract algebra library by formalizing properties about factorization in commutative rings regarding both unique factorization domains and Euclidean rings. Roughly, unique factorization domains are abstract structures for which a general version of the Fundamental Theorem of Arithmetic holds. On the other hand, Euclidean rings are equipped with a norm that allows defining a suitable generalization of Euclid's division lemma and, consequently, of notions such as the greatest common divisor (`gcd`). The practicality of `gcd` is well-known in the ring $\mathbb{Z}$. Nevertheless, mathematicians know this notion is of fundamental importance in abstract Euclidean domains for which, in general, `gcd` should and may be defined in different ways.

Figure 1 highlights the subtheories subject of the extension to the PVS theory `algebra` discussed in this paper. The red ones are related to Euclidean rings, and `gcd` algorithms for Euclidean domains,

[*]Project supported by FAPDF DE 00193.00001175/21-11, CNPq Universal 409003/21-2, and FAPEG 202310267000223 grants. Last author partially funded by CNPq grant 313290/21-0.

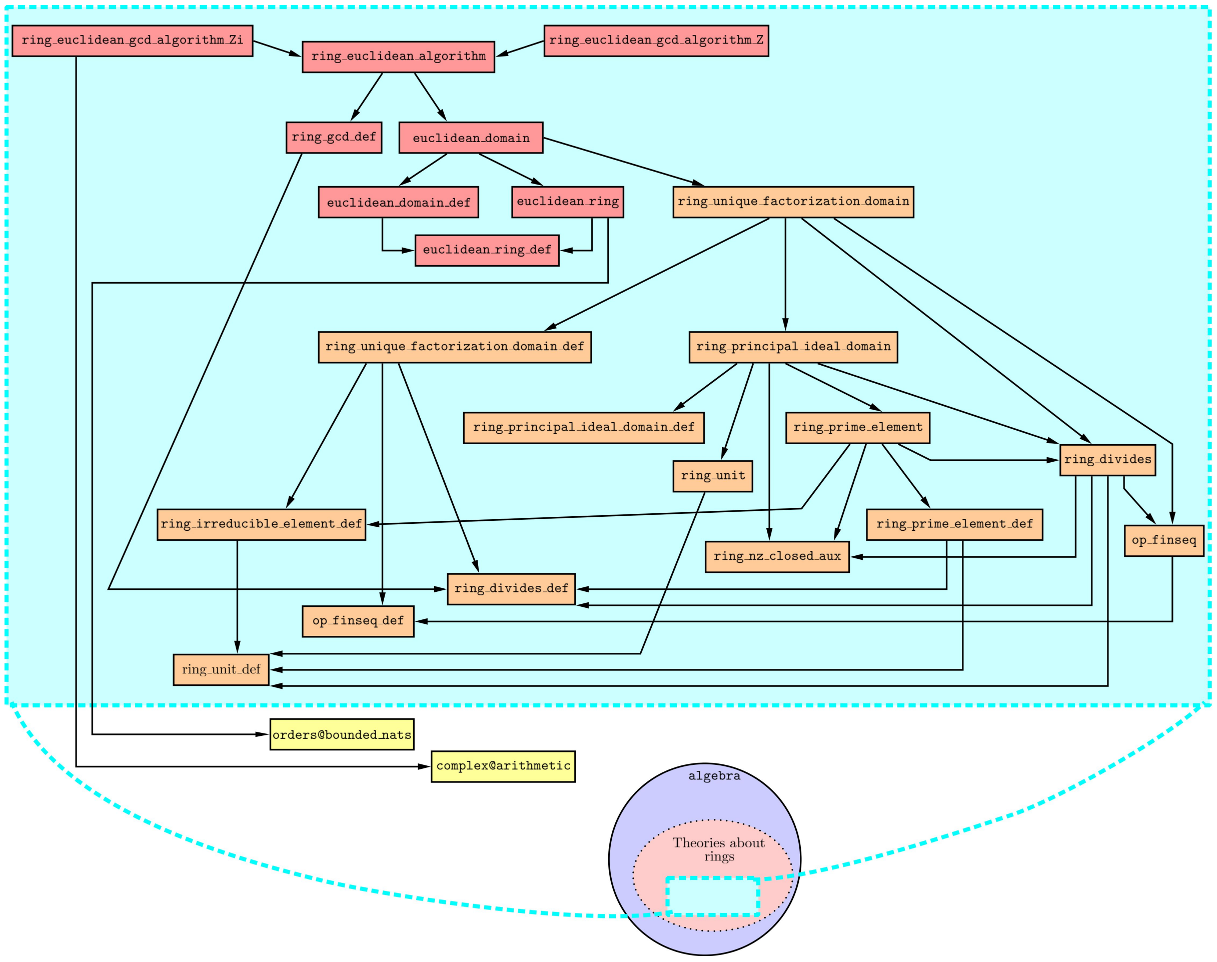


Figure 1: Ring theories expanding the PVS `algebra` library

and the orange ones are those related to unique factorization domains. The extension includes 210 new formulas enlarging the theory `algebra` from 1356 (cf [22]) to 1566 formalized lemmas.

The primary motivation to formalize such structures is their potential theoretical and practical applications. Using the example of `gcd`, one can provide a general abstract version of the Euclidean algorithm to determine a `gcd` between two elements (Euclidean `gcd` algorithm) in a Euclidean domain. Since the ring of integers $\mathbb{Z}$, the Gaussian integers $\mathbb{Z}[i]$ (which are the subset of complex numbers whose real and imaginary parts are integer numbers) and rings of polynomials over integral domains are particular Euclidean domain structures, the Euclidean `gcd` algorithm can be applied over them, in a relatively straightforward manner, to compute `gcd`s in different manners, not only for the structures mentioned above but for a variety of Euclidean domains.

Also, every element of a unique factorization domain can be factorized as a finite number of irreducible elements, and one can prove that Euclidean domains are unique factorization domains. These properties allow us to introduce modular arithmetic, verify generic versions of Euler's Theorem and Fermat's Little Theorem for Euclidean domains, and promote factorization in Euclidean domains as a convenient feature to develop efficient algorithms in symbolic computation [21], [12]. Thus, formalizing the main results about unique factorization and Euclidean domains would allow the formal verification of more complex theories involving such structures in their scope.

The main contributions of this paper are listed below.

- We formalize the abstract notions of divisibility, prime, and irreducible elements in commutative rings, which are essential to deal with unique factorization domains. In integral domains, prime elements are irreducible. The converse is not true in general. Among other properties, we formalize the theorem that establishes that irreducible elements are also prime in principal ideal domains (as well-known, it holds in $\mathbb{Z}$).
- We specify unique factorization domains and formalize the theorem that every principal ideal domain is a unique factorization domain, which is a landmark result in abstract algebra.
- We specify the notion of Euclidean domains and formally verify that the rings $\mathbb{Z}$ and $\mathbb{Z}[i]$ and any arbitrary field are Euclidean domains.
- We specify the general abstract notion of `gcd` for commutative rings, providing a general Euclidean `gcd` algorithm for Euclidean domains, formalizing its correctness. Using this result, we parameterize the adequate norms and `gcd` relations for the rings $\mathbb{Z}$ and $\mathbb{Z}[i]$; thus, obtaining straightforwardly the correctness of such instantiations of the abstract algorithm for these Euclidean domains. In this manner, we illustrate the benefits of maintaining the abstract general discipline of formalization for algebraic theories and the potential of such a discipline for application in concrete algebraic structures.

Organization of the paper. Section 2 presents a theoretical overview of unique factorization and Euclidean domains, pointing out the main concepts and results. Also, it comments on some differences between pen-and-paper proofs presented in Hungerford's textbook [18] and this formalization. Section 3 discusses the aspects of the formalization of the Euclidean `gcd` Algorithm for Euclidean Domains, as well as its application for two particular cases. Section 4 discusses related work and work in progress. Finally, Section 5 concludes and suggests future work. The formalizations were developed using PVS and are available at algebra.

2 Formalization of Euclidean Domains

Notions such as prime element, division, and `gcd` between two elements and some landmark results, including the Fundamental Theorem of Arithmetic, Euclid's division lemma, and Euclidean Algorithm, are well established and widespread for the ring of integers. Such concepts and general versions of interesting results are extended for abstract algebraic structures ([18], [10], [13]) and are the scope of our formalization.

This section gives a theoretical overview of the central notions and properties and discusses the PVS features used in their formalization. An excellent description of the semantics of PVS is available as [24]. In addition, to highlight crucial differences between pen-and-paper vs formalized proofs, some analytical concepts and results are presented as enunciated in Chapter III of Hungerford's textbook [18].

2.1 Prime and irreducible elements on rings

The definitions of prime and irreducible elements rely on the general concept of divisibility on a ring. The specification of the notions of divisibility and associated elements are specified as the curried predicates given in Specification 1. These predicates are abstracted for any ring structure given as their first argument, `R`. In PVS, types are built from predicates; for example, `ring?` is used to build the type (ring?).

Specification 1: Divisibility and associated elements in the sub-theory ring_divides_def

```
divides?(R : (ring?))(a: (R - {zero}), b: (R)): bool =  EXISTS (x: (R)): a*x = b

associates?(R: (ring?))(a,b:(R - {zero})): bool = divides?(R)(a,b) AND
                                                  divides?(R)(b,a)
```

In Hungerford's textbook, the definition of divisibility relies on a commutative ring. It avoids the discrimination between an element's left or right divisor, and since the main results demand a commutative ring in the hypothesis, it is a reasonable requirement. However, commutativity is not a crucial property in such a notion since it only depends on the multiplication operation in a ring. Because of that, we opted to generalize the definition and specify divisibility on non-necessarily commutative rings as (`divides?(R)(a,b)`). Another interesting remark is related to the specification of `associates?(R)(a,b)`: Hungerford's textbook omits that the type of the parameters `a` and `b` are nonzero elements. Of course, this is obvious since it is required in the definition of `divides?(R)(a,b)`. However, the lack of such a hypothesis is recurrent in several statements throughout the textbook that require it (for example, in Theorem 2.1).

In the sub-theory ring_divides, we formalized the properties related to the divisibility stated in Theorem 2.1. Some of them involve the object "unit". In a ring $(R,+,*,zero,one)$ with multiplication identity *one*, an element u is called a *unit* if u is left- and right-invertible; that is, if there exist elements $u_1^{-1}, u_2^{-1} \in R$ such that $u * u_1^{-1} = u_2^{-1} * u = one$.

Theorem 2.1 (Th.3.2, Hungerford [18])**.** *Let a, b, and u be elements of a commutative ring R with identity.*

(i) *a divides b (denoted as $a \mid b$) if and only if $(b) \subset (a)$, where (x) denotes the principal ideal generated by x.*

(ii) *a and b are associates if and only if $(a) = (b)$.*

(iii) *u is a unit if and only if $u \mid r$ for all $r \in R$.*

(iv) *u is a unit if and only if $(u) = R$.*

(v) *The relation "a and b are associates" is an equivalence relation on R.*

(vi) *If $a = br$, where $r \in R$ is a unit, then a and b are associates. If R is an integral domain, then the converse is true.*

Theorem 2.1 has a straightforward formalization due to the robustness of the formal framework previously developed for rings and principal ideals [22]. The formalization of the properties *(i)*, *(ii)*, and *(iv)* illustrates it clearly. In fact, by definition, (a) denotes the intersection of all ideals in R containing the element a. The lemma principal_ideal_charac in theory `ring_principal_ideal` characterizes (a) as the set one_gen(R)(a) in the theory `ring_one_generator`. The last characterization depends on a sum, specified as `R_sigma`, over elements of a function in the ring R, defined over abstract types, as given in the theory ring_basic_properties. The constructor `R_sigma` generalizes constructors in the NASA PVS library (nasalib) built for specific theories as the theory of reals. Also, since R is a commutative ring with identity, the lemma commutative_id_one_gen_charac provides a much simpler characterization of the set `one_gen(R)(a)`; indeed, such characterization simplifies the analysis of properties *(i)*, *(ii)*, and *(iv)* since (a) can be built as the set $aR = \{ar \,:\, r \in R\}$.

From the concepts of divisibility and unit, we specified prime and irreducible elements on a ring with identity as the predicates given in the Specification 2.

Specification 2: Irreducible and prime elements in the subtheories `ring_irreducible_element_def` and `ring_prime_element_def`, respectively

```
R_irreducible_element?(R : (ring_with_one?))(x:(R)): bool = x/=zero AND
    (NOT unit?(R)(x)) AND
    (FORALL (a,b:(R)): x = a*b IMPLIES (unit?(R)(a) OR unit?(R)(b)))

R_prime_element?(R : (ring_with_one?))(x:(R)): bool = x/=zero AND (NOT unit?(R)(x)) AND
 (FORALL (a,b:(R)): divides?(R)(x, a*b) IMPLIES
                    divides?(R)(x, a) OR divides?(R)(x, b))
```

In the ring of integers, prime and irreducible elements are indistinguishable. However, this is not true for all rings. For instance, 2 is prime but not irreducible in $\mathbb{Z}_6$. Theorem 2.2 gives some properties regarding prime and irreducible elements formalized in the subtheories `ring_principal_ideal_domain` and `ring_prime_and_irreducible_element`. It shows, among other results, that prime and irreducible elements are equal over principal ideal domains.

Theorem 2.2 (Th.3.4, Hungerford [18])**.** *Let p and c be nonzero elements in an integral domain R.*

(i) p is prime if and only if (p) is a nonzero prime ideal;

(ii) c is irreducible if and only if (c) is maximal in the set S of all proper principal ideals of R.

(iii) Every prime element of R is irreducible.

(iv) If R is a principal ideal domain, then p is prime if and only if p is irreducible.

(v) Every associate of an irreducible [resp. prime] element of R is irreducible [resp. prime].

(vi) The only divisors of an irreducible element of R are its associates and the units of R.

Although the result is stated for integral domains, Hungerford advises that a weakened hypothesis can be considered in some parts of the theorem. We formalize the results using the minimum number of required conditions and detect that items (i) and (vi) of the Theorem 2.2 hold for commutative rings with identity.

Properties *(i)*, *(ii)*, and *(iii)* form the basis for the formalization of the characterization of primes as irreducible elements over principal ideal domains, given in property *(iv)* and specified as the lemma `PID_prime_el_iff_irreducible`. The sufficiency of the property *(iv)*, established in the property *(iii)*, is verified as the lemma `prime_el_is_irreducible`. It follows in a relatively straightforward manner from the definition of prime elements and the result that the multiplicative cancelation law holds for non-zero elements in integral domains (lemma `nzd_R_cancel_left`) since integral domains have no zero divisors.

On the other hand, the necessity of *(iv)* is trickier since it depends on properties *(i)*, *(ii)*, and other additional previous results developed for rings with identity and maximal ideals. Property *(i)* is specified by the lemma `prime_el_iff_prime_ideal`. Its proof depends on the lemmas `prime_ideal_prop1` and `prime_ideal_prop2` formalized in theory `ring_prime_ideal`, which provide a characterization of prime ideals over commutative rings. Lemma `el_irred_iff_one_gen_maximal` specifies property *(ii)*, which establishes that the principal ideal generated by an irreducible element is a maximal element in the set S of all proper principal ideals of a ring. It is important to stress here that in the pen-and-paper proof of property *(iv)* given in [18], Hungerford assumes the vital result that maximal elements in the previously mentioned set S are maximal ideals in R. We formalized this property without this assumption as the lemma `el_max_iff_one_gen_maximal` in the sub-theory

Specification 3: Theory `ring_unique_factorization_domain_def` with the definition of unique factorization domain

```
fsIr?(R)(fsI: finseq[(R)]): bool = FORALL (i: below[length(fsI)]):
     R_irreducible_element?(R)(fsI(i))

unique_factorization_domain?(R): bool = integral_domain_w_one?(R) AND
FORALL(a: (R)): a /= zero AND NOT unit?(R)(a) IMPLIES
 EXISTS(fsI:(fsIr?(R))):a = op_fseq(fsI) AND
 FORALL(fsIp:fsIr(R)):a = op_fseq(fsIp) IMPLIES length(fsI) = length(fsIp) AND
 EXISTS(phi:[below[length(fsI)]->below[length(fsI)]]): (bijective?(phi)) AND
 FORALL(i:below[length(fsI)]): associates?(R)(fsIp(phi(i)),fsI(i))
```

`ring_principal_ideal_domain`. Finally, the necessity of property *(iv)* is concluded as follows. If p is an irreducible element, then (p) is a maximal element, according to `el_max_iff_one_gen_maximal`. Since R is a ring with identity, $R^2 = R$ by lemma `ring_w_one_is_idempotent`, which is formalized in the sub-theory `ring_with_one_basic_properties`. Consequently, (p) is a prime ideal by the lemma `maximal_prime_ideal` and, by property (i), p is a prime element.

2.2 Unique Factorization Domains

The well-known Fundamental Theorem of Arithmetic for integers states that any positive integer greater than 1 can be factorized as a unique product of primes up to a permutation of such factors. Unique Factorization Domains (UFDs) are integral domains with an analogous theorem. The Specification 3 shows the definition of UFDs. It depends on a sequence of irreducible elements `fsIr?`(R)(`fsI`) on a ring R with identity and a recursive operator `op_fseq`(`fsI`), as specified in the sub-theory `op_finseq_monoid_def`, which multiplies the elements of such a sequence. The operator `op_fseq`(`fsI`) is specified over an abstract structure $(T, *, one)$ equipped with a binary operation $*$ and a constant *one*.

From the point of view of formalization, such a general specification is very useful for two reasons: firstly, it allows the use of the operator `op_fseq`(`fsI`) in a variety of abstract and concrete structures (monoids, monads, groups, rings, integers, reals) by only adequately parameterizing the sub-theory `op_finseq_def`; secondly, it avoids proof obligations, called in PVS *Type Correctness Conditions (TCCs)*, automatically generated by the system, since the operator is defined for elements of an abstract type, which provides more automation in our formal verification. Indeed, suppose such an operator was defined over elements of an algebraic structure, for example, a monad. To each application of that definition in a specific context, PVS will automatically generate a proof obligation to verify that `op_fseq`(`fsI`) acts on a sequence whose elements belong to a monad. This specification design would make the theory verification more onerous. It is advantageous to use polymorphism to formalize concepts and properties that hold for a non-interpreted type since it allows the reuse of such results in multiple contexts.

In sub-theory `ring_unique_factorization_domain`, we formalized the Theorem 2.3, which is a landmark result about UFDs.

Theorem 2.3 (Th.3.7, Hungerford [18])**.** *Every principal ideal domain is a unique factorization domain.*

The formalization of the Theorem 2.3 has two main steps. We briefly comment on them.

Step 1 - Existence of a factorization

First, previous subtheories established in the PVS theory `algebra` were enriched with auxiliary results. The new lemma `chain_ideal_ union_ideal`, which states that the union of a chain of ideals in a ring R is an ideal, is included in the sub-theory `ring_ideal`. The new lemma `nonzero_`

Specification 4: Auxiliary function to build an ascending chain of ideals

```
phi(n:nat, R:principal_ideal_domain, a:(non_fact_el_set(R))):
 RECURSIVE (non_fact_el_set(R)) =
  IF n = 0 then a
  ELSE  choose ({x : (non_fact_el_set(R))|
                strict_subset?(one_gen(R)(phi(n-1, R, a)),one_gen(R)(x))})
  ENDIF
 MEASURE n
```

ring_ exists_maximal_ideal_aux, which proves that every ideal in a ring R with identity, except R itself, is contained in a maximal ideal in R, is added to the sub-theory ring_with_ one_maximal_ideal.

The formalization of this lemma considers an ideal $A \neq R$, $S = \{B \subset R;\ B$ is ideal in $R, B \neq R$ and $A \subset B\}$ and $\mathscr{C} = \{C_i \mid i \in I\}$ an arbitrary chain of ideals in S. We prove that the ideal $C = \bigcup C_i$ is an upper bound of the chain $\mathscr{C}$ in S and, by using Zorn's lemma (available in the NASA PVS theory orders), we conclude that S has a maximal element, which is a maximal ideal in R. In the sub-theory ring_principal_ideal, we add the new lemma stable_chain, which states that if R is a principal ideal ring and $(a_1) \subset (a_2) \ldots$ is a chain of ideals in R, then for some positive integer n, $(a_j) = (a_n)$ for all $j \geq n$. The new lemma nonzero_nonunit_irreducible_divides, formalized in the sub-theory ring_principal_ideal_domain, states that every nonzero and non-unit element in a principal ideal domain is divided by an irreducible element.

We conclude Step 1 by verifying that the subset of R below, a principal ideal domain, is empty.

$$\texttt{non\_fact\_el\_set}(R) = \left\{ \begin{array}{lll} x & : & x \text{ is a nonzero non-unit element} \\ & & \text{in } R \text{ and cannot be finitely} \\ & & \text{factorized into irreducible elements} \end{array} \right\}$$

In fact, if $a \in$ non_fact_el_set(R), we could build an ascending chain of ideals, $(a) \subset (a_1) \subset \ldots$, contradicting the lemma stable_chain. The key to verifying it is to specify the recursive function phi(n,R,a) showed in Specification 4 (sub-theory ring_principal_ideal_domain) and verify that it is well defined whenever non_fact_el_ set(R) is non-empty. In PVS, recursive definitions must be accompanied by a measure to prove termination. The measure has to decrease after each recursive step.

Whenever $a \in$ non_fact_el_set(R), the choice of the element a_1, obtained by the function choose in Specification 4, is guaranteed. In fact, the lemma nonzero_nonunit_ irreducible_divides ensures that $a = ca_1$, where c is irreducible. It implies that a_1 belongs to non_fact_el_set(R) and satisfies the condition $(a) \subset (a_1)$ by Theorem 2.1*(i)*.

Step 2: "Uniqueness" of a factorization

By uniqueness we mean the existence of a bijective function between the elements of two factorizations mapping associated elements. First, we formalized the lemma prime_el_ divides (sub-theory ring_prime_and_irreducible_element) which states if a prime element p in an integral domain divides the product $a_1 \ldots a_n$ then there exists i, $1 \leq i \leq n$, such that p divides a_i. By 2.2*(iii)*, p is an irreducible element since p is a prime element. From this, if $a_1 \ldots a_n = a = b_1 \ldots b_m$, where $a_i, 1 \leq i \leq n$, and b_j, $1 \leq j \leq m$, are irreducible elements, then a_1 divides b_j, for some j. By Theorem 2.2*(vi)*, a_1 and b_j are associates. Using induction on n, we prove that $n = m$ and establish the required bijective function.

Specification 5: Definitions of Euclidean rings and Euclidean domains

```
euclidean_ring?(R): bool = commutative_ring?(R) AND
EXISTS (phi: [(R - {zero}) -> nat]): FORALL(a,b: (R)):
  ((a*b /= zero IMPLIES phi(a) <= phi(a*b)) AND
   (b /= zero IMPLIES EXISTS(q,r:(R)):
    (a = q*b+r AND (r = zero OR (r /= zero AND phi(r) < phi(b))))))


euclidean_domain?(R): bool = euclidean_ring?(R) AND integral_domain_w_one?(R)
```

2.3 Euclidean Rings

A Euclidean ring is a commutative ring R equipped with a norm φ over $R-\{zero\}$, where an abstract version of the well-known Euclid's division lemma holds. Euclidean rings and domains are specified in the subtheories `Euclidean_ring_def` and `Euclidean_domain_def` (Specification 5).

In sub-theory `Euclidean_domain`, we formalized that elements of Euclidean ring can be factorized as irreducible elements by verifying Theorem 2.4.

Theorem 2.4 (Th.3.9, Hungerford [18])**.** *A Euclidean ring R is a principal ideal ring with identity. Consequently, every Euclidean domain is a unique factorization domain.*

The verification makes use of the well-ordering principle over $\varphi(I^*) = \{\varphi(x) \in \mathbb{N}; \text{x} \in I-\{zero\}\}$, where I is a nonzero ideal in R and φ is a norm on $R-\{zero\}$. By choosing $a \in I$ such that $\varphi(a)$ is the minimum element of $\varphi(I^*)$, $b \in I$ satisfies $b = qa + r$, for some $q \in R$ and $r \in I$. From this, we infer that $r = 0$, since $r \neq 0$ contradicts the minimality of $\varphi(a)$. Consequently, $b = qa$ and $I \subset Ra \subset (a) \subset I$. The last guarantees that every ideal in R is a principal ideal. By Theorem 2.3, we have that a Euclidean principal ideal domain is a unique factorization domain.

We also formalized the results stating that the ring of integers (integers_is_euclidean_domain) and any arbitrary field (field_is_euclidean_domain) are Euclidean domains.

3 Formalization of `gcd` Algorithm for Euclidean Domains

The theory `Euclidean_ring_def` includes two additional definitions to allow abstraction of acceptable Euclidean norms and associated functions fulfilling the properties of Euclidean rings (see Specification 6).

The first definition is the relation `Euclidean_pair?`. Given a Euclidean ring R and a Euclidean norm of non-zero elements over the naturals $\phi : R \setminus \{zero\} \to \mathbb{N}$, the predicate `Euclidean_pair?`$(R,\phi)$ holds whenever ϕ satisfies the constraints of a Euclidean norm over R.

The second definition is the curried relation given as `Euclidean_f_phi?`$(R,\phi)(f_\phi)$. This relation holds whenever `Euclidean_pair?`(R,ϕ) holds, and f_ϕ is a function from $R \times R \setminus \{zero\}$ to $R \times R$, such that for all pair of elements of R in its domain, $f_\phi(a,b)$ gives a pair of elements, say (div, rem) satisfying the constraints of Euclidean rings regarding the norm ϕ: if $a \neq zero$, $a = div * b + rem$, and if $rem \neq zero$, $\phi(rem) < \phi(b)$. These definitions are correct since the existence of such a ϕ and f_ϕ is guaranteed by the fact that R is a Euclidean ring. Also, notice that the decrement of the norm, i.e., $\phi(rem) < \phi(b)$, is the key to building an abstract Euclidean terminating procedure.

By using the previous two relations, a general abstract recursive Euclidean `gcd` algorithm is specified in the sub-theory `ring_euclidean_algorithm` as the curried definition `Euclidean_gcd_algorithm`

Specification 6: Additional definitions in the sub-theory `Euclidean_ring_def`

```
Euclidean_pair?(R : (Euclidean_ring?), phi: [(R - {zero}) -> nat]) : bool =
    FORALL(a,b: (R)): ((a*b /= zero IMPLIES phi(a) <= phi(a*b)) AND
                       (b /= zero IMPLIES
                         EXISTS(q,r:(R)): (a = q*b+r AND
                              (r = zero OR (r /= zero AND phi(r) < phi(b))))))

Euclidean_f_phi?(R : (Euclidean_ring?),
                  phi : [(R - {zero}) -> nat] | Euclidean_pair?(R,phi))
                 (f_phi : [(R) , (R - {zero}) -> [(R),(R)]]) : bool =
                  FORALL (a : (R), b :(R - {zero})):
                   IF a = zero THEN f_phi(a,b) = (zero, zero)
                   ELSE LET div = f_phi(a,b)`1, rem = f_phi(a,b)`2 IN
                      a = div * b + rem AND
                     (rem = zero OR (rem /= zero AND phi(rem) < phi(b)))
                   ENDIF
```

(See Specification 7). The types of its arguments guarantee the correctness of this algorithm. Indeed, since allowed arguments R,ϕ, and f_ϕ should satisfy `Euclidean_f_phi?`$(R,\phi)(f_\phi)$, R is a Euclidean ring with associated Euclidean norm ϕ and adequate division and remainder functions given by f_ϕ. The termination of the algorithm is a *proof obligation* (termination TCC) automatically generated by PVS using the lexicographical `MEASURE` given in the specification. This measure decreases after each possible recursive call: for `Euclidean_gcd_algorithm`$(R,\phi,f_\phi)(a,b)$, if $a \neq zero$, $\phi(a) \geq \phi(b)$ and $rem \neq zero$, the recursive call is `Euclidean_gcd_algorithm`$(R,\phi,f_\phi)(b,rem)$; thus, the pair $(\phi(b),\phi(a))$ is lexicographically greater than $(\phi(rem),\phi(b))$, since $\phi(b) > \phi(rem)$. In the other case, the recursive call is `Euclidean_gcd_algorithm`$(R,\phi,f_\phi)(b,a)$. This happens if $a \neq zero$, and $\phi(b) > \phi(a)$; therefore, $(\phi(b),\phi(a))$ is lexicographically greater than $(\phi(a),\phi(b))$.

It is worth mentioning that such termination TCCs are generated automatically by PVS, but in general, as in this case, the mandatory proof must be formalized manually.

The proof of correctness of the recursive algorithm is given as a straightforward corollary of the `Euclid_theorem` (in Specification 7) that establishes the correctness of each recursive step regarding the abstract definition of `gcd` given in Specification 8. Essentially, this theorem states that given an adequate Euclidean norm ϕ and associated function f_ϕ, the `gcd` of a pair (a,b) is equal to the `gcd` of the pair (rem,b), where rem is computed through f_ϕ, i.e., rem is equal to the second projection of $f_\phi(a,b)$. Notice that since Euclidean rings allow a variety of Euclidean norms and associated functions (e.g., [18], [13]), the definition of `gcd` is not specified as a function but as the relation `gcd?`.

Finally, the proof of correctness of the abstract Euclidean algorithm is obtained by induction, using the lexicographic `MEASURE` of the algorithm. The theorem `Euclidean_gcd_alg_correctness` (in Specification 7) formalizes this fact. For an input pair (a,b), in the inductive step of the proof, when $\phi(b) > \phi(a)$ and the recursive call swaps the arguments, one assumes that

$$\texttt{gcd?}(R)(\{b,a\},\texttt{Euclidean\_gcd\_algorithm}(R,\phi,f_\phi)(b,a)),$$

which means that `Euclidean_gcd_algorithm`$(R,\phi,f_\phi)(b,a)$ computes correctly the `gcd` of the pair (b,a). From this assumption, one concludes that

$$\texttt{gcd?}(R)(\{a,b\},\texttt{Euclidean\_gcd\_algorithm}(R,\phi,f_\phi)(a,b)).$$

Otherwise, when the recursive call is `Euclidean_gcd_algorithm`$(R,\phi,f_\phi)(b,rem)$, which happens if $\phi(a) \geq \phi(b)$, then $rem = (f_\phi(a,b))'2$, the second component of $f_\phi(a,b)$; by induction hypothesis one

Specification 7: Sub-theory `ring_euclidean_algorithm`: abstract gcd Euclidean algorithm for Euclidean rings

```
Euclidean_gcd_algorithm(R : (Euclidean_domain?[T,+,*,zero,one]),
                        (phi: [(R - {zero}) -> nat] | Euclidean_pair?(R,phi)),
                        (f_phi: [(R),(R - {zero}) -> [(R),(R)]] |
                                       Euclidean_f_phi?(R,phi)(f_phi)))
                        (a: (R), b: (R - {zero})) : RECURSIVE (R - {zero}) =
  IF  a = zero THEN b
  ELSIF  phi(a) >= phi(b) THEN
      LET rem = (f_phi(a,b))`2 IN
        IF rem = zero THEN b
        ELSE Euclidean_gcd_algorithm(R,phi,f_phi)(b,rem)
        ENDIF
  ELSE  Euclidean_gcd_algorithm(R,phi,f_phi)(b,a)
  ENDIF
MEASURE lex2(phi(b), IF a = zero THEN 0 ELSE phi(a) ENDIF)

Euclid_theorem : LEMMA
  FORALL(R:(Euclidean_domain?[T,+,*,zero,one]),
        (phi: [(R - {zero}) -> nat] | Euclidean_pair?(R, phi)),
        (f_phi: [(R),(R - {zero}) -> [(R),(R)]] |
                       Euclidean_f_phi?(R,phi)(f_phi)),
        a: (R), b: (R - {zero}), g : (R - {zero})) :
           gcd?(R)({x : (R) | x = a OR x = b}, g) IFF
           gcd?(R)({x : (R) | x = (f_phi(a,b))`2 OR x = b}, g)

Euclidean_gcd_alg_correctness : THEOREM
  FORALL(R:(Euclidean_domain?[T,+,*,zero,one]),
         (phi: [(R - {zero}) -> nat] | Euclidean_pair?(R, phi)),
         (f_phi: [(R),(R - {zero}) -> [(R),(R)]] |
                        Euclidean_f_phi?(R,phi)(f_phi)),
         a: (R), b: (R - {zero}) ) :
      gcd?(R)({x : (R) | x = a OR x = b},
              Euclidean_gcd_algorithm(R,phi,f_phi)(a,b))
```

has that

$$\texttt{gcd?}(R)(\{b, rem\}, \texttt{Euclidean\_gcd\_algorithm}(R, \phi, f_\phi)(b, rem)).$$

Finally, by application of `Euclid_theorem`, one concludes that the abstract general Euclidean algorithm correctly computes a gcd for the pair (a,b).

Now, we show how the correctness of the abstract algorithm `Euclidean_gcd_algorithm` is easily inherited, under adequate instantiations, for the structures of integers $\mathbb{Z}$ and Gaussian integers $\mathbb{Z}[i]$. The lines of reasoning follow those given in discussions on factorization in commutative rings and multiplicative norms in textbooks (e.g., Section 47 in [13], or Chapter 3, Section 3 in [18]).

The Specification 9 presents the case of the Euclidean ring $\mathbb{Z}$. The Euclidean norm $\phi_{\mathbb{Z}}$ is selected as the absolute value while the associated function $f_{\phi_{\mathbb{Z}}}$ is built using the integer division and remainder,

Specification 8: gcd definition for commutative rings - sub-theory `ring_gcd_def`

```
gcd?(R)(X: {X | NOT empty?(X) AND subset?(X,R)}, d:(R - {zero})): bool =
    (FORALL a: member(a, X) IMPLIES divides?(R)(d,a)) AND
         (FORALL (c:(R - {zero})):
           (FORALL a: member(a, X) IMPLIES divides?(R)(c,a)) IMPLIES
    divides?(R)(c,d))
```

Specification 9: Correctness of the parameterization of the abstract Euclidean algorithm for the Euclidean ring $\mathbb{Z}$ - sub-theory ring_euclidean_gcd_algorithm_Z

```
phi_Z(i : int | i /= 0) : posnat =  abs(i)

f_phi_Z(i : int, (j : int | j /= 0)) : [int, below[abs(j)]] =
 ((IF j > 0 THEN ndiv(i,j) ELSE -ndiv(i,-j) ENDIF), rem(abs(j))(i))

phi_Z_and_f_phi_Z_ok  : LEMMA Euclidean_f_phi?[int,+,*,0](Z,phi_Z)(f_phi_Z)

Euclidean_gcd_alg_correctness_in_Z  : COROLLARY
  FORALL(i: int, (j: int | j /= 0)  ) :
    gcd?[int,+,*,0](Z)({x : (Z) | x = i OR x = j},
            Euclidean_gcd_algorithm[int,+,*,0,1](Z, phi_Z,f_phi_Z)(i,j))
```

specified in the PVS prelude libraries as ndiv and rem: for $a \in \mathbb{Z}, b \in \mathbb{Z}\setminus\{0\}$, $\texttt{div}(a,b)$ computes the integer division of a by b, and, for $b \in \mathbb{Z}^+\setminus\{0\}$, $\texttt{rem}(b)(a)$ computes the remainder of a by b.

The correctness of the Euclidean algorithm for the ring of integers is specified as the corollary Euclidean_gcd_alg_correctness_in_Z. It states that for the Euclidean ring of integers $\mathbb{Z}$, and any $i, j \in \mathbb{Z}, j \neq 0$, the parameterized abstract algorithm, Euclidean_gcd_algorithm[int,+,*,0,1] satisfies the relation gcd?[int,+,*,0]:

$$\texttt{gcd?}[int,+,*,0](\mathbb{Z})(\{i,j\},\\ \texttt{Euclidean\_gcd\_algorithm}[int,+,*,0,1](\mathbb{Z},\phi_{\mathbb{Z}},f_{\phi_{\mathbb{Z}}})(i,j))$$

The formalization of this corollary follows from the theorem of correctness for the abstract Euclidean algorithm, Euclidean_gcd_alg_correctness theorem (Specification 7), which essentially requires proving that the chosen Euclidean measure $\phi_{\mathbb{Z}}$, and the associated function $f_{\phi_{\mathbb{Z}}}$ fulfill the conditions in the definition of Euclidean rings. The latter is formalized as lemma phi_Z_and_f_phi_Z_ok: $\texttt{Euclidean\_f\_phi?}[\texttt{int},+,*,0](\mathbb{Z},\phi_{\mathbb{Z}})(f_{\phi_{\mathbb{Z}}})$.

The Specification 10 presents the formalization of the correctness of the Euclidean algorithm for the Euclidean ring $\mathbb{Z}[i]$ of Gaussian integers. The Euclidean norm of a Gaussian integer $x = (\texttt{Re}(x) + i\,\texttt{Im}(x)) \in \mathbb{Z}[i]$ is considered as the natural number given by $\phi_{\mathbb{Z}[i]}(x) = x * \texttt{conjugate}(x) = \texttt{Re}(x)^2 + \texttt{Im}(x)^2$, where $\texttt{conjugate}(x) = \texttt{Re}(x) - i\,\texttt{Im}(x)$. The construction of an adequate associated function $f_{\phi_{\mathbb{Z}[i]}}$ (f_phi_Zi in Specification 10) requires additional explanations and is specified through the auxiliary function div_rem_appx. For a pair of integers (a,b), $b \neq 0$, this function computes the pair of integers (q,r) such that $a = qb + r$, and $|r| \leq |b|/2$; thus, qb is the integer closest to a. The equality $a = qb + r$ is formalized as lemma div_rev_appx_correctness. Several properties about the field of complex numbers are imported from the PVS complex theory.

Now, we explain the construction of the function $f_{\phi_{\mathbb{Z}[i]}}$. For y, a Gaussian integer and x, a positive integer, let $\texttt{Re}(y) = q_1x + r_1$ and $\texttt{Im}(y) = q_2x + r_2$, where (q_1, r_1) and (q_2, r_2) are computed with the auxiliary function div_rem_appx (with respective inputs $(\texttt{Re}(y), x)$ and $(\texttt{Im}(y), x)$). Let $q = q_1 + iq_2$ and $r = r_1 + ir_2$, then $y = qx + r$. Also, notice that if $r \neq 0$ then $\phi_{\mathbb{Z}[i]}(r) \leq \phi_{\mathbb{Z}[i]}(x)$, since $r_1^2 + r_2^2 \leq x^2/2 \leq x^2$. For the case in which x is a nonzero Gaussian integer, $\phi_{\mathbb{Z}[i]}(x) > 0$ holds.

Then, we can compute $\texttt{div\_rem\_appx}(y\,\texttt{conjugate}(x), x\,\texttt{conjugate}(x))$, obtaining $q, r' \in \mathbb{Z}[i]$ such that $y\,\texttt{conjugate}(x) = q\,(x\,\texttt{conjugate}(x)) + r'$, and $r' = 0$ or $\phi_{\mathbb{Z}[i]}(r') < \phi_{\mathbb{Z}[i]}(x\,\texttt{conjugate}(x))$.

By selecting $r = y - qx$, we obtain $y = qx + r$ and $r\,\texttt{conjugate}(x) = r'$.

Finally, when $r \neq 0$, since $\phi_{\mathbb{Z}[i]}(r\,\texttt{conjugate}(x)) < \phi_{\mathbb{Z}[i]}(x\,\texttt{conjugate}(x))$, by application of the lemma phi_Zi_is_multiplicative, we conclude that $\phi_{\mathbb{Z}[i]}(r) < \phi_{\mathbb{Z}[i]}(x)$.

Specification 10: Correctness of the parameterization of the abstract Euclidean algorithm for $\mathbb{Z}[i]$ - subtheory ring_euclidean_gcd_algorithm_Zi

```
Zi: set[complex] = {z : complex | EXISTS (a,b:int): a = Re(z) AND b = Im(z)}

Zi_is_ring: LEMMA ring?[complex,+,*,0](Zi)

Zi_is_integral_domain_w_one: LEMMA integral_domain_w_one?[complex,+,*,0,1](Zi)

phi_Zi(x:(Zi) | x /= 0): nat = x * conjugate(x)

phi_Zi_is_multiplicative: LEMMA
   FORALL((x: (Zi) | x /= 0), (y: (Zi) | y /= 0)):
                  phi_Zi(x * y) = phi_Zi(x) * phi_Zi(y)

div_rem_appx(a: int, (b: int | b /= 0)) : [int, int] =
  LET r = rem(abs(b))(a),
      q = IF b > 0 THEN ndiv(a,b) ELSE -ndiv(a,-b) ENDIF  IN
   IF r <= abs(b)/2 THEN (q,r)
   ELSE IF b > 0 THEN (q+1, r - abs(b))
        ELSE (q-1, r - abs(b))
        ENDIF
   ENDIF

div_rev_appx_correctness : LEMMA
   FORALL (a: int, (b: int | b /= 0)) :
      abs(div_rem_appx(a,b)`2) <= abs(b)/2 AND
      a = b * div_rem_appx(a,b)`1 +  div_rem_appx(a,b)`2

f_phi_Zi(y: (Zi), (x: (Zi) | x /= 0)): [(Zi),(Zi)] =
  LET q = div_rem_appx(Re(y * conjugate(x)), x * conjugate(x))`1 +
          div_rem_appx(Im(y * conjugate(x)), x * conjugate(x))`1 * i,
      r = y - q * x IN (q,r)

 phi_Zi_and_f_phi_Zi_ok: LEMMA
    Euclidean_f_phi?[complex,+,*,0](Zi,phi_Zi)(f_phi_Zi)

 Euclidean_gcd_alg_in_Zi: COROLLARY
  FORALL(x: (Zi), (y: (Zi) | y /= 0)  ) :
      gcd?[complex,+,*,0](Zi)({z :(Zi) | z = x OR z = y},
       Euclidean_gcd_algorithm[complex,+,*,0,1](Zi, phi_Zi,f_phi_Zi)(x,y))
```

The formalization of the correctness of the Euclidean algorithm for Gaussian integers obtained by parameterizations with $\mathbb{Z}[i]$, its Euclidean norm $\phi_{\mathbb{Z}[i]}$ and associated function $f_{\phi_{\mathbb{Z}[i]}}$ follows as the simple corollary Euclidean_gcd_alg_ in_Zi in Specification 10. This is proved using the correctness of the abstract Euclidean algorithm (Specification 7) and lemma phi_Zi_and_f_phi_Zi_ok. The latter states that the Euclidean norm $\phi_{\mathbb{Z}[i]}$ and its associated function $f_{\phi_{\mathbb{Z}[i]}}$ are adequate for the Euclidean ring $\mathbb{Z}[i]$: $\texttt{Euclidean\_f\_phi?}[\texttt{complex}, +, *, 0](\mathbb{Z}[i], \phi_{\mathbb{Z}[i]})(f_{\phi_{\mathbb{Z}[i]}})$.

4 Related Work and work in progress

4.1 Related work

Several formalizations focus on specific ring structures as the ring of integers. Such developments range from simple formalization exercises, such as correctness proofs of gcd algorithms for $\mathbb{Z}$, to elaborated mechanical proofs of the Chinese Remainder theorem for $\mathbb{Z}$. The latter started from Zhang and Hua's RRL (Rewrite Rule Laboratory) mechanization [31], followed by different approaches in Mizar, HOL

Light, hol98, Coq [29], ACL2 [27], and VeriFun [30]. Also, in the theory `ints@gcd.pvs` from NASA PVS library, one finds Euclid's algorithm restricted to integer numbers. Nevertheless, the general algebraic abstract approach is followed by a few developments. In particular, such an approach is followed in the Isabelle/HOL Algebra Library (see [2], [1], and [4]); a library that provides a wide range of theorems on mathematical structures, including results on rings, groups, factorization over ideals, rings of integers and polynomial rings, as well as formalization of an algorithm to compute echelon forms over Euclidean domains, and so characteristic polynomials of matrices. Also, the Lean mathlib library [9] specifies unique factorization domains, prime and irreducible elements in commutative rings, and relations with principal ideal domains. In addition, it specifies the notion of `gcd` for Euclidean domains and formalizes several properties as the correctness of the extended Euclidean algorithm by applying Bézout's `gcd` lemma. The library mathlib formalizes that a Euclidean domain is a principal ideal domain, and a principal ideal domain is a unique factorization domain. The former is given as formally verified construction from a definition. From this instance, it is possible to infer that the Gaussian integers are a Euclidean domain and thus a principal ideal. Also, the Euclidean algorithm can be adapted to structures such as the Gaussian integers. A recent extension of mathlib specifies the ring of Witt vectors and formalizes the isomorphism between the ring of Witt vectors over $\mathbb{Z}/p\mathbb{Z}$ and the ring of p-adic integers $\mathbb{Z}p$, for a prime p [8].

In Coq, results about groups, rings, and ordered fields were formalized as part of the FTA project [15]; this work gave rise to the formalization of the Feit and Thompson's proof of the Odd Order Theorem [16]. Also, there are formalizations in Coq of real ordered fields [7], finite fields [26], and rings with explicit divisibility [6]. In Nuprl and Mizar, there are proofs of the Binomial Theorem for rings in [19] and [28], respectively, and a Mizar formalization of the First Isomorphism Theorem for rings [20]. ACL2 has a hierarchy of algebraic structures ranging from setoids to vector spaces that aims the formalization of computer algebra systems [17].

Regarding the paper-and-pen proofs in [18] and the formalization reported in this paper, the last one comprises about twenty pages of Hungerford's textbook. We estimate it took ten months of human labor. Some results in the book appear as trivial remarks only. Nevertheless, they required the formalization of a significative sequence of auxiliary lemmas. An excellent example of the lack of details in this respect is a remark after Definition 3.5. in [18] that we have used in tutorials to motivate mathematicians to deal with proof assistants. It states that:

> *"every irreducible element in a unique factorization domain is necessarily prime by Definition 3.5. Consequently, irreducible and prime elements coincide, by Theorem 3.4."*

Indeed, the formalization of this remark required the application of additional properties related to bijective functions, the equivalence relation "associates", and the composition of finite sequences, among others inherited from the abstract structure integral domain. Also, several particular cases had to be analyzed to ensure the result when the elements involved are units or equal to zero.

Finally, we would like to stress that the project is focused on the formalization side, but aspects related to code extraction can be explored through tools provided by PVS. For instance, PVSio is an animation tool that extends the ground evaluator of PVS with a predefined library of programming features [23]. The evaluation is possible whenever algorithms are specified constructively. For our purposes, this means that we can run the formalized `gcd` algorithm with the help of the ground evaluator for any Euclidean domain for which the Euclidean norm ϕ and the associated function f_ϕ are specified constructively, which is the case of our specifications for $\mathbb{Z}$ and $\mathbb{Z}[i]$. Based on PVSio, elaborated approaches use this animation tool to evaluate the formal models on a set of randomly generated test cases, comparing the computed results against output values obtained by actual software [11]. After applying such approaches, performance and comparison with other implementations would be possible.

4.2 Work in progress and applications

Work in progress to be reported in the future includes formalizing the general theory of quaternions built from any abstract structure of fields specified in PVS as commutative division rings. The specification of quaternions is given from an abstract type `T` with binary operators for addition and multiplication, with constants `zero` and `one`, respectively. The type `T` with addition and `zero` is an Abelian group, and the multiplication is associative. The specification includes axioms for quaternion addition and multiplication ($i^2 = a$, $j^2 = b$, for some given parameters a and b of T), associativity for quaternion multiplication, distributivity of quaternion addition and multiplication, and properties for the scalar product between elements of the field and of the quaternion. All that is provided in the theory quaternion_def. Afterward, in the PVS theory quaternions, using these axioms, a series of general properties of quaternions are provided, which range from the characterization of quaternion multiplication to the characterization of quaternions as division rings. Once again, following the general approach to specifying quaternions from abstract fields, we can obtain the specific structure of Hamilton's quaternions as the theory quaternions_Hamilton, using as the parameter to build the quaternions the specific field of reals [3]. As far as we know, there are formalizations of Hamilton's quaternions in HOL Light and Isabelle/HOL (e.g., [14], [25]). In contrast, some elements of the general theory of quaternions built over any abstract field, as in our case, were developed as part of the Lean mathlib library [9].

We do not argue that any proof assistant is a better or worse framework than any other for formalizing algebraic notions and properties. However, we are confident that the current formalization work adequately explores the inductive and higher-order possibilities available in PVS and substantially contributes to completing the theory of the algebraic properties of rings by providing the most general and abstract possible presentation of such algebraic structures, as also given in some of the previous references, mainly as done by the approaches mentioned above in Isabelle/HOL and Lean ([1], [9]).

5 Conclusions and Future Work

In contrast to other works, restricted to specific ring structures, our formalization approach focuses on the theory of abstract rings, as done in the Lean- and Isabelle-related libraries (cf [9] and [4], respectively) discussed in the related work. Advantages of such an approach include increasing the interest of mathematicians in formalizations and having practical general presentations of computational algebraic properties portable to specific ring structures. In particular, in [22], the Chinese Remainder Theorem was formalized for (non-necessarily commutative) rings, obtaining, as a corollary, the CRT version for the ring of integers. This work substantially extends the `algebra` PVS library by specifying Euclidean rings and factorization domains and formalizing the correspondence between principal ideal domains and unique factorization domains. Also, it proves the correctness of a general Euclidean `gcd` algorithm for Euclidean domains. The usefulness of such an abstract verified `gcd` algorithm is evident by its adaptation to specific Euclidean domain structures. Indeed, this versatility is illustrated by showing how simple corollaries establish the correctness of the Euclidean algorithm (parameterized) for the rings of integers and Gaussian integers ($\mathbb{Z}$ and $\mathbb{Z}[i]$).

In future work, we will include the specification of modular arithmetic and verification of generic versions of Euler's Theorem and Fermat's Little Theorem for Euclidean domains.

References

[1] Jesús Aransay, Clemens Ballarin, Martin Baillon, Paulo Emílio de Vilhena, Stephan Hohe, Florian Kammüller & Lawrence C. Paulson (2019): *The Isabelle/HOL Algebra Library*. Technical Report, Isabelle Library, University of Cambridge Computer Laboratory and Technische Universität München. Available at https://isabelle.in.tum.de/dist/library/HOL/HOL-Algebra/document.pdf.

[2] Jesús Aransay & Jose Divasón (2016): *Formalisation of the computation of the echelon form of a matrix in Isabelle/HOL*. *Formal Aspects Comput.* 28(6), pp. 1005–1026, doi:10.1007/s00165-016-0383-1.

[3] Mauricio Ayala-Rincón, Thaynara Arielly de Lima, André Luiz Galdino & Andréia Borges Avelar (2023): *Formalization of Algebraic Theorems in PVS (Invited Talk)*. In: *Proc. 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning LPAR*, *EPiC Series in Computing* 94, pp. 1–10, doi:10.29007/7jbv.

[4] Clemens Ballarin (2019): *Exploring the Structure of an Algebra Text with Locales*. *Journal of Automated Reasoning* 64, pp. 1093–1121, doi:10.1007/s10817-019-09537-9.

[5] Ricky Butler, David Lester & et al. (2007): *A PVS* Theory *for Abstract Algebra*. Available at https://github.com/nasa/pvslib/tree/master/algebra. Accessed in June 12, 2023.

[6] Guillaume Cano, Cyril Cohen, Maxime Dénès, Anders Mörtberg & Vincent Siles (2016): *Formalized linear algebra over Elementary Divisor Rings in Coq*. *Logical Methods in Computer Science* 12(2:7), pp. 1–23, doi:10.2168/LMCS-12(2:7)2016.

[7] Cyril Cohen & Assia Mahboubi (2012): *Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination*. *Logical Methods in Computer Science* 8(1:2), pp. 1–40, doi:10.2168/LMCS-8(1:2)2012.

[8] Johan Commelin & Robert Y. Lewis (2021): *Formalizing the ring of Witt vectors*. In: *Proceedings of the 10th International Conference on Certified Programs and Proofs CPP*, ACM SIGPLAN, pp. 264–277, doi:10.1145/3437992.3439919.

[9] The mathlib Community (2020): *The Lean Mathematical Library*. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020*, ACM, pp. 367–381, doi:10.1145/3372885.3373824.

[10] David S. Dummit & Richard M. Foote (2003): *Abstract Algebra*, 3 edition. Wiley.

[11] Aaron Dutle, César A. Muñoz, Anthony Narkawicz & Ricky W. Butler (2015): *Software Validation via Model Animation*. In: *Proceedings of the 9th International Conference on Tests and Proofs (TAP@STAF)*, *Lecture Notes in Computer Science* 9154, Springer, pp. 92–108, doi:10.1007/978-3-319-21215-9_6.

[12] Christian Eder, Gerhard Pfister & Adrian Popescu (2017): *On Signature-Based Gröbner Bases Over Euclidean Rings*. In: *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC*, ACM, pp. 141–148, doi:10.1145/3087604.3087614.

[13] John B. Fraleigh (2003): *A First Course in Abstract Algebra*, 7th edition. Pearson.

[14] Andrea Gabrielli & Marco Maggesi (2017): *Formalizing Basic Quaternionic Analysis*. In: *Proceedings of the 8th International Conference on Interactive Theorem Proving, ITP*, *Lecture Notes in Computer Science* 10499, Springer, pp. 225–240, doi:10.1007/978-3-319-66107-0_15.

[15] Herman Geuvers, Randy Pollack, Freek Wiedijk & Jan Zwanenburg (2002): *A Constructive Algebraic Hierarchy in Coq*. *Journal of Symbolic Computation* 34(4), pp. 271–286, doi:10.1006/jsco.2002.0552.

[16] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi & Laurent Théry (2013): *A Machine-Checked Proof of the Odd Order Theorem*. In: *Proceedings of the 4th International Conference on Interactive Theorem Proving ITP*, *Lecture Notes in Computer Science* 7998, Springer, pp. 163–179, doi:10.1007/978-3-642-39634-2_14.

[17] Jónathan Heras, Francisco Jesús Martín-Mateos & Vico Pascual (2015): *Modelling algebraic structures and morphisms in ACL2*. *Applicable Algebra in Engineering, Communication and Computing* 26(3), pp. 277–303, doi:10.1007/s00200-015-0252-9.

[18] Thomas W. Hungerford (1980): *Algebra*. *Graduate Texts in Mathematics* 73, Springer-Verlag, New York-Berlin, doi:10.1007/978-1-4612-6101-8. Reprint of the 1974 original.

[19] Paul Bernard Jackson (1995): *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra*. Ph.D. thesis, Cornell University. Available at https://ecommons.cornell.edu/handle/1813/7167.

[20] Artur Kornilowicz & Christoph Schwarzweller (2014): *The First Isomorphism Theorem and Other Properties of Rings*. *Formalized Mathematics* 22(4), pp. 291– 301, doi:10.2478/forma-2014-0029.

[21] Daniel Lichtblau (2013): *Applications of Strong Gröbner Bases over Euclidean Domains*. *International Journal of Algebra* 7(8), pp. 369–390, doi:10.12988/ija.2013.13037.

[22] Thaynara Arielly de Lima, André Luiz Galdino, Andréia Borges Avelar & Mauricio Ayala-Rincón (2021): *Formalization of Ring Theory in PVS - Isomorphism Theorems, Principal, Prime and Maximal Ideals, Chinese Remainder Theorem*. *J. Autom. Reason.* 65(8), pp. 1231–1263, doi:10.1007/s10817-021-09593-0.

[23] César Muñoz & Richard Butler (2003): *Rapid prototyping in PVS*. Technical Report NASA/CR-2003-212418, NIA-2003-03, NASA Langley Research Center (NIA).

[24] Sam Owre & Natarajan Shankar (1997, revised 1999): *The Formal Semantics of PVS*. Technical Report 97-2R, SRI International Computer Science Laboratory, Menlo Park CA 94025 USA.

[25] Lawrence C. Paulson (2018): *Quaternions*. *Arch. Formal Proofs* 2018. Available at https://www.isa-afp.org/entries/Quaternions.html.

[26] Jade Philipoom (2018): *Correct-by-Construction Finite Field Arithmetic in Coq*. Master's thesis, Master of Engineering in Computer Science, MIT. Available at https://dspace.mit.edu/handle/1721.1/119582.

[27] David M. Russinoff (2000): *A Mechanical Proof of the Chinese Remainder Theorem*. UTCS Technical Report - no longer available - ACL2 Workshop 2000 TR-00-29, University of Texas at Austin. Available at https://www.cs.utexas.edu/users/moore/acl2/workshop-2000/final/russinoff-short/paper.pdf.

[28] Christoph Schwarzweller (2003): *The Binomial Theorem for Algebraic Structures*. *Journal of Formalized Mathematics* 12(3), pp. 559–564. Available at http://mizar.org/JFM/Vol12/binom.html.

[29] Christoph Schwarzweller (2009): *The Chinese Remainder Theorem, its Proofs and its Generalizations in Mathematical Repositories*. *Studies in Logic, Grammar and Rhetoric* 18(31), pp. 103–119. Available at https://philpapers.org/rec/SCHTCR-12.

[30] Christoph Walther (2018): *A Machine Assisted Proof of the Chinese Remainder Theorem*. Technical Report VFR 18/03, FB Informatik, Technische Universität Darmstadt.

[31] Hantao Zhang & Xin Hua (1992): *Proving the Chinese Remainder Theorem by the Cover Set Induction*. In: *Proceegins of the 11th International Conference on Automated Deduction, CADE*, *Lecture Notes in Computer Science* 607, Springer, pp. 431–445, doi:10.1007/3-540-55602-8_182.

More Church-Rosser Proofs in BELUGA

Alberto Momigliano[*]
Dipartimento di Informatica,
Università degli Studi di Milano, Italy

Martina Sassella
Dipartimento di Matematica
Università degli Studi di Milano, Italy

We report on yet another formalization of the Church-Rosser property in lambda-calculi, carried out with the proof environment BELUGA. After the well-known proofs of confluence for β-reduction in the untyped settings, with and without Takahashi's complete developments method, we concentrate on η-reduction and obtain the result for $\beta\eta$ modularly. We further extend the analysis to typed-calculi, in particular System F. Finally, we investigate the idea of pursuing the encoding directly in BELUGA's meta-logic, as well as the use of BELUGA's logic programming engine to search for counterexamples.

1 Introduction

Stop me if you heard this before: the Church-Rosser theorem for β-reduction (CR(β)) is a good case study for proof assistants. In fact, Church-Rosser theorems are arguably the most formalized results in mechanized meta-theory of deductive systems.

In the beginning, the thrust was to see whether the theorem could be formalized at all: Shankar's proof in the Boyer-Moore theorem-prover [25] was a break-through and a tour de (brute) force. A few years later, Nipkow [17] made the proof much more abstract and extended it to $\beta\eta$-reduction. The workhorse encoding technique was de Bruijn indices and it was a bullet that one just had to bite.

In the following years, the Church-Rosser theorem became a benchmark to showcase how to mechanize the variable binding problem. Some partial data-points, with the understanding that these are not exhaustive not disjoint: if interested in mirroring the informal practice of working mathematicians, see the paper by Vestergaard and Brotherston [27] and the recent work by Copello et al. [6]. If you want to reason about α-conversion explicitly via quotients, see Ford and Mason's [9]. For the *locally nameless* representation, see McKinna and Pollack [14] and its modern take [5]. Library support is showcased by AUTOSUBST [23]. Nominal techniques are also well-represented, see the recent proof by Nagele and al. [16].

We shall use higher-order abstract syntax (HOAS) following up on the seminal proof of CR(β) by Pfenning in 1992 [18], when Twelf was merely Elf. Once the non-trivial issue of totality-checking was settled, that proof stood as a shining example of the benefits of HOAS, which we shall not repeat here. Confluence results did not attract further attention in this community, until Accattoli, in his proof pearl [2], liberated Huet's Coq encoding of residual theory [10] from its concrete-syntaxed infrastructure.

So why bother with yet another HOAS-based encoding? Our aim was to go beyond β-reduction and replicate Nipkow's results in the HOAS setting. Further, to extend it to typed calculi, with a minimum amount of change. One way to achieve that is via *intrinsically-typed terms* [3] and for the latter, BELUGA [20] is the natural (if not only) modern system that natively supports HOAS together with dependent types. We were also curious to see if we could achieve a significant formalization as BELUGA's novices,

[*]Member of the Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM)

with just a cursory understanding of its intricate meta-theory.[1] This is part of a more general project of developing a curriculum to teach the classic theory of the lambda-calculus to graduate students.

In passing, we evaluate (in the negative) Accattoli's suggestion [2] that even in two-level systems such as BELUGA, the specification of the semantics of the (untyped) lambda-calculus should be carried out directly in the meta-logic. In the Appendix, we explore the concept of *mechanized meta-theory model-checking* within the Beluga framework. This approach aligns with the increasing tendency to enhance proof assistants with a form of *counterexample* search. QuickChick serves as one example of this trend [12]..

In this short paper, we assume knowledge of confluence in the lambda-calculus, for which we refer to the concise presentation in [24]. We recall some basic notions; given binary relations over a set A, and their Kleene and reflexive closure $(\_)^*$, $(\_)^=$, we define:

$$\begin{tikzcd} {} \arrow[r, "R"] \arrow[d, "R"'] & {} \arrow[d, dashed, "R"] \\ {} \arrow[r, dashed, "R"'] & {} \end{tikzcd}$$

diamond for R

$$\begin{tikzcd} {} \arrow[r, "S^*"] \arrow[d, "R^*"'] & {} \arrow[d, dashed, "R^*"] \\ {} \arrow[r, dashed, "S^*"'] & {} \end{tikzcd}$$

R and S commute

$$\begin{tikzcd} {} \arrow[r, "S"] \arrow[d, "R"'] & {} \arrow[d, dashed, "R^="] \\ {} \arrow[r, dashed, "S^*"'] & {} \end{tikzcd}$$

R and S strongly commute

We say that R is *confluent* if and only it commutes with itself.

We also assume a passing familiarity with BELUGA. We simply recall that the specification of the syntax and judgments of the system under study is done in (contextual) LF, while theorems are realized as total functions in BELUGA's meta-logic. LF contexts are reified into first-class objects that can be abstracted and quantified over, being classified by context *schemas*. First-class substitutions map contexts to each other realizing properties such as weakening, strengthening and subsumption. In this development, we have taken care to state every theorem quantifying over the smallest context schema where it makes sense [8].

2 Sketch of the Formal Development

We will only sketch some of the highlights of the encoding, referring the reader to the repository[2] for most, if not all the details.

2.1 CR(β)

Pientka [19] ported to BELUGA Pfenning's encoding [18] in Twelf of the traditional parallel reduction proof à la Tait/Martin-Löf. We did the same for Licata's proof [13] via Takahashi's *complete developments* [26] This porting was similarly uneventful, save of course for the improvements that the BELUGA brings in w.r.t. Twelf. This applies in particular to the streamlined handling of context reasoning in contrast to Twelf's rather fragile notion of *regular worlds*. A detailed comparison of the relative merits of BELUGA vs. Twelf, among others, can be found in [7].

[1] With the partial exception of Kaiser's dissertation [11], all existing BELUGA's code originates exclusively from Pientka and her students.

[2] `https://github.com/martinasassella/More_CR_Proofs_Beluga`

In both proofs, there are *no* technical lemmas about variables, renamings etc. We do have to prove that parallel reduction is stable under substitution, but it is from first principles, meaning it does nor rely on properties such as weakening and exchange. The direct proof of the diamond property for parallel reduction has a complex case-analysis, just as in the informal case. Takahashi's proof is based on a *relational* rather than functional encoding of complete developments, which is fine as the proof only needs totality, not uniqueness of developments.

For future reference, we list here the HOAS encoding of the syntax of untyped lambda terms and of parallel reduction, featuring a crucial use of hypothetical judgments to internalize the variable cases:

```
LF term : type =
 | lam : (term → term) → term
 | app : term → term → term;

LF pred : term → term → type =
 | beta : ({x:term} pred x x → pred (M1 x) (M1' x)) → pred M2 M2'
                                     → pred (app (lam M1) M2) (M1' M2')
 | lm : ({x:term} pred x x → pred (M x) (M' x)) → pred (lam M) (lam M')
 | ap : pred M1 M1' → pred M2 M2' → pred (app M1 M2) (app M1' M2');
```

2.2 CR(η)

While CR(η), as well as CR($\beta\eta$) can be proved via complete developments, here we follow Nipkow's more modular *commutation* approach. We separately consider η-reduction as the congruence over the η rule $\Gamma \vdash \lambda x.M\ x \longrightarrow_\eta M$, provided $x \notin FV(M)$: η-reduction (or η-expansion, if viewed from the right to the left) is encoded in BELUGA as the type family

```
LF eta_red : term → term → type =
 | eta : eta_red (lam \x.(app M x)) M % … congruence rules as above, omitted
```

Note how the above proviso is realized within HOAS as the meta-variable `M` *not* depending on `x` in the LF function `\x.(app M x)`. There is nothing "tricky" in this encoding, as per Nipkow's discussion (§ 4.3 of op.cit.). Only one technical lemma is required:

Lemma 2.1 *(Strengthening) If $\Gamma, x \vdash M \longrightarrow_\eta N$, and M does not depend on x, neither does N and $\Gamma \vdash M \longrightarrow_\eta N$.*

Formalizing this takes a little thought, but BELUGA's primitives make it relatively easy to state and prove.

The main proof strategy relies on a classic result [22] that provides a sufficient condition for *commutation*:

Lemma 2.2 *(Commutation – Hindley-Rosen) Two strongly commuting reductions commute.*

Since both LF and BELUGA are (roughly) first-order type theories, we cannot express operations on abstract relations, in particular closures, nor can we prove the above lemma once and for all for any such relation. Hence, we instantiate R and S to η-reduction:

$$\begin{tikzcd}
\cdot \arrow[r, "\eta"] \arrow[d, "\eta"'] & \cdot \arrow[d, dashed, "\eta^="] & & \cdot \arrow[r, "\eta^*"] \arrow[d, "\eta^*"'] & \cdot \arrow[d, dashed, "\eta^*"] \\
\cdot \arrow[r, dashed, "\eta^*"'] & \cdot & \Longrightarrow & \cdot \arrow[r, dashed, "\eta^*"'] & \cdot
\end{tikzcd}$$

We define the relevant closures at the LF level, with families `eta_red*` and `eta_red=`, and prove that η strongly commutes with itself as the function `square`. Recall that neither LF nor BELUGA have first-class sigma types and therefore existential propositions need to be encoded separately:

```
LF eta*_eta=_joinable : term → term → type =
 | eta*_eta=_result : eta_red* M1 N → eta_red= M2 N → eta*_eta=_joinable M1 M2;

schema ctx = term

rec square : (γ:ctx) {M : [γ ⊢ term]}{M1 : [γ ⊢ term]}{M2 : [γ ⊢ term]}
          [γ ⊢ eta_red M M1] → [γ ⊢ eta_red M M2] → [γ ⊢ eta*_eta=_joinable M1 M2] = …
```

This is proven by induction on `[γ ⊢ eta_red M M1]` and inversion on `[γ ⊢ eta_red M M2]`, with an appeal to strengthening when a critical pair is created by the η and ξ rules. We then state, in the rather long-winded way that BELUGA requires, the above instance of the Commutation lemma:

```
LF confl_prop : term → term → type =
 | confl_result : eta_red* M1 N → eta_red* M2 N → confl_prop M1 M2;

rec commutation_lemma:(γ:ctx)({M: [γ ⊢ term]}{M1: [γ ⊢ term]}{M2: [γ ⊢ term]}
        [γ ⊢ eta_red M M1] → [γ ⊢ eta_red M M2] → [γ ⊢ eta*_eta=_joinable M1 M2])
            → ({M’: [γ ⊢ term]}{M1’: [γ ⊢ term]}{M2’: [γ ⊢ term]}
            [γ ⊢ eta_red* M’ M1’] → [γ ⊢ eta_red* M’ M2’] → [γ ⊢ confl_prop M1’ M2’]) = …
```

The proof approximates Nipkow's diagrammatic way via a concrete instance of the *Strip* lemma. In the end, what we decisively gained in elegance and succinctness with HOAS vs de Bruijn, we somewhat lose by the limitations of a first-order framework.

2.3 CR($\beta\eta$)

We take $\beta\eta$-reduction as the union of β and η-reductions. Since we have already shown confluence for each relation separately, it makes sense to exploit this other classic [22] result:

Lemma 2.3 (Commutative Union) *If R and S are confluent and commute, then $R \cup S$ is confluent.*

To establish the commutation assumption in the above lemma, we will again appeal to the Commutation lemma 2.2, hence we start with a version of the `square` function above, with $R := \beta$ and $S := \eta$. The proof requires a strengthening lemma for β-reduction, as well as two beautifully clean substitution lemmas for η and η^*. Again, this beauty is short-lived, as we have to replay the diagrammatic proof of the Commutation lemma with the current instantiation; this is a routine rework of the η case, but tedious.

The final ingredient is the proof of lemma 2.3 for $\beta\eta$, which is entailed by the following steps:

1. $(\beta^* \cup \eta^*)^* = (\beta \cup \eta)^*$;
2. $\beta^* \cup \eta^*$ satisfies the diamond property;
3. a Strip lemma for the above two relations.

2.4 Typed Calculi

We now switch gears and address confluence in *typed* lambda-calculi. While the main definitions still apply, we must be mindful to reduce only well-typed terms [24]. In a dependently typed proof environment such as BELUGA, this can be very elegantly accomplished using *intrinsically-typed terms* [3], that is, by ruling out pre-terms and indexing the judgments under study by well-typed terms only.

To illustrated the idea, we list the specification of well-typed terms in the polymorphic lambda-calculus (System F) and a fragment of parallel reduction (congruence rules for type abstraction and application together with the two beta rules):

```
LF ty : type =                     LF tm : ty → type =
 | arr : ty → ty → ty                | lam : (tm A → tm B) → tm (arr A B)
 | all : (ty → ty) → ty;             | app : tm (arr A B) → tm A → tm B
                                     | tlam : ({a:ty} tm (A a)) → tm (all A)
                                     | tapp : tm (all A) → {B:ty} tm (A B);

LF pred : tm A → tm A → type =
 | tlm : ({a:ty} pred (M a) (M' a)) → pred (tlam M) (tlam M')
 | tap : pred M M' → pred (tapp M A) (tapp M' A)
 | beta : ({x:tm A} pred x x → pred (M1 x) (M1' x)) → pred M2 M2'
                                  → pred (app (lam M1) M2) (M1' M2')
 | tbeta : ({a:ty} pred (M1 a) (M1' a)) → pred (tapp (tlam M1) A) (M1' A) …
```

Note how the signature of `pred` enforces the invariant that (parallel) reduction preserves typing. Since we now have two kinds of variables, the judgment is hypothetical both on terms and on types. On the reasoning level, this induces a context schema that accounts for this alternation, namely

```
schema pctx = some [A:ty] block(x:tm A, v:pred x x) + ty
```

What is remarkable is that literally the same proof structure of the Church-Rosser theorem carries over from the untyped case to the typed one. To wit, we have formalized Takahashi's style CR(β) for System F and the proof is *conservative* in a very strong sense: not only do we use the same sequence of lemmata, but, being BELUGA's scripts explicit proof-terms, the derivations for the untyped calculus directly *embed* into the derivations for System F: the user just needs to refine the statements of the theorems with the indexed judgments, on occasion making some implicit arguments explicit to aid type inference and of course adding cases for the new constructors. The proof of CR(β) for System F consists of around 460 loc, as opposed to 340 for the untyped case.

2.5 CR(β) at the Meta-Level

In the two-level approach adopted by BELUGA, the syntax and the semantics of a formal system are specified at the LF level, whereas reasoning is carried out at the computational level in form of recursive functions. LF features hypothetical judgments and those give for free properties of contexts such as exchange, weakening and substitution; by "for free", we mean that they need not to be proved on a case by case base. Nonetheless, the communication of context information to the reasoning level requires the definition of possibly complex context schemas and relations among them that may complicate the statements and pollute the proofs. This may be particularly annoying in case studies such as the untyped lambda calculus, where contexts do not seem to play a large part, as observed by Accattoli in his remarkable paper [2] w.r.t. the "cousin" system Abella.

Since [20], BELUGA allows one to define `inductive` and `stratified` relations at the meta-level. Therefore, we can test Accattoli's proposal w.r.t. the standard proof of CR(β) for the untyped case. This entail keeping at the LF level only the syntax and move all the other judgments at the computation level: to wit, parallel reduction has now this (non-hypothetical) representation:

```
inductive predM : (γ:ctx) [γ ⊢ term] → [γ ⊢ term] → ctype =
 | var : isVar [γ ⊢ M] → predM [γ ⊢ M] [γ ⊢ M]
 | betaM : predM [g, x:term ⊢ M1[…,x]] [g, x:term ⊢ M1'[…,x]] → predM [γ ⊢ M2] [γ ⊢ M2']
        → predM [γ ⊢ app (lam \x.M1[…,x]) M2] [γ ⊢ M1'[…,M2']] % other cases omitted

inductive isVar : (γ : ctx) {M: [γ ⊢ term]} ctype =
 | vp : {#q: [γ ⊢ term]} isVar [γ ⊢ #q]
```

```
schema rctx = block(x:term, t:pred x x)

rec rpar: {g:rctx}{M: [γ ⊢ term]}[γ ⊢ pred M M] =
mlam g ⇒ mlam M ⇒ case [γ ⊢ M] of
 | [γ ⊢ #p.1] ⇒ [γ ⊢ #p.2]
 | [γ ⊢ lam \x.M'[…,x]] ⇒
   let [g, b:block(x:term,v:pred x x) ⊢ IH[…,b.1,b.2]] =
     rpar [g, b:block(x:term,v:pred x x)] [g,b ⊢ M'[…,b.1]] in
       [γ ⊢ lm \x.\v.IH[…,x,v]]
 | [γ ⊢ app M1 M2] ⇒
   let [γ ⊢ IH1] = rpar [g] [γ ⊢ M1] in
   let [γ ⊢ IH2] = rpar [g] [γ ⊢ M2] in
       [γ ⊢ ap IH1 IH2];

rec rpar: {γ:ctx}{M : [γ ⊢ term]} predM [γ ⊢ M] [γ ⊢ M] =
mlam g ⇒ mlam M ⇒ case [γ ⊢ M] of
 | [γ ⊢ #p] ⇒ var (vp _)
 | [γ ⊢ lam \x.M'[…,x]] ⇒
    let h = rpar [g,x:term] [g,x:term ⊢ M'] in
    lm h
 | [γ ⊢ app M1 M2] ⇒
    let h1 = rpar [g] [γ ⊢ M1] in
    let h2 = rpar [g] [γ ⊢ M2] in
    ap h1 h2;
```

Figure 1: Reflexivity of `pred` vs `predM`

We have now a separate case for reducing variables, via the `isVar` judgment — recall that `#q` is a *parameter* variable, ranging over elements of the context. This is forced upon us by the usual positivity restriction on inductive types.

The more ominous consequence of this choice is that now establishing substitution properties for a given judgment requires a proof of *weakening*, and in turn the latter calls for a proof of context *exchange*, which is more delicate than expected. First, we must establish a clear and appropriate definition for determining that the ordering of a context is irrelevant : binary variable-swapping will suffice, although we will need to witness the swapping with a first-class substitution. Unfortunately, showing that reduction is stable under swapping, which basically amounts to equivariance, is a hassle: we have to "explain" to BELUGA's meta-logic what it means to be a variable. For the gory details, please see directory `Beta/beta_comp` in the repository. After that, the main proof proceeds smoothly in the usual way.

Is the switch to the meta-logic worth it in BELUGA? Not really. The elegance of Accattoli's approach stems from meta-level contexts in Abella being equivariant, and this preempts any issue with exchange. On the other hand, the idea works only for contexts that track "bare" bound variables, making it suitable just for (certain) untyped calculi. While BELUGA can easily overcome this via intrinsically typed terms, it seems that we have to rebuild a sort of de Bruijn infrastructure specific to each case study; furthermore, we had to struggle to prove that a substitutive judgment promoted to the meta-level is preserved by swapping. This seems too steep of a price for the mostly cosmetic improvements shown in Figure 1, which depicts the proof of reflexivity of parallel reduction in the two flavors.

3 Conclusions

Beluga's support for HOAS, paired with sophisticated context reasoning, makes the development of the traditional proof of CR(β) very elegant and devoid of technical lemmas foreign to the mathematics of the problem. Since specifications can be dependently-typed, the extension of the proof from the untyped to the typed case is conservative. It would be easy to encode other proof techniques such as establishing confluence for the simply typed case using Newman's lemma and existent SN proofs in BELUGA [1] or other classical results such as η-postponement, standardization, and residuals theory as in [2].

On the flip side, BELUGA (and LF) not allowing quantification over relations prevents us from a more succinct development via abstract rewriting system as per Nipkow's account, see the repetitions around the Commutation lemma. Combining natively HOAS and a full Agda-like type theory is under active research [21]. Reader, you may expect more Church-Rosser proofs in the future.

Acknowledgments Thanks to Brigitte Pientka for her help with the development.

References

[1] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer & Kathrin Stark (2019): *POPLMark reloaded: Mechanizing proofs by logical relations*. *J. Funct. Program.* 29, p. e19, doi:10.1017/S0956796819000170.

[2] Beniamino Accattoli (2012): *Proof Pearl: Abella Formalization of λ-Calculus Cube Property*. In Chris Hawblitzel & Dale Miller, editors: *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, *Lecture Notes in Computer Science* 7679, Springer, pp. 173–187, doi:10.1007/978-3-642-35308-6_15.

[3] Nick Benton, Chung-Kil Hur, Andrew Kennedy & Conor McBride (2012): *Strongly Typed Term Representations in Coq*. *J. Autom. Reason.* 49(2), pp. 141–159, doi:10.1007/s10817-011-9219-0.

[4] Roberto Blanco, Dale Miller & Alberto Momigliano (2019): *Property-Based Testing via Proof Reconstruction*. In Ekaterina Komendantskaya, editor: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, ACM, pp. 5:1–5:13, doi:10.1145/3354166.3354170.

[5] Arthur Charguéraud: *Locally nameless representation with cofinite quantification*. https://www.chargueraud.org/softs/ln. Accessed: November 2023.

[6] Ernesto Copello, Nora Szasz & Álvaro Tasistro (2021): *Formalization of metatheory of the Lambda Calculus in constructive type theory using the Barendregt variable convention*. *Math. Struct. Comput. Sci.* 31(3), pp. 341–360, doi:10.1017/S0960129521000335.

[7] Amy P. Felty, Alberto Momigliano & Brigitte Pientka (2015): *The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations - Part 2 - A Survey*. *J. Autom. Reason.* 55(4), pp. 307–372, doi:10.1007/s10817-015-9327-3.

[8] Amy P. Felty, Alberto Momigliano & Brigitte Pientka (2018): *Benchmarks for reasoning with syntax trees containing binders and contexts of assumptions*. *Math. Struct. Comput. Sci.* 28(9), pp. 1507–1540, doi:10.1017/S0960129517000093.

[9] Jonathan M. Ford & Ian A. Mason (2001): *Operational Techniques in PVS - A Preliminary Evaluation*. In Colin J. Fidge, editor: *Computing: The Australasian Theory Symposium, CATS 2001, Gold Coast, Australia, January 29-30, 2001*, *Electronic Notes in Theoretical Computer Science* 42, Elsevier, pp. 124–142, doi:10.1016/S1571-0661(04)80882-X.

[10] Gérard P. Huet (1994): *Residual Theory in lambda-Calculus: A Formal Development*. *J. Funct. Program.* 4(3), pp. 371–394, doi:10.1017/S0956796800001106.

[11] Jonas Kaiser, Brigitte Pientka & Gert Smolka (2017): *Relating System F and Lambda2: A Case Study in Coq, Abella and Beluga*. In Dale Miller, editor: *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*, *LIPIcs* 84, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 21:1–21:19, doi:`10.4230/LIPIcs.FSCD.2017.21`.

[12] Leonidas Lampropoulos & Benjamin C. Pierce (2023): *QuickChick: Property-Based Testing in Coq*. `https://softwarefoundations.cis.upenn.edu/qc-current`. Accessed: November 2023.

[13] Dan Licata (2008): *Church-Rosser theorem for β-reduction via complete development in Twelf*. `http://twelf.org/wiki/Church-Rosser_via_complete_development`. Accessed: November 2023.

[14] James McKinna & Robert Pollack (1993): *Pure Type Systems Formalized*. In Marc Bezem & Jan Friso Groote, editors: *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, *Lecture Notes in Computer Science* 664, Springer, pp. 289–305, doi:`10.1007/BFb0037113`.

[15] Alberto Momigliano (2000): *Elimination of Negation in a Logical Framework*. In Peter Clote & Helmut Schwichtenberg, editors: *Computer Science Logic, 14th Annual Conference of the EACSL, Fischbachau, Germany, August 21-26, 2000, Proceedings*, *Lecture Notes in Computer Science* 1862, Springer, pp. 411–426, doi:`10.1007/3-540-44622-2_28`.

[16] Julian Nagele, Vincent van Oostrom & Christian Sternagel (2016): *A Short Mechanized Proof of the Church-Rosser Theorem by the Z-property for the $\lambda\beta$-calculus in Nominal Isabelle*. *CoRR* abs/1609.03139. Available at `http://arxiv.org/abs/1609.03139`.

[17] Tobias Nipkow (2001): *More Church-Rosser Proofs*. *J. Autom. Reason.* 26(1), pp. 51–66, doi:`10.1023/A:1006496715975`.

[18] Frank Pfenning (1992): *A Proof of the Church-Rosser Theorem and its Representation in a Logical Framework*. Technical Report, Carnegie Mellon University. Tech. Rep. CMU-CS-92-186.

[19] Brigitte Pientka: *Adaptation to Beluga of Pfenning's proof of the Church-Rosser theorem for β-reduction*. `https://github.com/Beluga-lang/Beluga/tree/master/examples/church-rosser`. Accessed: November 2023.

[20] Brigitte Pientka & Andrew Cave (2015): *Inductive Beluga: Programming Proofs*. In Amy P. Felty & Aart Middeldorp, editors: *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, *Lecture Notes in Computer Science* 9195, Springer, pp. 272–281, doi:`10.1007/978-3-319-21401-6_18`.

[21] Brigitte Pientka, David Thibodeau, Andreas Abel, Francisco Ferreira & Rébecca Zucchini (2019): *A Type Theory for Defining Logics and Proofs*. In: *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, IEEE, pp. 1–13, doi:`10.1109/LICS.2019.8785683`.

[22] Barry K. Rosen (1973): *Tree-Manipulating Systems and Church-Rosser Theorems*. *J. ACM* 20(1), pp. 160–187, doi:`10.1145/321738.321750`.

[23] Steven Schäfer, Tobias Tebbi & Gert Smolka (2015): *Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions*. In Christian Urban & Xingyuan Zhang, editors: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, *Lecture Notes in Computer Science* 9236, Springer, pp. 359–374, doi:`10.1007/978-3-319-22102-1_24`.

[24] Peter Selinger (2013): *Lecture notes on the lambda calculus*. arXiv:`0804.3434`.

[25] Natarajan Shankar (1988): *A mechanical proof of the Church-Rosser theorem*. *J. ACM* 35(3), pp. 475–522, doi:`10.1145/44483.44484`.

[26] Masako Takahashi (1995): *Parallel Reductions in lambda-Calculus*. *Inf. Comput.* 118(1), pp. 120–127, doi:`10.1006/inco.1995.1057`.

[27] René Vestergaard & James Brotherston (2001): *A Formalised First-Order Confluence Proof for the lambda-Calculus Using One-Sorted Variable Names*. In Aart Middeldorp, editor: *Rewriting Techniques and Appli-*

cations, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings, *Lecture Notes in Computer Science* 2051, Springer, pp. 306–321, doi:10.1007/3-540-45127-7_23.

A Appendix: Counterexamples Search

The theory of confluence is rife with counterexamples. It is a well-known fact that (single step) β-reduction does *not* satisfy the diamond property, since redexes can be discarded or duplicated, and this is why the notion of parallel reduction is so useful. It would be nice, both for research and educational purposes, for a proof environment to assist us in *refuting* unprovable conjectures and witnessing such counter-examples. A lightweight approach to this endeavor is *property-based testing* (PBT), see for example the integration of *QuickChick* within Coq [12]. If we view a property as a logical formula $\forall x : \tau.P(x) \supset Q(x)$, providing a counter-example consists of negating the property, and searching for a proof of $\exists x : \tau.P(x) \wedge \neg Q(x)$. This points to a logic programming solution, where the specification is a fixed set of assumptions and the negated property is the goal. A full proof-theoretic reconstruction of PBT has been presented in [4] and can be adapted to BELUGA, which in the Twelf's tradition has a logic programming engine built-in.

To exemplify, let us search for a counter-example to diamond(β). First, we need to state the conjecture that we want to test, namely the negation of $M_1 \leftarrow M \rightarrow M_2$ entails $\exists N$, $M_1 \rightarrow N \leftarrow M_2$. This needs a little care, since BELUGA does not have negation — the usual solution in logic programming, i.e. *negation-as-failure*, is incompatible with BELUGA's foundations: indeed, what could be the proof term witnessing a proof failure? One solution [15] is to state in the positive when two terms are *non*-joinable: i.e., when they are different and if they reduce in one step, they do not reduce to the same term. This requires a notion of *inequality* for lambda terms (simplified here from the actual code):

```
LF diff : term → term → type =
 | dal : diff (lam _) (app _ _)
 | dla : diff (app _ _) (lam _)
 | da1 : diff E1 F1 → diff (app E1 E2) (app F1 F2)
 | da2 : diff E2 F2 → diff (app E1 E2) (app F1 F2)
 | dll : ({x:term} → diff (M x) (N x)) → diff (lam M) (lam N);

LF not_joinable : term → term → type =
| nj : diff M1 M2 → step M1 P1 → step M2 P2 → diff P1 P2 → not_joinable M1 M2;
```

The second ingredient is a *generator* for terms. For the sake of this paper we do not implement the full architecture of [4], but simply program an exhaustive height-bounded term generator. We also show the test harness predicate, combining generation with the to-be-tested conjecture:

```
LF heigth : nat → term → type =
 | h1 : heigth H M → heigth H N → heigth (s H) (app M N)
 | h2 : ({x:term} ({h:nat} heigth h x) → heigth H (M x)) → heigth (s H) (lam M);

LF gencex : nat → term → term → term → type =
| cx : not_joinable M1 M2 → step M M1 → step M M2 → hei I M → gencex I M M1 M2;
```

A query to BELUGA's logic programming engine with a bound of 3 will generate the following (pretty-printed) counterexample to diamond(β), namely M $=(\lambda x.\, x\, x)(I\, I)$, for I the identity combinator. In fact, it steps to $(I\, I)(I\, I)$ and $(\lambda x.\, x\, x)\, I$.

Another application is witnessing the failure of diamond(η) in a typed calculus with unit and surjective pairing. Here we exploit intrinsically-typed terms to encode typed η-reduction and obtain well-typed term generators for free. All the details in the repository, under directory PBT.

Embedding Differential Dynamic Logic in PVS

J. Tanner Slagel
NASA Langley Research Center
Hampton, VA, 23666, USA
j.tanner.slagel@nasa.gov

Mariano Moscato
National Institute of Aerospace*
Hampton, VA, 23666, USA

Lauren White
NASA Langley Research Center
Hampton, VA, 23666, USA

César A. Muñoz
NASA Langley Research Center
Hampton, VA, 23666, USA

Swee Balachandran
National Institute of Aerospace*
Hampton, VA, 23666, USA

Aaron Dutle
NASA Langley Research Center
Hampton, VA, 23666, USA

Differential dynamic logic (dL) is a formal framework for specifying and reasoning about hybrid systems, i.e., dynamical systems that exhibit both continuous and discrete behaviors. These kinds of systems arise in many safety- and mission-critical applications. This paper presents a formalization of dL in the Prototype Verification System (PVS) that includes the semantics of hybrid programs and dL's proof calculus. The formalization embeds dL into the PVS logic, resulting in a version of dL whose proof calculus is not only formally verified, but is also available for the verification of hybrid programs within PVS itself. This embedding, called Plaidypvs (**P**roper**l**y **A**ssured **I**mplementation of **dL** for H**y**brid **P**rogram **V**erification and **S**pecification), supports standard dL style proofs, but further leverages the capabilities of PVS to allow reasoning about entire classes of hybrid programs. The embedding also allows the user to import the well-established definitions and mathematical theories available in PVS.

1 Introduction

Systems that exhibit both discrete and continuous dynamics, known as *hybrid systems*, have emerged in numerous safety- and mission-critical applications such as avionics systems, robotics, medical devices, railway operations, and autonomous vehicles. To formally reason about these systems, it is often useful to model them as *hybrid programs* (HPs), where the discrete variables evolve through assignments like traditional imperative programs and the continuous variables are defined by a system of differential equations. Hybrid programs are suitable to model complex dynamics where the continuous and discrete dynamics are largely intertwined, but due to their complexity, efficient and effective formal reasoning about properties of such programs can be a challenge.

Differential dynamic logic (dL) enables the specification and reasoning of HPs using a small set of proof rules [42, 44, 50, 52]. Conceptually dL can be split into two parts: (1) a framework for the logical specifications of HPs and their properties and (2) a proof calculus that is a collection of axioms and deductive rules for reasoning about these logical specifications. The KeYmaera X[1] theorem prover is a software implementation of dL built up from a small, trusted core that assumes the axioms of dL [15, 26, 22] with a web-based interface for specification and reasoning of HPs [25]. KeYmaera X has been used in the formal verification of several cyber-physical systems [18, 24, 20, 6, 5, 16, 23, 29].

This paper presents a formal embedding of dL in the Prototype Verification System (PVS). PVS is a proof assistant that integrates a fully typed functional specification language supporting predicate subtypes and dependent types with an interactive theorem prover based on higher order logic. PVS

*Institute at time of contribution.
[1]https://keymaerax.org

allows users to write formal specifications and reason about them using a collection of built-in proof rules and user-defined proof strategies. Strategies are built on top of proof rules in a conservative way so that they do not introduce additional soundness concerns. Formal PVS developments are structured in theories and a collection of theories form a library. The NASA PVS Library (NASALib)[2] is a collection of formal developments contributed by the PVS community and maintained by the Formal Methods Team at NASA Langley Research Center. Currently, it consists of over $38,000$ proven lemmas spanning across 69 folders related to a wide range of topics in mathematics, logic, and computer science. The work presented in this paper relies on and contributes to NASALib.

The primary contribution of this work is a formal development called *Plaidypvs* (**P**roper**l**y **A**ssured **I**mplementation of **D**ifferential Dynamic Logic for H**y**brid **P**rogram **V**erification and **S**pecification), which is publicly available as part of NASALib[3]. Plaidypvs includes the *specification* of dL's HPs and their properties through an embedding in the PVS specification language, the *verification* of correctness of dL's axioms and deductive rules, and the *implementation* of these rules through the strategy language of PVS, resulting in a formally verified and interactive implementation of the proof calculus of dL within PVS.

While reasoning about HPs using a formally verified implementation of dL is already an achievement, the integration in PVS brings additional opportunities for extending the functionality of dL beyond what is available in a stand-alone dL system such as KeYmaera X. For example, new or existing functions and definitions in PVS can be used inside of the dL framework. This includes trigonometric and other transcendental functions already specified in NASAlib, as well as the corresponding properties concerning their derivatives and integrals. In addition, meta-reasoning about HPs and their properties can be performed in PVS using the dL embedding. Examples include specifying HPs with a parametric number of variables, which can be used to reason about situations with an unknown but finite number of actors, and reasoning about entire classes of HPs, which can be specified using the PVS type system.

The rest of this paper proceeds as follows. Section 2 details the formal development of HP specifications in Plaidypvs. Section 3 gives an overview of the formal verification approach to prove dL statements in PVS, as well the implementation of the proof calculus of dL in the PVS prover interface. Section 4 shows an example of utilizing the features of Plaidypvs beyond the capabilities of dL alone. Related work is discussed in 5. Finally, conclusions and future work are discussed in 6.

2 Specification of hybrid programs

This section describes the syntax, semantics, and logical specifications of HPs developed in Plaidypvs. Before these are introduced, a few preliminary concepts are needed.

2.1 Environment, real expressions, Boolean expressions

Hybrid programs manipulate real number values using discrete and continuous operations. At any moment, the state of a hybrid program is given by an environment of type $\mathscr{E} \triangleq [\mathbb{V} \to \mathbb{R}]$ that maps program variables in $\mathbb{V}$ to real number values in $\mathbb{R}$, where $\mathbb{V}$ is an infinite, but enumerable set of variables and $\mathbb{R}$ is the set of real numbers. For simplicity, variables are represented by indices, i.e., $\mathbb{V}$ is just the set of natural numbers.

[2]https://github.com/nasa/pvslib

[3]https://github.com/nasa/pvslib/tree/master/dL

The sets $\mathscr{R}$ and $\mathscr{B}$ of real and Boolean hybrid program expressions, respectively, are defined by a shallow embedding meaning they are represented by their evaluations functions, i.e., $\mathscr{R} \triangleq [\mathscr{E} \to \mathbb{R}]$ and $\mathscr{B} \triangleq [\mathscr{E} \to \mathbb{B}]$. For instance, $\mathbf{cnst}(c) \triangleq \lambda(e : \mathscr{E}).c$ represents the constant expression that returns the value $c \in \mathbb{R}$ in any environment and $\mathbf{val}(v) \triangleq \lambda(e : \mathscr{E}).e(v)$ represents the real expression that returns the value of variable v in the environment e. Similarly, $\top \triangleq \lambda(e : \mathscr{E}).\mathbf{True}$ and $\bot \triangleq \lambda(e : \mathscr{E}).\mathbf{False}$ represent the Boolean hybrid program constants that always return $\mathbf{True} \in \mathbb{B}$ and $\mathbf{False} \in \mathbb{B}$, respectively. While real and Boolean expressions can be arbitrary functions, Plaidypvs provides support for standard arithmetic and Boolean operators by lifting them to the domain of $\mathscr{R}$ and $\mathscr{B}$. Given $r, r_1, r_2 \in \mathscr{R}$ and $n \in \mathbb{N}$ the following are recognized to be of type $\mathscr{R}$: $r_1 + r_2$, $r_1 - r_2$, r_1/r_2, $r_1 \cdot r_2$, $r_1 = r_2$, $-r$, $\sqrt{r}$, and r^n. It is important to notice that, for instance, in the real expression $r_1 + r_2$, the operator $+$ is not the arithmetic addition, but it is of type $\mathscr{R} \times \mathscr{R} \to \mathscr{R}$. Similarly, given Boolean expressions $b, b_1, b_2 \in \mathscr{B}$, the following are recognized to be of type $\mathscr{B}$: $b_1 \wedge b_2$, $b_1 \vee b_2$, $b_1 \to b_2$, $b_1 \leftrightarrow b_2$, and $\neg b$.

Example 2.1 (Environments, Real and Boolean Expressions) *Let $x, y \in \mathbb{V}$ and $c \in \mathbb{R}_{\geq 0}$, the following Boolean expression denotes a circle of radius c centered at $(0,0)$:*

$$\boldsymbol{val}(x)^2 + \boldsymbol{val}(y)^2 = \boldsymbol{cnst}(c)^2. \tag{1}$$

*Furthermore, assuming the environment $e \triangleq (\lambda(v : \mathbb{V}).0)$ **with** $\{x \mapsto c/2, y \mapsto \sqrt{3} \cdot c/2\}$, the following Boolean statement holds.*

$$(\boldsymbol{val}(x)^2 + \boldsymbol{val}(y)^2 = \boldsymbol{cnst}(c)^2)(e) = \boldsymbol{True}.$$

Henceforth, for ease of presentation, the **val** and **cnst** operators are suppressed in much of the remainder of the paper. The Boolean expression in Formula 1, for example, will be presented instead as $x^2 + y^2 = c^2$.

2.2 Hybrid programs

Hybrid programs are syntactically defined as a datatype $\mathscr{H}$ in PVS according to the following grammar.

$$\alpha ::= \mathbf{x} := \ell \mid \mathbf{x}' = \ell \& P \mid ?P \mid x := * \mid \alpha_1 ; \alpha_2 \mid \alpha_1 \cup \alpha_2 \mid \alpha_1^*.$$

Here, $\mathbf{x} := \ell$ is a list of pairs in $\mathbb{V} \times \mathscr{R}$, where the first entries are unique, intended to represent a discrete assignment of the variables indexed by these first elements. The differential equation $\mathbf{x}' = \ell \& P$, where $\mathbf{x}' = \ell$ is another such list in $\mathbb{V} \times \mathscr{R}$ and $P \in \mathscr{B}$ is a Boolean expression, is meant to symbolize the continuous evolution of the variables in $\mathbf{x}'$ according to the first order differential equation described by ℓ. Note that use of the symbol & is distinct from Boolean conjunction and is used here purely syntactically to represent that the solution of the differential equation satisfies P along the evolution. To reference a variable used in a discrete assignment or differential equation, the notation $v \in \mathbf{x}$ (respectively, $v \in \mathbf{x}'$) will be used. The real expression associated with v in ℓ will be denoted $\ell(v)$. The program $?P$ represents a check of the Boolean expression P. The program $x := *$ represents a discrete assignment of the variable x to an arbitrary real number value. The program $\alpha_1 ; \alpha_2$ represents the sequential execution of the subprograms α_1 and α_2, while $\alpha_1 \cup \alpha_2$ symbolizes a nondeterministic choice between two subprograms. Finally, α_1^* represents repetition of a HP a finite but unknown (possibly zero) number of times.

Formally, the predicate **s_rel** defines the semantic relation of a hybrid program α with respect to input and output environments $e_i, e_o \in \mathscr{E}$. It is inductively defined on α as follows.

$$\mathbf{s\_rel}(\alpha)(e_i)(e_o) \triangleq \begin{cases} \forall k : k \notin \mathbf{x} \to e_o(k) = e_i(k) & \text{if } \alpha = (\mathbf{x} := \ell), \\ \quad \wedge k \in \mathbf{x} \to e_o(k) = \ell(k)(e_i) & \\ e_o = e_i \vee & \text{if } \alpha = (\mathbf{x}' = \ell \& P), \\ \quad \exists D : \mathbf{s\_rel\_diff}(D, \mathbf{x}', \ell, P, e_i, e_o) & \\ e_o = e_i \wedge P(e_i) & \text{if } \alpha = ?P, \\ \exists r : e_o(x) = r \wedge Q(r)(e_i) & \text{if } \alpha = (x := * \& Q), \\ \exists e : \mathbf{s\_rel}(\alpha_1)(e_i)(e) & \text{if } \alpha = \alpha_1 ; \alpha_2, \\ \quad \wedge \mathbf{s\_rel}(\alpha_2)(e)(e_o) & \\ \mathbf{s\_rel}(\alpha_1)(e_i)(e_o) & \text{if } \alpha = \alpha_1 \cup \alpha_2, \\ \vee \mathbf{s\_rel}(\alpha_2)(e_i)(e_o) & \\ e_o = e_i \vee & \text{if } \alpha = \alpha_1^*. \\ \quad \exists e : \mathbf{s\_rel}(\alpha_1)(e_i)(e) & \\ \quad\quad \wedge \mathbf{s\_rel}(\alpha)(e)(e_o) & \end{cases}$$

The correspondence between the informal description of semantics and the **s_rel** function is standard in all cases except the differential equation branch. For differential equations, the domain D is $\mathbb{R}_{\geq 0}$, or some closed interval starting at 0, and the semantics is given by the following function.

$$\begin{aligned} \mathbf{s\_rel\_diff}(D, \mathbf{x}', \ell, P, e_i, e_o) \triangleq\ & \exists r : \exists! f : D(r) \wedge \mathbf{sol?}(D, \mathbf{x}', \ell, e_i)(f) \wedge \\ & e_o = \mathbf{e\_at\_t}(\mathbf{x}', \ell, f, e_i)(r) \wedge \\ & \forall t : (D(t) \wedge t \leq r) \\ & \quad \to P(\mathbf{e\_at\_t}(\mathbf{x}', \ell, f, e_i)(t)). \end{aligned}$$

Unpacking this further,

$$\mathbf{e\_at\_t}(\mathbf{x}', \ell, f, e_i) \triangleq \lambda(r : \mathbb{R}). \lambda(j : \mathbb{V}). \begin{cases} e_i(j) & \text{if } j \notin \mathbf{x}', \\ f(j)(r) & \text{if } j \in \mathbf{x}', \end{cases}$$

is a function that characterizes the environment e_i, with the continuously evolving variables $\mathbf{x}'$ replaced by values from a function $f : [\mathbb{R}^k \to [\mathbb{R} \to \mathbb{R}]]$. The definition

$$\begin{aligned} \mathbf{sol?}(D, \mathbf{x}', \ell, e_i)(f) \triangleq\ & \forall (i \in \mathbf{x}', t \in \mathbf{D}) : \\ & (f(i))'(t) = \ell(i)(\mathbf{e\_at\_t}(\mathbf{x}', \ell, f, e_i)(t)) \end{aligned}$$

ensures that f is the solution to the k-dimensional differential equation $\mathbf{x}' = \ell$ throughout the domain D. Note in the definition of **s_rel_diff** this solution f is further assumed to be unique on the domain D.

Below is a colloquial description of the semantics of each type of HPs, where e_i, e_o are the input and output environments, respectively.

$\mathbf{x} := \ell$	Discrete variable assignment. This means that e_i and e_o agree on all the variables not mentioned in ℓ, and for the variables in ℓ, a discrete jump has taken place.
$\mathbf{x}' := \ell \& P$	Continuous variable assignment. Continuous jumps take place where the output variable that is included in ℓ has evolved according to the first order differential equation defined in ℓ. The solution to the differential equation satisfies P.

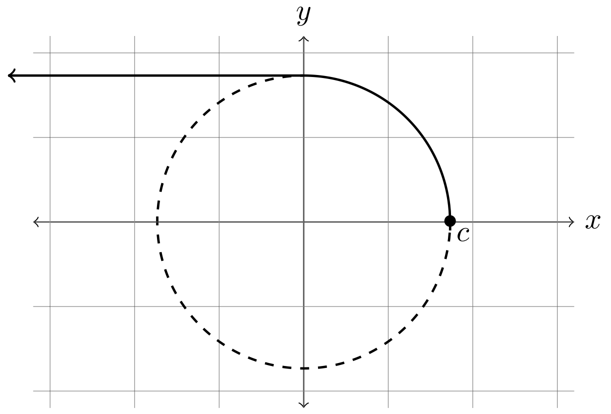


Figure 1: Dubins path modeling an aircraft turning.

$?P$	Test HP. An input/output pair is related only if they are equal and e_i satisfies P.
$x := *$	Random discrete assignment. Random assignment of variable x, where the random assignment is some value r and $e_o(n) = r$.
$\alpha_1;\ \alpha_2$	Sequential HP. Runs two HPs α_1 and α_2 in order such that there is an environment e that is semantically related to e_i through α_1, and semantically related to e_o through α_2.
$\alpha_1 \cup \alpha_2$	Nondeterministic choice HP. This HP nondeterministically chooses one of α_1 or α_2. Here e_o is semantically related to e_i through α_1 or α_2.
α^*	Loop HP. This is the repeat of HP α_1 a finite but undisclosed number of times. The environment e_o is either equal to e_i or is it semantically related to another environment e through α and e is semantically related to e_o through α^*.

Example 2.2 (HP) *The hybrid program*

$$\begin{gathered}((?(x>0);(x'=-y,y'=x\ \&x\geq 0))\cup\\ (?(x\leq 0);(x'=-c,y'=0)))^*,\end{gathered}$$

where $x,y \in \mathbb{V}$, $x \neq y$, and $c \in \mathbb{R}$, represents the dynamic systems where x and y progress according to the differential equation $x' = -y$, $y' = x$ when $x > 0$, but when $x \leq 0$ the variables progress according to the differential equation $x' = -c$, $y' = 0$. Note that the test statements, introduced by the operator $?$, determine which branch of $\cup$ in the HP is applicable, and the domain $x \geq 0$ in the first differential equation prevents the dynamics from continuing when $x = 0$, forcing the other branch of the HP to take place. The operator $*$ allows repetition so that both branches of the dynamics are carried out.

The hybrid program in Example 2.2 will be used as running example through this paper. It models a Dubins curve representing the trajectory of an aircraft turning and then proceeding in a straight line (see Figure 1).

2.3 Quantified statements about hybrid programs

A hybrid program can have potentially many different executions or *runs*. This means that given an input environment e_i, there may be infinitely many output environments e_o semantically related to it (by repetition, random assignment, etc.). To reason about these runs, universal and existential quantifiers over the potentially infinite number of executions of an HP are defined. These quantifies are called *allruns*, denoted $[\cdot]$, and *someruns*, denoted $\langle\cdot\rangle$. For $\alpha \in \mathscr{H}$ and $P \in \mathscr{B}$, $[\alpha]P \in \mathscr{B}$ is defined as follows.

$$[\alpha]P \triangleq \lambda(e_i : \mathscr{E}).\forall e_o : \mathbf{s\_rel}(\alpha)(e_i)(e_o) \to P(e_o),$$

Analogously, $\langle\alpha\rangle P \in \mathscr{B}$ is defined as follows.

$$\langle\alpha\rangle P \triangleq \lambda(e_i : \mathscr{E}).\exists e_o : \mathbf{s\_rel}(\alpha)(e_i)(e_o) \wedge P(e_o).$$

These quantifiers state that every (some, respectively) run of the HP α starting at environment e_i and ending at environment e_o satisfies P.

Example 2.3 (Allruns) *Let α be the HP in Example 2.2, $\boldsymbol{circ}(c) \triangleq x^2 + y^2 = c^2$ and*

$$\boldsymbol{path}(c) \triangleq (x > 0 \to \boldsymbol{circ}(c)) \wedge (x \leq 0 \to y = c).$$

Then, the Boolean expression

$$(x = c \wedge y = 0) \to [\alpha]\boldsymbol{path}(c), \tag{2}$$

states that if the value of x is c and the value of y is 0*, then for all runs of the HP α, the values of x and y stay inside $\boldsymbol{path}(c)$. In other words, x and y stay on the circle of radius c until $x = 0$ and then stay on the line $y = c$.*

3 Embedding differential dynamic logic

With the formal specification of hybrid programs established, the embedding of the sequent calculus of dL in PVS can be discussed. First, dL-sequents will be defined, then a description of the formal verification process encoding the axioms and rules of dL as lemmas in PVS is provided.

3.1 dL-sequents

A dL-sequent is denoted $\Gamma \vdash \Delta$, where Γ and Δ, known as the *antecedent* and the *consequent*, respectively, are lists of Boolean expressions. In PVS, a dL-sequent is defined by

$$\Gamma \vdash \Delta \triangleq \forall e \in \mathscr{E} : \bigwedge \Gamma(e) \implies \bigvee \Delta(e),$$

where $\implies$ is the PVS implication. Intuitively, this means that the conjunction of the antecedent formulas implies the disjunction of the consequent formulas.

The dL approach for proving statements about hybrid programs relies on a set of deductive rules of the form

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \ldots \quad \Gamma_k \vdash \Delta_k}{\Gamma \vdash \Delta.}$$

Rules without hypothesis, i.e., where $k = 0$, are called *axioms*. A rule of this form states that the conjunction of the sequents above the inference line implies the sequent below the inference line. When proving statements, these rules are used in a bottom-up fashion forming an inverted (proof) tree, where the root of the tree is the sequent to be proven, branches are related by instances of deductive rules, and leaves are instances of axioms.

To formally verify dL each rule of dL is specified as a PVS lemma, which takes essentially the following form.

$$\textbf{notR}\ \frac{\Gamma, P \vdash \Delta}{\Gamma \vdash \neg P, \Delta} \qquad \textbf{impliesR}\ \frac{\Gamma, P \vdash Q, \Delta}{\Gamma \vdash P \rightarrow Q, \Delta}$$

$$\textbf{notL}\ \frac{\Gamma \vdash P, \Delta}{\Gamma, \neg P \vdash \Delta} \qquad \textbf{impliesL}\ \frac{\Gamma \vdash P, \Delta \quad \Gamma, Q \vdash \Delta}{\Gamma, P \rightarrow Q \vdash \Delta}$$

$$\textbf{andR}\ \frac{\Gamma \vdash P, \Delta \quad \Gamma \vdash Q, \Delta}{\Gamma \vdash P \wedge Q, \Delta} \qquad \textbf{iffR}\ \frac{\Gamma, P \vdash Q, \Delta \quad \Gamma, Q \vdash P, \Delta}{\Gamma \vdash P \leftrightarrow Q, \Delta}$$

$$\textbf{andL}\ \frac{\Gamma, P, Q \vdash \Delta}{\Gamma, P \wedge Q \vdash \Delta} \qquad \textbf{iffL}\ \frac{\Gamma, P \wedge Q \vdash \Delta \quad \Gamma, \neg P \wedge \neg Q \vdash \Delta}{\Gamma, P \leftrightarrow Q \vdash \Delta}$$

$$\textbf{orR}\ \frac{\Gamma \vdash P, Q, \Delta}{\Gamma \vdash P \vee Q, \Delta} \qquad \textbf{falseL}\ \frac{}{\Gamma, \bot \vdash \Delta}$$

$$\textbf{orL}\ \frac{\Gamma, P \vdash \Delta \quad \Gamma, Q \vdash \Delta}{\Gamma, P \vee Q \vdash \Delta} \qquad \textbf{trueR}\ \frac{}{\Gamma \vdash \top, \Delta}$$

$$\textbf{cut}\ \frac{\Gamma \vdash C, \Delta \quad \Gamma, C \vdash \Delta}{\Gamma \vdash \Delta} \qquad \textbf{axiom}\ \frac{}{\Gamma, P \vdash P, \Delta}$$

$$\textbf{weakR}\ \frac{\Gamma \vdash P, \Delta \quad P \vdash Q}{\Gamma \vdash Q, \Delta} \qquad \textbf{weakL}\ \frac{P, \Gamma \vdash \Delta \quad Q \vdash P}{\Gamma, Q \vdash \Delta}$$

Figure 2: Propositional dL rules

Lemma <**dL-rule-name**> *For all lists of Boolean expressions* Γ, Δ,

$$\bigwedge_{i=1}^{k} \Gamma_i \vdash \Delta_i \quad \Longrightarrow \quad \Gamma \vdash \Delta.$$

With such lemmas proven in PVS, a user can bring them into a proof environment and instantiate them as needed for proving a specific sequent. To automate this process, these lemmas are further implemented as *(proof) strategies* in PVS. These strategies parse the current sequent, identify instantiations that apply, hide unneeded formulas, and prove type-checking conditions that may appear, among other capabilities. More complex strategies are built on top of these strategies to simplify the proof process. Some of these rules, including details about their specification, verification, and implementation as strategies in PVS, are discussed below. A Plaidypvs "cheat sheet" is available for users with the development.[4]

3.2 Basic logical and structural rules of dL

The propositional rules in dL allow manipulation of the basic logical connectives ($\wedge$, $\vee$, $\neg$, $\rightarrow$, $\iff$) and operators ($\top$, $\bot$) in the dL-sequent (see Figure 2). For example, the rule **impliesR**, defined as

$$\frac{\Gamma, P \vdash Q, \Delta}{\Gamma \vdash P \rightarrow Q, \Delta},$$

allows an implication in the dL-consequent, $P \rightarrow Q$ to be simplified to P in the dL-antecedent and Q in the dL-consequent. Here, $\Gamma \vdash P \rightarrow Q, \Delta$ is the dL-sequent that **impliesR** can be applied to and $\Gamma, P \vdash Q, \Delta$ is the simplified dL-sequent. Note that the standard logical notation being used for **impliesR** above is for ease of presentation, whereas the PVS specification of such a rule, generally hidden from a user by a strategy, is closer to that described in Section 3.1. Additionally, there are quantification rules for

[4]https://github.com/nasa/pvslib/tree/master/dL/cheatsheet.pdf

existsR $$\frac{\Gamma \vdash p(e), \Delta}{\Gamma \vdash \exists x : p(x), \Delta} \text{ (any } e\text{)}$$

forallL $$\frac{\Gamma, p(e) \vdash \Delta}{\Gamma, \forall x : p(x) \vdash \Delta} \text{ (any } e\text{)}$$

forallR $$\frac{\Gamma \vdash p(y), \Delta}{\Gamma \vdash \forall x : p(x), \Delta} \text{ (} y \text{ Skolem symbol)}$$

existsL $$\frac{\Gamma, p(y) \vdash \Delta}{\Gamma, \exists x : p(x) \vdash \Delta} \text{ (} y \text{ Skolem symbol)}$$

Figure 3: Quantification dL rules

moveR $$\frac{\Gamma \vdash Q, P, \Delta}{\Gamma \vdash P, Q, \Delta}$$

hideR $$\frac{\Gamma \vdash \Delta}{\Gamma \vdash P, \Delta}$$

moveL $$\frac{\Gamma, Q, P \vdash \Delta}{\Gamma, P, Q \vdash \Delta}$$

hideL $$\frac{\Gamma \vdash \Delta}{\Gamma, P \vdash \Delta}$$

Figure 4: Structural dL rules

Skolemization and instantiation in the dL-sequent (see Figure 3) and there are structural rules that allow expressions to be moved or deleted (Figure 4).

In addition to the propositional, quantification, and structural rules, Plaidypvs provides a collection of powerful proof commands that combine the more basic dL strategies. A list these additional proof commands is given in Figure 5.

Example 3.1 (dL-sequent example) *The* dL*-sequent*

$$\vdash (x = c \wedge y = 0) \rightarrow [\alpha]\boldsymbol{path}(c).$$

expresses the validity of the expression in Formula 2 from Example 2.3. Invoking the rule ***dl-flatten*** *to the sequent above applies* ***impliesR*** *and* ***andL****, which separates conjunctions in the antecedent, resulting in the following* dL*-sequent:*

$$x = c, y = 0 \vdash [\alpha]\boldsymbol{path}(c). \tag{3}$$

3.3 Hybrid program rewriting rules

While the rules in Section 3.2 manipulate the logical structure of a dL-sequent, further rules act on the hybrid program components of such a sequent. Properties given in Figure 6 allow direct rewriting of hybrid programs. Other rules about hybrid programs in a sequent are given in Figure 7. Most of these rules manipulate the allruns $[\cdot]$ or someruns $\langle\cdot\rangle$ operators and the proofs were largely concerned with reasoning about the semantic relation function **s_rel** defined in Section 2. In addition to each of these rules becoming strategies, the command **dl-assert** uses all the hybrid program rewriting rules in Table 6 to simplify an expression.

There are a few intricacies worth mentioning in the formal verification and implementation of these rules in PVS. In the rewriting rules **assignb** and **assignd**, an allruns or someruns of an assignment HP is

dl-flatten Disjunctively simplifies the dL sequent by applying **trueR**, **falseL**, **orR**, **impliesR**, **notR**, **axiom**, **falseL**.

dl-ground Disjunctively and conjunctively simplifies the dL sequent by applying **dl-flatten** and additional splitting lemmas **andR**, **orL**, and **impliesL**.

dl-inst Instantiates a universal quantifier in the dL-antecedent by applying **forallL** or an existential quantifier in the dL-consequent by applying **existsL**.

dl-skolem Skolemizes an existential quantifier in dL-antecedent by applying **existsR** or a universal quantifier in the dL-consequent by applying **forallR**.

dl-grind Repeatedly uses **dl-ground** and **skolem** and serveral rewriting rules related to real expressions. This strategy has the option to use the MetiTarski automatic theorem prover as an outside oracle to discharge the proof if possible.

dl-assert Repeatedly applies hybrid program rewriting rules in Figure 6.

Figure 5: dL proof commands

equated to a substitution. Substitution is defined at the environment level as follows.

$$\textbf{assign\_sub}(\mathbf{x} := \ell)(e)(i) \triangleq \begin{cases} \ell(i)(e) & \text{if } i \in \mathbf{x} \\ e(i) & \text{if } i \notin \mathbf{x}. \end{cases} \tag{4}$$

Substitution of a general Boolean expression is therefore defined as follows.

$$\textbf{SUB}(\mathbf{x} := \ell)(P) \triangleq \lambda(e : \mathscr{E}).P(\textbf{assign\_sub}(\mathbf{x} := \ell)(e)).$$

While the definition of substitution above applies to any Boolean expression P and can be reasoned about by a user of Plaidypvs, the standard level of manipulation in dL is not often at the environment level. To increase the level of automation, several rewriting rules for reducing expressions containing **SUB** have been implemented. This led to formally verifying substitution properties for real expressions, inequalities of real expressions, and hybrid programs, so that a substitution at the top level of an expression could be pushed down to the level of **val** and **cnst**, where atomic substitutions are applied. The implementation of these rules required a calculus for reducing the substitution down to atomic expressions, written in the strategy language of PVS. This allows the **assignb** and **assignd** strategies to automatically compute a substitution for any propositional expression composed of equalities and inequalities of polynomial real expressions. For example, the substitution

$$\textbf{SUB}(x := y, y := 10)(x^2 + y^2 = 11),$$

is transformed automatically into $y^2 + 10^2 = 11$ as follows.

$$\begin{aligned}\textbf{SUB}(x := y, y := 10)(x^2 + y^2 = 11) &= \big(\textbf{SUB\_re}(x := y, y := 10)(x^2 + y^2) = \textbf{SUB\_re}(x := y, y := 10)(11)\big) \\ &= \big(\textbf{SUB\_re}(x := y, y := 10)(x^2) + \textbf{SUB\_re}(x := y, y := 10)(y^2) = 11\big) \\ &= \big(\textbf{SUB\_re}(x := y, y := 10)(x)^2 + \textbf{SUB\_re}(x := y, y := 10)(y)^2 = 11\big) \\ &= y^2 + 10^2 = 11,\end{aligned}$$

where **SUB_re** is substitution defined on real expressions $r \in \mathscr{R}$ as

$$\textbf{SUB\_re}(\mathbf{x} := \ell)(r) \triangleq \lambda(e : \mathscr{E}).r(\textbf{assign\_sub}(\mathbf{x} := \ell)(e)).$$

boxd	$\langle\alpha\rangle P \leftrightarrow \neg[\alpha]\neg P$
assignb	$[\mathbf{x} := \ell]\, P = \mathbf{SUB}(\mathbf{x} := \ell)(P)$
assignd	$\langle \mathbf{x} := \ell\rangle\, P = \mathbf{SUB}(\mathbf{x} := \ell)(P)$
testb	$[?Q]\, P = Q \to P$
testd	$\langle ?Q\rangle\, P = Q \wedge P$
choiceb	$[\alpha_1 \cup \alpha_2]\, P \leftrightarrow [\alpha_1]\, P \wedge [\alpha_2]\, P$
choiced	$\langle \alpha_1 \cup \alpha_2\rangle\, P \leftrightarrow \langle\alpha_1\rangle\, P \vee \langle\alpha_2\rangle\, P$
composeb	$[\alpha_1;\alpha_2]\, P \leftrightarrow [\alpha_1]\,[\alpha_2]\, P$
composed	$\langle\alpha_1;\alpha_2\rangle\, P \leftrightarrow \langle\alpha_1\rangle\,\langle\alpha_2\rangle\, P$
iterateb	$[\alpha^*]\, P = P \wedge [\alpha]\,[\alpha^*]\, P$
iterated	$\langle\alpha^*\rangle\, P = P \vee \langle\alpha\rangle\,\langle\alpha^*\rangle\, P$
anyb	$[x := *]\, P(x) = \forall x : P(x)$
anyd	$\langle x := *\rangle\, P(x) = \exists x : P(x)$

Figure 6: Hybrid program rewriting rules.

The automated substitution of more general Boolean expressions (for example, a statement of the form $[\alpha]\, P$) is still incomplete in Plaidypvs, and an area of future work.

Another challenge in formal verification occurs in some hybrid program rules. The **ghost**, **VRb**, and **VRd** rules require the concept of *freshness*. A fresh variable y is defined as

$$\mathbf{fresh?}(P)(y) \;\triangleq\; \forall e \in \mathscr{E},\, r \in \mathbb{R},\, P(e) = P(e \text{ with } y \mapsto r)].$$

In other words, the value of the Boolean expression P does not depend on the value of the variable y. Analogous definitions exist to express that a variable is fresh relative to a real expression or a hybrid program. Furthermore, an entire hybrid program can be checked for freshness relative to a Boolean expression as follows.

$$\mathbf{fresh?}(P)(\alpha) \;\triangleq\; \begin{cases} \forall k \in \mathbf{x}\;\; \mathbf{fresh?}(P)(k) & \text{if } \alpha = (\mathbf{x} := \ell), \\ \forall k \in \mathbf{x}'\;\; \mathbf{fresh?}(P)(k) & \text{if } \alpha = (\mathbf{x}' = \ell \,\&\, Q), \\ \mathbf{True} & \text{if } \alpha = ?Q, \\ \mathbf{fresh?}(P)(x) & \text{if } \alpha = (x := * \,\&\, Q), \\ \mathbf{fresh?}(P)(\alpha_1) \wedge \mathbf{fresh?}(P)(\alpha_2) & \text{if } \alpha = \alpha_1;\alpha_2, \\ \mathbf{fresh?}(P)(\alpha_1) \wedge \mathbf{fresh?}(P)(\alpha_2) & \text{if } \alpha = \alpha_1 \cup \alpha_2, \\ \mathbf{fresh?}(P)(\alpha) & \text{if } \alpha = \alpha_1^*. \end{cases} \tag{5}$$

Note that the recursive definition of freshness above ensures the value of P does not change for any run of the hybrid program α by checking if all the variables potentially changing in α are fresh relative to P.

The need for a fresh variable, as in the rule **ghost**, requires a mechanism for producing fresh variables relative to a dL-sequent. Since variables are represented by indices, a fresh variable can be generated by computing the smallest natural number in a dL-sequent not being used as a variable index. Plaidypvs also provides strategies for automatically proving freshness of variables in dL-sequents.

$$\textbf{Mb}\ \frac{\vdash P \to Q}{\Gamma \vdash [\alpha]P \to [\alpha]Q, \Delta}$$

$$\textbf{Md}\ \frac{\vdash P \to Q}{\Gamma \vdash \langle\alpha\rangle P \to \langle\alpha\rangle Q, \Delta}$$

$$\textbf{K}\ \frac{\Gamma \vdash [\alpha](P \to Q), \Delta}{\Gamma \vdash [\alpha]P \to [\alpha]Q, \Delta}$$

$$\textbf{loop}\ \frac{\Gamma \vdash J, \Delta \quad J \vdash [\alpha]J \quad J \vdash P}{\Gamma \vdash [\alpha^*]P, \Delta}$$

$$\textbf{mbR}\ \frac{\Gamma \vdash [\alpha]Q, \Delta \quad Q \vdash P}{\Gamma \vdash [\alpha]P, \Delta}$$

$$\textbf{mbL}\ \frac{\Gamma, [\alpha]Q \vdash \Delta \quad P \vdash Q}{\Gamma, [\alpha]P \vdash \Delta}$$

$$\textbf{ghost}\ \frac{\Gamma \vdash [y := e]P, \Delta}{\Gamma \vdash P, \Delta}\ \textbf{fresh?}(P)(y)$$

$$\textbf{Gb}\ \frac{\vdash P}{\Gamma \vdash [\alpha]P, \Delta}$$

$$\textbf{Gd}\ \frac{\vdash \langle\alpha\rangle\top \quad \vdash P}{\Gamma \vdash \langle\alpha\rangle P, \Delta}$$

$$\textbf{VRb}\ \frac{\Gamma \vdash P, \Delta}{\Gamma \vdash [\alpha]P, \Delta}\ \textbf{fresh?}(P)(\alpha)$$

$$\textbf{VRd}\ \frac{\vdash \langle\alpha\rangle\top \quad \Gamma \vdash P, \Delta}{\Gamma \vdash \langle\alpha\rangle P, \Delta}\ \textbf{fresh?}(P)(\alpha)$$

$$\textbf{mdR}\ \frac{\Gamma \vdash \langle\alpha\rangle Q, \Delta \quad Q \vdash P}{\Gamma \vdash \langle\alpha\rangle P, \Delta}$$

$$\textbf{mdL}\ \frac{\Gamma, \langle\alpha\rangle Q \vdash \Delta \quad P \vdash Q}{\Gamma, \langle\alpha\rangle P \vdash \Delta}$$

Figure 7: Hybrid program rules.

Example 3.2 (dL-sequent example continued) *Expanding α in the sequent given by Formula 3, from Example 3.1, and using **loop** with $J = (\textbf{\textit{path}}(c) \wedge y \geq 0)$ produces three subgoals,[5] one of which is*

$$\textbf{\textit{path}}(c), y \geq 0 \vdash \big[(?(x>0); (x' = -y, y' = x, \& x \geq 0)) \cup \\ (?(x \leq 0); (x' = -c, y' = 0))\big]\, \textbf{\textit{path}}(c) \wedge y \geq 0.$$

*Using **dl-assert** to simplify with hybrid program rewriting rules and applying propositional simplifications with the command **dl-ground** result in the following two dL-sequents*

$$\begin{aligned} (x > 0, \textbf{\textit{circ}}(c), y \geq 0) &\vdash \big[x' = -y, y' = x, \& x \geq 0))\big]\, \textbf{\textit{path}}(c) \wedge y \geq 0, \\ (x \leq 0, y = c) &\vdash \big[(x' = -c, y' = 0)\big]\, \textbf{\textit{path}}(c) \wedge y \geq 0. \end{aligned} \tag{6}$$

3.4 Rules for differential equations

The rules for differential equations are given in Figure 8. The differential equation rules required significant mathematical underpinnings to be added to PVS for their formal verification. For the implementation of the **dI** rule, a calculus to automatically compute the derivative of a Boolean expression P was necessary. To do this, an embedding of non-quantified Boolean expressions was developed as a data type with the following grammar.

$$b ::= b_1 \wedge_{nqB} b_2 \mid b_1 \vee_{nqB} b_2 \mid \neg_{nqB} b_1 \mid rel_{nqB}(r_1, r_2),$$

where rel_{nqB} is of type **NQB_rel**, which is itself an embedding of the following inequality operators.

$$rel_{nqB} ::= \ \leq_{nqB} \mid \ \geq_{nqB} \mid \ <_{nqB} \mid \ >_{nqB} \mid \ =_{nqB} \mid \ \neq_{nqB}.$$

[5]The other two dL-sequents generated can be proven easily. For full details of the examples in this paper, see the PVS implementation at https://github.com/nasa/pvslib/tree/master/dL/examples

$$\textbf{dinit}\ \frac{\Gamma, Q \vdash [x' = f(x) \,\&\, Q]\, P, \Delta}{\Gamma \vdash [x' = f(x) \,\&\, Q]\, P, \Delta}$$

$$\textbf{dW}\ \frac{Q \vdash P}{\Gamma \vdash [x' = f(x) \,\&\, Q]\, P, \Delta}$$

$$\textbf{dI}\ \frac{\Gamma, Q \vdash P, \Delta \quad Q \vdash [x' := f(x)]\,(P)'}{\Gamma \vdash [x' = f(x) \,\&\, Q]\, P, \Delta}$$

$$\textbf{dC}\ \frac{\Gamma \vdash [x' = f(x) \,\&\, Q]\, C, \Delta \quad \Gamma \vdash [x' = f(x) \,\&\, (Q \wedge C)]\, P, \Delta}{\Gamma \vdash [x' = f(x) \,\&\, Q]\, P, \Delta}$$

$$\textbf{dG}\ \frac{\Gamma \vdash G,\ G \vdash P,\ \Gamma \vdash \exists y\, [x' = f(x),\, y' = a(x) \cdot y + b(x) \,\&\, Q]\, G, \Delta}{\Gamma \vdash [x' = f(x) \,\&\, Q]\, P, \Delta}\ \textbf{fresh?}(y)$$

$$\textbf{dS}\ \frac{\Gamma \vdash \forall t \geq 0\, (\forall 0 \leq s \leq t\, Q(y(s))) \rightarrow [x := y(t)]\, P}{\Gamma \vdash [x' = f(x) \,\&\, Q]\, P}$$

Figure 8: Differential Equation Rules. For **dG**, a and b are continuous on Q, and y is fresh relative to $x' = f(x)$, Q, a, b, P, Γ and Δ.

With this structure, the derivative b' of a Boolean expression b is defined as

$$b' \triangleq \begin{cases} b_1' \wedge b_2' & \text{if } b = b_1 \wedge_{nqB} b_2 \text{ or } b = b_1 \vee_{nqB} b_2 \\ r_1' \leq r_2' & \text{if } b = r_1 \leq_{nqB} r_2 \text{ or } b = r_1 <_{nqB} r_2 \\ r_1' \geq r_2' & \text{if } b = r_1 \geq_{nqB} r_2 \text{ or } b = r_1 >_{nqB} r_2 \\ r_1' = r_2' & \text{if } b = (r_1 =_{nqB} r_2) \text{ or } b = (r_1 \neq_{nqB} r_2)\,. \end{cases}$$

In PVS, $[x' := f(x)]\,(P)'$ is computed by replacing P with its equivalent non-quantified Boolean, and the derivative of any real expression r occurring in P is the real expression given by:

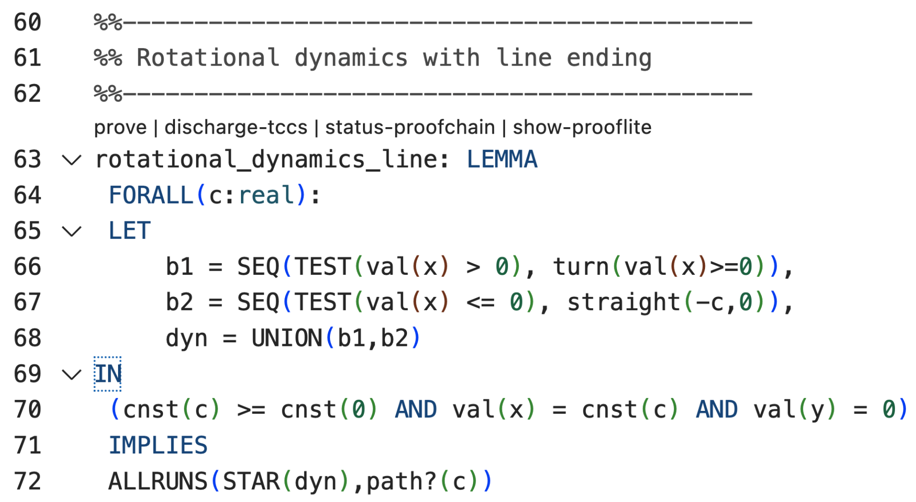
$$r' = \sum_{i \in \mathbf{x}} \partial r_i \cdot \ell(i).$$

This is the derivative of the real expression r in terms of the explicit variable that all the variables in $\mathbf{x}$ are a function of. To arrive at this formulation, differentiability and partial differentiability had to be defined for real expressions as well as the multivariate chain rule.

For the Differential Ghost rule **dG**, adding an equation to the differential equation $x' = \ell$ required that the new differential equation $x' = \ell, y' = a(x) \cdot y + b(x)$ had a unique solution. The Picard-Lindelöff theorem can be used to show that if a and b are continuous on Q, then there is a unique solution to $y' = a(x) \cdot y + b(x)$. Given a solution to $x' = \ell$ that is contained in Q, it follows that $x' = \ell, y' = a(x) \cdot y + b(x)$ has a unique solution. These properties of differential equations, including the Picard-Lindelöff theorem, were developed in PVS specifically to prove these rules.

Example 3.3 (dL-sequent example continued) *Applying **dC** with $C = \textbf{circ}(c)$ to the first branch of the proof in Example 3.2, eq. 6 produces two subgoals, the first of which (with expanded **circ**) is*

$$(x > 0, x^2 + y^2 = c^2, y \geq 0) \vdash \left[x' = -y, y' = x \& x \geq 0))\right] (x^2 + y^2 = c^2).$$



```
%%-------------------------------------------
%% Rotational dynamics with line ending
%%-------------------------------------------
rotational_dynamics_line: LEMMA
 FORALL(c:real):
 LET
     b1 = SEQ(TEST(val(x) > 0), turn(val(x)>=0)),
     b2 = SEQ(TEST(val(x) <= 0), straight(-c,0)),
     dyn = UNION(b1,b2)
IN
 (cnst(c) >= cnst(0) AND val(x) = cnst(c) AND val(y) = 0)
 IMPLIES
 ALLRUNS(STAR(dyn),path?(c))
```

Figure 9: The specification of Example 2.3 in Plaidypvs.

Using **dI** reduces to two cases:

$$\begin{gathered} x \geq 0 \vdash (2 \cdot x \cdot -y + 2 \cdot y \cdot x = 0) \\ (x \geq 0, x^2 + y^2 = c^2, y \geq 0) \vdash x^2 + y^2 = c^2, \end{gathered} \tag{7}$$

both of which can be proven with basic algebraic and logical simplifications included in command **dl-grind**.

4 Using Plaidypvs

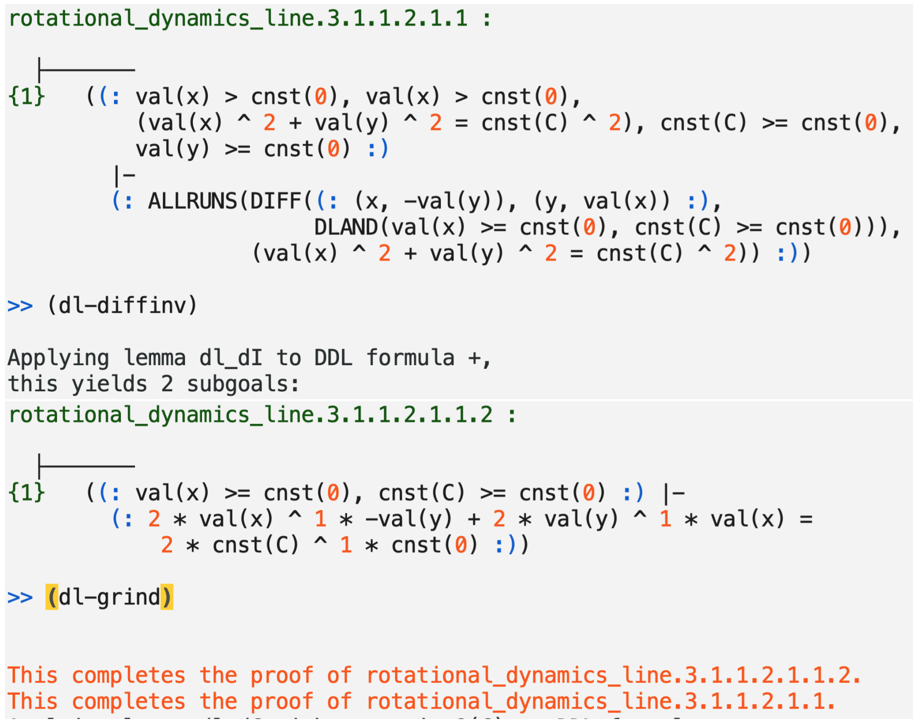
Plaidypvs provides the functionality of dL within the PVS environment. Numerous examples can be found in the directory `examples` of the Plaidypvs library. Figures 9 and 10 illustrate using dL for specification and verification of hybrid systems in Plaidypvs. However, Plaidypvs is not limited to just these applications, the embedding allows additional features to be used for formal reasoning of hybrid programs. For example, the definition of other functions from PVS libraries can be imported into a formal development that uses Plaidypvs. Furthermore, meta-properties about hybrid programs can be specified and proven. The example below illustrates these features.

Example 4.1 (Verified connection to Dubins paths) *An aircraft moving at a constant speed $c > 0$ with a turn rate of* 1 *can be modeled by a Dubins path:*

$$\theta' = 1, x' = -c\sin(\theta), y' = c\cos(\theta).$$

Furthermore, it can be shown that the hybrid program β defined as

$$\begin{aligned} ((?(x \geq 0); (\theta' = 1, x' = -c\sin(\theta), y' = c\cos(\theta) \,\&\, x \geq 0)) \cup \\ (?(x < 0); (x' = -c, y' = 0)))^*, \end{aligned}$$



```
rotational_dynamics_line.3.1.1.2.1.1 :

  |-------
{1}   ((: val(x) > cnst(0), val(x) > cnst(0),
          (val(x) ^ 2 + val(y) ^ 2 = cnst(C) ^ 2), cnst(C) >= cnst(0),
          val(y) >= cnst(0) :)
        |-
        (: ALLRUNS(DIFF((: (x, -val(y)), (y, val(x)) :),
                        DLAND(val(x) >= cnst(0), cnst(C) >= cnst(0))),
                   (val(x) ^ 2 + val(y) ^ 2 = cnst(C) ^ 2)) :))

>> (dl-diffinv)

Applying lemma dl_dI to DDL formula +,
this yields 2 subgoals:
rotational_dynamics_line.3.1.1.2.1.1.2 :

  |-------
{1}   ((: val(x) >= cnst(0), cnst(C) >= cnst(0) :) |-
        (: 2 * val(x) ^ 1 * -val(y) + 2 * val(y) ^ 1 * val(x) =
            2 * cnst(C) ^ 1 * cnst(0) :))

>> (dl-grind)


This completes the proof of rotational_dynamics_line.3.1.1.2.1.1.2.
This completes the proof of rotational_dynamics_line.3.1.1.2.1.1.
```

Figure 10: The proof steps that complete the proof discussed in Example 3.3.

is equivalent to the hybrid program α defined in Example 2.2, for appropriate initial values. Formally, this is a property relating the ***s_rel*** *function associated with each of these programs, namely for environments e_i, e_o such that $e_i(x) = c$ and $e_i(y) = 0$*

$$(\exists t : \textbf{\textit{s\_rel}}(\beta)(e_i \textit{ with } [\theta := 0])(e_o \textit{ with } \; [\theta := t])) \iff \textbf{\textit{s\_rel}}(\alpha)(e_i)(e_o \textit{ with } [\theta := e_i(\theta)]).$$

Note the property above involves generic hybrid programs rather than particular instances. Thus, for a Boolean expression Q that does not change according to θ:

$$(x = c, y = 0 \rightarrow [\alpha]\, Q) \iff (x = c, y = 0, \theta = 0 \rightarrow [\beta]\, Q).$$

5 Related work

There is a long line of research on the formal verification of hybrid systems. The development of dL itself ([42, 44, 50, 52]) and its use in formal verification of hybrid systems ([5, 6, 16, 18, 20, 23, 24, 29]) is well-known. Additionally, there has been significant work done in the PVS theorem prover [1, 54], Event-B [12], and Isabelle/HOL [14, 30, 31, 32, 53, 56, 58, 59] verifying hybrid systems outside of the dL framework.

The most similar verification effort to the current development is [4], where the authors formally verified the soundness of dL in Coq and Isabelle. The work in [4] focuses on a full formal verification of soundness of dL, with the goal of a formally verified prover kernel for KeYmaera X. The result are proof checkers in Coq and Isabelle for dL proofs. The goal of Plaidypvs is a verified operational embedding of dL in the theorem prover PVS, allowing specification and reasoning about HPs *interactively* within PVS.

While the work in [4] proves soundness of most of the proof calculus of dL, the present work focuses on verifying the proof *rules* of dL. Particularly, the substitution axiom in dL that allows rules and axioms to be applied to specifications of HPs in dL is proven in [4] but not directly proven for the PVS embedding. Instead, substitution is handled by the instantiation functionalities of PVS itself, specifically when dL rules and axioms are applied as strategies to a particular dL-sequent in the interactive prover. Additionally, there are several places where the embedding of dL in this work is more general than the work in [4]. Differential Ghost and Differential Effect in [4] are shown for a single ordinary differential equation rather than the more general *system* of ordinary differential equations. Differential Solve is only shown for differential equations with linear solutions, whereas the corresponding rule in Plaidypvs automatically solves differential equations with linear and quadratic solutions and is proven for any ODE where the solution is known. Differential Invariant in [4] is restricted to propositions of the form $P = (f(x) \geq g(x))$ and $P = (f(x) > g(x))$ and it is remarked that other cases can be derived in dL from these two cases, but in Plaidypvs Differential Invariant is fully implemented for any proposition that is the conjunction or disjunction of inequalities.

The current work formalizes a version of dL based on Parts I and II in [52], though there are many extensions as well. For adversarial cyber-physical systems there is differential game logic in [49, 51], and Part 5 of [52]. There are also extensions for distributed hybrid systems (quantified differential dynamic logic, [47]), stochastic hybrid systems (stochastic differential dynamic logic, [48]), differential algebraic programs (differential-algebraic dynamic logic, [45]), and a temporal extension of dL called differential temporal dynamic logic [43], [46, Chapter 4].

In addition to verification of hybrid systems, the present work falls more generally into the category of formal verification or simulation of logical systems inside theorem provers. PVS0 is an embedding of a fragment of the specification language of PVS *within* PVS, used in termination analysis of recursive functions [33]. Other efforts to model or verify theorem provers include work on the prover kernel of Hol Light [17], the type-checker of Coq [55], the soundness of ACL2 [10]. The goal of Plaidypvs is to add to hybrid systems reasoning to the toolbox of PVS increasing its proving capabilities. PVS has been used in verification projects in domains such as aircraft avoidance systems [36], path planning algorithms [7], unmanned aircraft systems [34], position reporting algorithms of aircraft [13], sensor uncertainty mitigation [40], floating point error analysis [27, 57], genetic algorithms [41], nonlinear control systems [3], and requirements written in linear temporal logic and FRETish [8]. In addition to advanced real number reasoning capabilities [9, 35, 38, 28, 37, 39] provided by PVS, previous work has connected PVS to the automated theorem prover MetiTarski [2], for automated reasoning of universally quantified statements about real numbers, including several transcendental functions [11]. The capability to use MetiTarski in PVS is leveraged in the **dl-assert** command in Plaidypvs.

6 Conclusion and future work

This work describes Plaidypvs, a logical embedding of dL in PVS. This embedding extends the formal verification abilities of PVS by giving a framework for specifying and reasoning about HPs and allowing features of PVS to be used naturally within the dL embedding. Novel features include support for importing user-defined functions and theories such as the extensive math and computer science developments available in NASAlib. Additionally, this embedding allows for meta reasoning about HPs and dL at the

PVS level. An example was given that illustrates capabilities of Plaidypvs that go beyond what could be be accomplished in a stand-alone implementation of dL alone such as KeYmaera X.

Regarding future work, one natural next step is to apply Plaidypvs to safety-critical applications of interest to NASA. This will include formal verification of hybrid systems related to urban air mobility and wildland fire fighting among others. Another direction is to increase the usability of Plaidypvs. To do so, a Visual Studio Code extension is under development to display specifications and the proof calculus in a natural and user-friendly way. To increase the automation of dL within Plaidypvs, a more complete substitution calculus to include Boolean expressions containing statements about hybrid programs will be implemented. Additionally, formal verification of liveness properties is intended, with implementations of strategies to match. Furthermore, a more robust ordinary differential equation solver to enhance the capabilities of the *differential solve* command would increase the usability of Plaidypvs greatly. Finally, a detailed description of the multivariate analysis and ordinary differential equation library developed to support this embedding will be written similar to the semi-algebraic set library ([54]), which was done to support verification of liveness properties in upcoming work.

The semantic structure of dL in Plaidypvs is based on the input/output semantics. Future work on defining the trace semantics of hybrid programs will extend the analysis capabilities of the embedding, such as being able to define properties in linear temporal logic like the work in [19]. It has been noted that quantifier elimination, is often the bottleneck for formal verification of hybrid programs, due to the computational complexity of the general problem. Implementation of techniques to make this process faster would help the usability of Plaidypvs. There are many directions to go for this effort, but one direction will be implementation of the active corners method for a specific class of quantifier elimination [21] geared towards formalized reasoning of aircraft operations.

References

[1] Erika Ábrahám-Mumm, Ulrich Hannemann & Martin Steffen (2001): *Verification of hybrid systems: Formalization and proof rules in PVS*. In: *Proceedings Seventh IEEE International Conference on Engineering of Complex Computer Systems*, IEEE, pp. 48–57, doi:10.1109/ICECCS.2001.930163.

[2] Behzad Akbarpour & Lawrence Charles Paulson (2010): *MetiTarski: An automatic theorem prover for real-valued special functions*. *Journal of Automated Reasoning* 44(3), pp. 175–205, doi:10.1007/s10817-009-9149-2.

[3] Cinzia Bernardeschi & Andrea Domenici (2016): *Verifying safety properties of a nonlinear control by interactive theorem proving with the Prototype Verification System*. *Information Processing Letters* 116(6), pp. 409–415, doi:10.1016/j.ipl.2016.02.001.

[4] Brandon Bohrer, Vincent Rahli, Ivana Vukotic, Marcus Völp & André Platzer (2017): *Formally verified differential dynamic logic*. In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pp. 208–221, doi:10.1145/3018610.3018616.

[5] Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Andrew Sogokon & André Platzer (2019): *A Formal Safety Net for Waypoint Following in Ground Robots*. *IEEE Robotics and Automation Letters* 4(3), pp. 2910–2917, doi:10.1109/LRA.2019.2923099.

[6] Rachel Cleaveland, Stefan Mitsch & André Platzer (2023): *Formally Verified Next-Generation Airborne Collision Avoidance Games in ACAS X*. *ACM Trans. Embed. Comput. Syst.* 22(1), pp. 1–30, doi:10.1145/3544970.

[7] Brendon K Colbert, J Tanner Slagel, Luis G Crespo, Swee Balachandran & César A. Muñoz (2020): *PolySafe: A Formally Verified Algorithm for Conflict Detection on a Polynomial Airspace*. *IFAC-PapersOnLine* 53(2), pp. 15615–15620, doi:10.1016/j.ifacol.2020.12.2496.

[8] Esther Conrad, Laura Titolo, Dimitra Giannakopoulou, Thomas Pressburger & Aaron Dutle (2022): *A compositional proof framework for FRETish requirements*. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 68–81, doi:10.1145/3497775.3503685.

[9] Marc Daumas, David Lester & César A. Muñoz (2008): *Verified real number calculations: A library for interval arithmetic*. *IEEE Transactions on Computers* 58(2), pp. 226–237, doi:10.1109/tc.2008.213.

[10] Jared Davis & Magnus O Myreen (2015): *The reflective Milawa theorem prover is sound (down to the machine code that runs it)*. *Journal of Automated Reasoning* 55(2), pp. 117–183, doi:10.1007/s10817-015-9324-6.

[11] William Denman & César A. Muñoz (2014): *Automated real proving in PVS via MetiTarski*. In: *International Symposium on Formal Methods*, Springer, pp. 194–199, doi:10.1007/978-3-319-06410-9_14.

[12] Guillaume Dupont (2021): *Correct-by-construction design of hybrid systems based on refinement and proof*. Ph.D. thesis, Université de Toulouse. Available at https://oatao.univ-toulouse.fr/28190/1/Dupont_Guillaume.pdf.

[13] Aaron Dutle, Mariano Moscato, Laura Titolo, César A. Muñoz, Gregory Anderson & Bobot François (2021): *Formal analysis of the compact position reporting algorithm*. *Formal Aspects of Computing* 33(1), pp. 65–86, doi:10.1007/s00165-019-00504-0.

[14] Simon Foster, Jonathan Julián Huerta y Munive, Mario Gleirscher & Georg Struth (2021): *Hybrid systems verification with Isabelle/HOL: Simpler syntax, better models, faster proofs*. In: *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings 24*, Springer, pp. 367–386, doi:10.1007/978-3-030-90870-6_20.

[15] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp & André Platzer (2015): *KeYmaera X: An axiomatic tactical theorem prover for hybrid systems*. In: *International Conference on Automated Deduction*, Springer, pp. 527–538, doi:10.1007/978-3-319-21401-6_36.

[16] Nathan Fulton & André Platzer (2018): *Safe Reinforcement Learning via Formal Methods: Toward Safe Control Through Proof and Learning*. In Sheila McIlraith & Kilian Weinberger, editors: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, February 2-7, 2018, New Orleans, Louisiana, USA.*, AAAI Press, pp. 6485–6492, doi:10.1609/aaai.v32i1.12107.

[17] John Harrison (2006): *Towards self-verification of HOL Light*. In: *International Joint Conference on Automated Reasoning*, Springer, pp. 177–191, doi:10.1007/11814771_17.

[18] Jean-Baptiste Jeannin, Khalil Ghorbal, Yanni Kouskoulas, Aurora Schmidt, Ryan Gardner, Stefan Mitsch & André Platzer (2017): *A Formally Verified Hybrid System for Safe Advisories in the Next-generation Airborne Collision Avoidance System*. *STTT* 19(6), pp. 717–741, doi:10.1007/s10009-016-0434-1.

[19] Jean-Baptiste Jeannin & André Platzer (2014): *dTL 2: differential temporal dynamic logic with nested temporalities for hybrid systems*. In: *International Joint Conference on Automated Reasoning*, Springer, pp. 292–306, doi:10.1007/978-3-319-08587-6_22.

[20] Aditi Kabra, Stefan Mitsch & André Platzer (2022): *Verified Train Controllers for the Federal Railroad Administration Train Kinematics Model: Balancing Competing Brake and Track Forces*. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 41(11), pp. 4409–4420, doi:10.1109/TCAD.2022.3197690.

[21] Nishant Kheterpal, Elanor Tang & Jean-Baptiste Jeannin (2022): *Automating Geometric Proofs of Collision Avoidance with Active Corners*. In: *Conference On Formal Methods In Computer-Aided Design–FMCAD 2022*, p. 359, doi:10.34727/2022/isbn.978-3-85448-053-2_43.

[22] Stefan Mitsch (2021): *Implicit and explicit proof management in keymaera x*. *arXiv preprint arXiv:2108.02965*, doi:10.48550/arXiv.2108.02965.

[23] Stefan Mitsch, Marco Gario, Christof J. Budnik, Michael Golm & André Platzer (2017): *Formal Verification of Train Control with Air Pressure Brakes*. In: *RSSRail 2017: Reliability, Safety, and Security of Railway Systems*, *LNCS* 10598, Springer, pp. 173–191, doi:10.1007/978-3-319-68499-4_12.

[24] Stefan Mitsch, Khalil Ghorbal, David Vogelbacher & André Platzer (2017): *Formal Verification of Obstacle Avoidance and Navigation of Ground Robots*. *I. J. Robotics Res.* 36(12), pp. 1312–1340, doi:10.1177/0278364917733549.

[25] Stefan Mitsch & André Platzer (2017): *The keymaera X proof IDE-concepts on usability in hybrid systems theorem proving*. *arXiv preprint arXiv:1701.08469*, doi:10.48550/arXiv.1701.08469.

[26] Stefan Mitsch & André Platzer (2020): *A Retrospective on Developing Hybrid System Provers in the KeYmaera Family: A Tale of Three Provers*. In: *Deductive Software Verification: Future Perspectives: Reflections on the Occasion of 20 Years of KeY*, Springer, pp. 21–64, doi:10.1007/978-3-030-64354-6_2.

[27] Mariano Moscato, Laura Titolo, Aaron Dutle & César A. Muñoz (2017): *Automatic estimation of verified floating-point round-off errors via static analysis*. In: *International Conference on Computer Safety, Reliability, and Security*, Springer, pp. 213–229, doi:10.1007/978-3-319-66266-4_14.

[28] Mariano M Moscato, César A. Muñoz & Andrew P Smith (2015): *Affine arithmetic and applications to real-number proving*. In: *International Conference on Interactive Theorem Proving*, Springer, pp. 294–309, doi:10.1007/978-3-319-22102-1_20.

[29] Andreas Müller, Stefan Mitsch, Werner Retschitzegger, Wieland Schwinger & André Platzer (2017): *Change and Delay Contracts for Hybrid System Component Verification*. In: *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Proceedings*, *Lecture Notes in Computer Science* 10202, Springer, pp. 134–151, doi:10.1007/978-3-662-54494-5_8.

[30] Jonathan Julian Huerta y Munive (2020): *Algebraic verification of hybrid systems in Isabelle/HOL*. Ph.D. thesis, University of Sheffield. Available at https://etheses.whiterose.ac.uk/28886/.

[31] Jonathan Julián Huerta y Munive & Georg Struth (2018): *Verifying hybrid systems with modal Kleene algebra*. In: *International Conference on Relational and Algebraic Methods in Computer Science*, Springer, pp. 225–243, doi:10.1007/978-3-030-02149-8_14.

[32] Jonathan Julián Huerta y Munive & Georg Struth (2022): *Predicate Transformer Semantics for Hybrid Systems*. *Journal of Automated Reasoning* 66(1), pp. 93–139, doi:10.1007/s10817-021-09607-x.

[33] César A. Muñoz, Mauricio Ayala-Rincón, Mariano Moscato, Aaron Dutle, Anthony Narkawicz, Ariane Alves, Andréia Avelar & Thiago Ferreira (2021): *Formal Verification of Termination Criteria for First-Order Recursive Functions*. In Liron Cohen & Cezary Kalisyk, editors: *Proceedings of the 12th International Conference on Interactive Theorem Proving (ITP 2021)*, 26, Dagstuhl Publishing, Germany, pp. 26:1–26:17, doi:10.4230/LIPIcs.ITP.2021.27.

[34] César A. Muñoz, Aaron Dutle, Anthony Narkawicz & Jason Upchurch (2016): *Unmanned aircraft systems in the national airspace system: a formal methods perspective*. *ACM SIGLOG News* 3(3), pp. 67–76, doi:10.1145/2984450.2984459.

[35] César A. Muñoz & Anthony Narkawicz (2013): *Formalization of Bernstein polynomials and applications to global optimization*. *Journal of Automated Reasoning* 51(2), pp. 151–196, doi:10.1007/s10817-012-9256-3.

[36] César A. Muñoz, Anthony Narkawicz, George Hagen, Jason Upchurch, Aaron Dutle, María Consiglio & James Chamberlain (2015): *DAIDALUS: detect and avoid alerting logic for unmanned systems*. In: *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, IEEE, pp. 5A1–1, doi:10.1109/DASC.2015.7311421.

[37] Anthony Narkawicz & César A. Muñoz (2013): *A formally verified generic branching algorithm for global optimization*. In: *Working Conference on Verified Software: Theories, Tools, and Experiments*, Springer, pp. 326–343, doi:10.1007/978-3-642-54108-7_17.

[38] Anthony Narkawicz & César A. Muñoz (2014): *A Formally Verified Generic Branching Algorithm for Global Optimization*. In Ernie Cohen & Andrey Rybalchenko, editors: *Proceedings of the 5th International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2013)*, *Lecture Notes in Computer Science* 8164, Springer, Menlo Park, CA, US, pp. 326–343, doi:10.1007/978-3-642-54108-7_17.

[39] Anthony Narkawicz, César A. Muñoz & Aaron Dutle (2015): *Formally-verified decision procedures for univariate polynomial computation based on Sturm's and Tarski's theorems*. Journal of Automated Reasoning 54(4), pp. 285–326, doi:10.1007/s10817-015-9320-x.

[40] Anthony Narkawicz, César A. Muñoz & Aaron Dutle (2018): *Sensor uncertainty mitigation and dynamic well clear volumes in DAIDALUS*. In: 2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC), IEEE, pp. 1–8, doi:10.1109/DASC.2018.8569468.

[41] M Saqib Nawaz, M IkramUllah Lali & MA Pasha (2013): *Formal verification of crossover operator in genetic algorithms using prototype verification system (PVS)*. In: 2013 IEEE 9th International Conference on Emerging Technologies (ICET), IEEE, pp. 1–6, doi:10.1109/ICET.2013.6743532.

[42] André Platzer (2007): *Differential Dynamic Logic for Verifying Parametric Hybrid Systems*. In Nicola Olivetti, editor: TABLEAUX, LNCS 4548, Springer, pp. 216–232, doi:10.1007/978-3-540-73099-6_17.

[43] André Platzer (2007): *A Temporal Dynamic Logic for Verifying Hybrid System Invariants*. In Sergei N. Artëmov & Anil Nerode, editors: LFCS, LNCS 4514, Springer, pp. 457–471, doi:10.1007/978-3-540-72734-7_32.

[44] André Platzer (2008): *Differential dynamic logic for hybrid systems*. Journal of Automated Reasoning 41(2), pp. 143–189, doi:10.1007/s10817-008-9103-8.

[45] André Platzer (2010): *Differential-algebraic Dynamic Logic for Differential-algebraic Programs*. J. Log. Comput. 20(1), pp. 309–352, doi:10.1093/logcom/exn070. Advance Access published on November 18, 2008.

[46] André Platzer (2010): *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg, doi:10.1007/978-3-642-14509-4.

[47] André Platzer (2010): *Quantified Differential Dynamic Logic for Distributed Hybrid Systems*. In Anuj Dawar & Helmut Veith, editors: CSL, LNCS 6247, Springer, pp. 469–483, doi:10.1007/978-3-642-15205-4_36.

[48] André Platzer (2011): *Stochastic Differential Dynamic Logic for Stochastic Hybrid Programs*. In Nikolaj Bjørner & Viorica Sofronie-Stokkermans, editors: CADE, LNCS 6803, Springer, pp. 446–460, doi:10.1007/978-3-642-22438-6_34.

[49] André Platzer (2015): *Differential Game Logic*. ACM Trans. Comput. Log. 17(1), pp. 1:1–1:51, doi:10.1145/2817824.

[50] André Platzer (2017): *A complete uniform substitution calculus for differential dynamic logic*. Journal of Automated Reasoning 59(2), pp. 219–265, doi:10.1007/s10817-016-9385-1.

[51] André Platzer (2017): *Differential Hybrid Games*. ACM Trans. Comput. Log. 18(3), pp. 19:1–19:44, doi:10.1145/3091123.

[52] André Platzer (2018): *Logical Foundations of Cyber-Physical Systems*. Springer, Cham, doi:10.1007/978-3-319-63588-0.

[53] Huanhuan Sheng, Alexander Bentkamp & Bohua Zhan (2023): *HHLPy: Practical Verification of Hybrid Systems Using Hoare Logic*. In: Formal Methods: 25th International Symposium, FM 2023, Lübeck, Germany, March 6–10, 2023, Proceedings, Springer, pp. 160–178, doi:10.1007/978-3-031-27481-7_11.

[54] J Tanner Slagel, Lauren White & Aaron Dutle (2021): *Formal verification of semi-algebraic sets and real analytic functions*. In: Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 278–290, doi:10.1145/3437992.3439933.

[55] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau & Théo Winterhalter (2019): *Coq coq correct! verification of type checking and erasure for coq, in coq*. Proceedings of the ACM on Programming Languages 4(POPL), pp. 1–28, doi:10.1145/3371076.

[56] Georg Struth (2021): *Hybrid Systems Verification with Isabelle/HOL: Simpler Syntax, Better Models, Faster Proofs*. In: Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings, 13047, Springer Nature, p. 367, doi:10.1007/978-3-030-90870-6_20.

[57] Laura Titolo, Marco A. Feliú, Mariano M. Moscato & César A. Muñoz (2018): *An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs*. In Isil Dillig & Jens Palsberg, editors: *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*, *Lecture Notes in Computer Science* 10747, Springer, pp. 516–537, doi:10.1007/978-3-319-73721-8_24.

[58] Shuling Wang, Naijun Zhan & Liang Zou (2015): *An improved HHL prover: an interactive theorem prover for hybrid systems*. In: *Formal Methods and Software Engineering: 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings 17*, Springer, pp. 382–399, doi:10.1007/978-3-319-25423-4_25.

[59] Liang Zou, Naijun Zhan, Shuling Wang & Martin Fränzle (2015): *Formal verification of Simulink/Stateflow diagrams*. In: *Automated Technology for Verification and Analysis: 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings 13*, Springer, pp. 464–481, doi:10.1007/978-3-319-24953-7_33.

Semi-Substructural Logics with Additives

Niccolò Veltri Cheng-Syuan Wan
Tallinn University of Technology, Estonia
niccolo@cs.ioc.ee cswan@cs.ioc.ee

This work concerns the proof theory of (left) skew monoidal categories and their variants (e.g. closed monoidal, symmetric monoidal), continuing the line of work initiated in recent years by Uustalu et al. Skew monoidal categories are a weak version of Mac Lane's monoidal categories, where the structural laws are not required to be invertible, they are merely natural transformations with a specific orientation. Sequent calculi which can be modelled in such categories can be identified as deductive systems for restricted substructural fragments of intuitionistic linear logic. These calculi enjoy cut elimination and admit a focusing strategy, sharing resemblance with Andreoli's normalization technique for linear logic. The focusing procedure is useful for solving the coherence problem of the considered categories with skew structure.

Here we investigate possible extensions of the sequent calculi of Uustalu et al. with additive connectives. As a first step, we extend the sequent calculus with additive conjunction and disjunction, corresponding to studying the proof theory of skew monoidal categories with binary products and coproducts satisfying a left-distributivity condition. We introduce a new focused sequent calculus of derivations in normal form, which employs tag annotations to reduce non-deterministic choices in bottom-up proof search. The focused sequent calculus and the proof of its correctness have been formalized in the Agda proof assistant. We also discuss extensions of the logic with additive units, a form of skew exchange and linear implication.

1 Introduction

Substructural logics are logical systems in which the usage of one or more structural rules is disallowed or restricted. A well-known example is the syntactic calculus of Lambek [13], in which all the structural rules of exchange, weakening and contraction are disallowed. Variants of the Lambek calculus allow exchange or a cyclic form of exchange, while others disallow even associativity [16]. In Girard's linear logic, which has been studied both in the presence and absence of an exchange rule [9, 1], selective versions of weakening and contraction can be recovered via the use of modalities. Applications of substructural logics are abundant in a variety of different fields, from computational investigations of natural languages to the design of resource-sensitive programming languages.

In recent years, in collaboration with Tarmo Uustalu and Noam Zeilberger, we initiated a program intended to study a family of *semi-substructural* logics, inspired by developments in category theory by Szlachányi, Street, Bourke, Lack and many others [19, 11, 18, 12, 7, 4, 5, 6]. Kornél Szlachányi introduced *skew monoidal categories* as a weakening of MacLane's monoidal categories in which the structural morphisms of associativity and unitality (often also called associator and unitors) are not required to be invertible, they are just natural transformation in a particular direction. As such, they can be regarded as *semi-associative* and *semi-unital* variants of monoidal categories. Bourke and Lack also introduced notions of braiding and symmetry for skew monoidal categories which involve three objects instead of two [6]. Skew monoidal categories arise naturally in semantics of programming languages [2] and semi-associativity has strong connections with combinatorial structures such as the Tamari lattice and Stasheff associahedra [26, 15].

Semi-substructural logics correspond to the internal languages of skew monoidal categories and their extensions, therefore sitting in between (certain fragments of) non-associative and associative intuitionistic linear logic. Semi-associativity and semi-unitality can be hard-coded in the sequent calculus following a two-step recipe. First, consider sequents of the form $S \mid \Gamma \vdash A$, where the antecedent is split into an optional formula S, called stoup, and an ordered list of formulae Γ. The succedent consists of a single formula A. Then restrict the application of introduction rules to allow only one of the directions of associativity and unitality, the one in the definition of skew monoidal category. For example, left-introduction rules are allowed to act only on the formula in stoup position, not on formulae in Γ.

In our previous investigations we have explored deductive systems for (i) skew semigroup [26], (ii) skew monoidal [23], (iii) skew (prounital) closed [21] and (iv) skew monoidal closed categories [20, 25], corresponding to skew variants of the fragments of non-commutative intuitionistic linear logic consisting of connectives (i) $\otimes$, (ii) $(\mathsf{I},\otimes)$, (iii) $\multimap$ and (iv) $(\mathsf{I},\otimes,\multimap)$. We have also studied partial normality conditions, when one or more among associator and unitors is allowed to have an inverse [22], and extensions with exchange à la Bourke and Lack [24].

When studying meta-theoretic properties of these semi-structural deductive systems, we have been mostly interested in categorical and proof-theoretic semantics. In the latter, we have particularly investigated normalization strategies inspired by Andreoli's focused sequent calculus for classical linear logic [3] and employed the resulting normal forms to solve the *coherence problem* for the corresponding categories with skew structure. For these categories, the word problem is more nuanced than in the normal non-skew case studied by MacLane [14]. Our study additionally revealed that the focused sequent calculi of semi-substructural logics can serve as cornerstones for a compositional and modular understanding of normalization techniques for other richer substructural logics.

In this work we begin the investigation of semi-substructural logics with *additive connectives*. We start in Section 2 by considering a fragment of non-commutative linear logic consisting of skew multiplicative unit I and conjunction $\otimes$, and additive conjunction $\wedge$ and disjunction $\vee$. We describe a cut-free sequent calculus and a congruence relation $\stackrel{\circ}{=}$ identifying derivations up-to η-equivalence and permutative conversions. In Section 3, we discuss categorical semantics in terms of skew monoidal categories with binary products and coproducts satisfying a left-distributivity condition.

In Section 4, we introduce a sequent calculus of proofs in normal form, i.e. canonical representative of the equivalence relation on derivations $\stackrel{\circ}{=}$. The design of the latter calculus is again inspired by the ideas of Andreoli and it describes a sound and complete root-first proof search strategy for the original sequent calculus. Completeness is achieved by marking sequents with lists of *tags*, a mechanism introduced by Uustalu et al. [20] and inspired by Scherer and Rémy's saturation technique [17], which helps to completely eliminate all undesired non-determinism in proof search and faithfully capture normal forms wrt. the congruence relation on derivations in the original sequent calculus.

To showcase the modularity of our normalization strategy, in Section 5 we discuss extensions of the logic with other connectives, such as additive units, the structural rule of exchange in the style of Bourke and Lack and linear implication. This provides evidence that our normalization technique is potentially scalable to other richer substructural logics arising as extensions of ours, e.g. full Lambek calculus or intuitionistic linear logic. Moreover, we conjecture that a similar use of tags can be ported to fragments of classical linear logic, such as MALL. The resulting notion of normal form should correspond to maximally multi-focused proofs [8] and therefore proof nets.

The sequent calculi of Sections 2 and 4, as well as the effective normalization procedure, have been fully formalized in the Agda proof assistant. The code is freely available at

`https://github.com/cswphilo/SkewMonAdd.`

2 Sequent Calculus

We start by describing a sequent calculus for a skew variant of non-commutative multiplicative intuitionistic linear logic with additive conjunction and disjunction.

Formulae are inductively generated by the grammar $A, B ::= X \mid \mathsf{I} \mid A \otimes B \mid A \wedge B \mid A \vee B$, where X comes from a set At of atomic formulae. We use $\mathsf{I}, \otimes, \wedge$ and $\vee$ to denote multiplicative verum, multiplicative conjunction, additive conjunction and additive disjunction, respectively. The additives are traditionally named & and $\oplus$ in linear logic literature.

A sequent is a triple of the form $S \mid \Gamma \vdash A$. The antecedent is split in two parts: an optional formula S, called *stoup* [10], and an ordered list of formulae Γ, called *context*. The succedent A is a single formula. The peculiar design of sequents, involving the presence of the stoup in the antecedent, comes from our previous work on deductive systems with skew structure in collaboration with Uustalu and Zeilberger [23, 22, 21, 24, 20, 25]. The metavariable S always denotes a stoup, i.e., S can be a single formula or empty, in which case we write $S = -$. Metavariables X, Y, Z are always names of atomic formulae.

Derivations of a sequent $S \mid \Gamma \vdash A$ are inductively generated by the following rules:

$$\frac{}{A \mid \ \vdash A}\ \mathsf{ax} \qquad \frac{A \mid \Gamma \vdash C}{- \mid A, \Gamma \vdash C}\ \mathsf{pass} \qquad \frac{- \mid \Gamma \vdash C}{\mathsf{I} \mid \Gamma \vdash C}\ \mathsf{IL} \qquad \frac{}{- \mid \ \vdash \mathsf{I}}\ \mathsf{IR}$$
$$\frac{A \mid B, \Gamma \vdash C}{A \otimes B \mid \Gamma \vdash C}\ \otimes\mathsf{L} \qquad \frac{S \mid \Gamma \vdash A \quad - \mid \Delta \vdash B}{S \mid \Gamma, \Delta \vdash A \otimes B}\ \otimes\mathsf{R}$$
$$\frac{A \mid \Gamma \vdash C}{A \wedge B \mid \Gamma \vdash C}\ \wedge\mathsf{L}_1 \qquad \frac{B \mid \Gamma \vdash C}{A \wedge B \mid \Gamma \vdash C}\ \wedge\mathsf{L}_2 \qquad \frac{S \mid \Gamma \vdash A \quad S \mid \Gamma \vdash B}{S \mid \Gamma \vdash A \wedge B}\ \wedge\mathsf{R} \tag{1}$$
$$\frac{A \mid \Gamma \vdash C \quad B \mid \Gamma \vdash C}{A \vee B \mid \Gamma \vdash C}\ \vee\mathsf{L} \qquad \frac{S \mid \Gamma \vdash A}{S \mid \Gamma \vdash A \vee B}\ \vee\mathsf{R}_1 \qquad \frac{S \mid \Gamma \vdash B}{S \mid \Gamma \vdash A \vee B}\ \vee\mathsf{R}_2$$

The inference rules are similar to the ones in non-commutative intuitionistic linear logic [1], but with some essential differences.

1. The left logical rules IL, $\otimes\mathsf{L}$, $\wedge\mathsf{L}_i$ and $\vee\mathsf{L}$, when read bottom-up, can only be applied on the formula in the stoup position. That is, it is generally not possible to remove a unit I, or decompose a tensor $A \otimes B$ or a disjunction $A \vee B$, when these formulae are located in the context.

2. The right tensor rule $\otimes\mathsf{R}$, when read bottom-up, splits the antecedent of the conclusion but the formula in the stoup, whenever this is present, always moves to the left premise. The stoup formula of the conclusion is prohibited to move to the second premise even if Γ is empty.

3. The presence of the stoup implies a distinction between antecedents of the form $A \mid \Gamma$ and $- \mid A, \Gamma$. The structural rule pass (for 'passivation'), when read bottom-up, allows the moving of the leftmost formula in the context to the stoup position whenever the stoup is initially empty.

These restrictions allow the derivability of sequents $(A \otimes B) \otimes C \mid \ \vdash A \otimes (B \otimes C)$ (semi-associativity), $\mathsf{I} \otimes A \mid \ \vdash A$ and $A \mid \ \vdash A \otimes \mathsf{I}$ (semi-unitality), while forbidding the derivability of their inverses, where the formulae in the stoup and in the succedent have been swapped. This is in line with the intended categorical semantics, see Section 3.

Notice that, similarly to the case of non-commutative intuitionistic linear logic [1], all structural rules of exchange, contraction and weakening are absent. We give names to derivations and we write $f : S \mid \Gamma \vdash A$ when f is a particular derivation of the sequent $S \mid \Gamma \vdash A$.

Theorem 2.1. *The sequent calculus enjoys cut admissibility: the following two cut rules are admissible*

$$\frac{S \mid \Gamma \vdash A \quad A \mid \Delta \vdash C}{S \mid \Gamma,\Delta \vdash C}\,\mathsf{scut} \qquad \frac{- \mid \Gamma \vdash A \quad S \mid \Delta_0, A, \Delta_1 \vdash C}{S \mid \Delta_0,\Gamma,\Delta_1 \vdash C}\,\mathsf{ccut}$$

While the left $\wedge$-rules only act on the formula in stoup position (as all the other left logical rules), other $\wedge$-rules $\wedge\mathsf{L}_i^{\mathsf{C}}$ acting on formulae in context are admissible.

$$\frac{S \mid \Gamma, A, \Delta \vdash C}{S \mid \Gamma, A \wedge B, \Delta \vdash C}\,\wedge\mathsf{L}_1^{\mathsf{C}} \qquad \frac{S \mid \Gamma, B, \Delta \vdash C}{S \mid \Gamma, A \wedge B, \Delta \vdash C}\,\wedge\mathsf{L}_2^{\mathsf{C}}$$

However, this is not the case for the other left logical rules. For example, there is no way of constructing a general left $\vee$-rule $\vee\mathsf{L}^{\mathsf{C}}$ acting on a disjunction in context. This rule should be forbidden since it would make some inadmissible sequents provable in the sequent calculus. For example, the sequent $X \wedge Y \mid Y \vee X \vdash (X \otimes Y) \vee (Y \otimes X)$ is not admissible (this can be proved using the normalization procedure of Section 4) but a proof could be found using $\vee\mathsf{L}^{\mathsf{C}}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{X \mid \ \vdash X}\,\mathsf{ax} \quad \dfrac{\dfrac{}{Y \mid \ \vdash Y}\,\mathsf{ax}}{- \mid Y \vdash Y}\,\mathsf{pass}}{X \mid Y \vdash X \otimes Y}\,\otimes\mathsf{R}}{X \wedge Y \mid Y \vdash X \otimes Y}\,\wedge\mathsf{L}_1}{X \wedge Y \mid Y \vdash (X \otimes Y) \vee (Y \otimes X)}\,\vee\mathsf{R}_1 \quad \dfrac{\dfrac{\dfrac{\dfrac{}{Y \mid \ \vdash Y}\,\mathsf{ax} \quad \dfrac{\dfrac{}{X \mid \ \vdash X}\,\mathsf{ax}}{- \mid X \vdash X}\,\mathsf{pass}}{Y \mid X \vdash Y \otimes X}\,\otimes\mathsf{R}}{X \wedge Y \mid X \vdash Y \otimes X}\,\wedge\mathsf{L}_2}{X \wedge Y \mid X \vdash (X \otimes Y) \vee (Y \otimes X)}\,\vee\mathsf{R}_2}{X \wedge Y \mid Y \vee X \vdash (X \otimes Y) \vee (Y \otimes X)}\,\vee\mathsf{L}^{\mathsf{C}}$$

We introduce a congruence relation $\overset{\circ}{=}$ on the sets of cut-free derivations:

$$\begin{array}{rll}
\mathsf{ax}_{\mathsf{I}} & \overset{\circ}{=} \mathsf{IL}\ (\mathsf{IR}) & \\
\mathsf{ax}_{A\otimes B} & \overset{\circ}{=} \otimes\mathsf{L}\ (\otimes\mathsf{R}\ (\mathsf{ax}_A, \mathsf{pass}\ \mathsf{ax}_B)) & \\
\mathsf{ax}_{A\wedge B} & \overset{\circ}{=} \wedge\mathsf{R}\ (\wedge\mathsf{L}_1\ \mathsf{ax}_A, \wedge\mathsf{L}_2\ \mathsf{ax}_B) & \\
\mathsf{ax}_{A\vee B} & \overset{\circ}{=} \vee\mathsf{L}\ (\vee\mathsf{R}_1\ \mathsf{ax}_A, \vee\mathsf{R}_2\ \mathsf{ax}_B) & \\
\otimes\mathsf{R}\ (\mathsf{pass}\ f, g) & \overset{\circ}{=} \mathsf{pass}\ (\otimes\mathsf{R}\ (f,g)) & (f : A' \mid \Gamma \vdash A, g : - \mid \Delta \vdash B) \\
\otimes\mathsf{R}\ (\mathsf{IL}\ f, g) & \overset{\circ}{=} \mathsf{IL}\ (\otimes\mathsf{R}\ (f,g)) & (f : - \mid \Gamma \vdash A, g : - \mid \Delta \vdash B) \\
\otimes\mathsf{R}\ (\otimes\mathsf{L}\ f, g) & \overset{\circ}{=} \otimes\mathsf{L}\ (\otimes\mathsf{R}\ (f,g)) & (f : A' \mid B', \Gamma \vdash A, g : - \mid \Delta \vdash B) \\
\otimes\mathsf{R}\ (\wedge\mathsf{L}_i\ f, g) & \overset{\circ}{=} \wedge\mathsf{L}_i\ (\otimes\mathsf{R}\ (f,g)) & (f : A' \mid \Gamma \vdash A, g : - \mid \Delta \vdash B) \\
\otimes\mathsf{R}\ (\vee\mathsf{L}\ (f_1,f_2), g) & \overset{\circ}{=} \vee\mathsf{L}\ (\otimes\mathsf{R}\ (f_1,g), \otimes\mathsf{R}\ (f_2,g)) & (f_1 : A' \mid \Gamma \vdash A, f_2 : B' \mid \Gamma \vdash A, \\
 & & \ g : - \mid \Delta \vdash B) \\
\mathsf{pass}\ (\wedge\mathsf{R}\ (f,g)) & \overset{\circ}{=} \wedge\mathsf{R}\ (\mathsf{pass}\ f, \mathsf{pass}\ g) & (f : A' \mid \Gamma \vdash A, g : A' \mid \Gamma \vdash B) \\
\mathsf{IL}\ (\wedge\mathsf{R}\ (f,g)) & \overset{\circ}{=} \wedge\mathsf{R}\ (\mathsf{IL}\ f, \mathsf{IL}\ g) & (f : - \mid \Gamma \vdash A, g : - \mid \Gamma \vdash B) \\
\otimes\mathsf{L}\ (\wedge\mathsf{R}\ (f,g)) & \overset{\circ}{=} \wedge\mathsf{R}\ (\otimes\mathsf{L}\ f, \otimes\mathsf{L}\ g) & (f : A' \mid B', \Gamma \vdash A, g : A' \mid B', \Gamma \vdash B) \\
\wedge\mathsf{L}_i\ (\wedge\mathsf{R}\ (f,g)) & \overset{\circ}{=} \wedge\mathsf{R}\ (\wedge\mathsf{L}_i\ f, \wedge\mathsf{L}_i\ g) & (f : A' \mid \Gamma \vdash A, g : A' \mid \Gamma \vdash B) \\
\vee\mathsf{L}\ (\wedge\mathsf{R}\ (f_1,g_1), \wedge\mathsf{R}\ (f_2,g_2)) & \overset{\circ}{=} \wedge\mathsf{R}\ (\vee\mathsf{L}\ (f_1,f_2), \vee\mathsf{L}\ (g_1,g_2)) & (f_1 : A' \mid \Gamma \vdash A, f_2 : B' \mid \Gamma \vdash A, \\
 & & \ g_1 : A' \mid \Gamma \vdash B, g_2 : B' \mid \Gamma \vdash B) \\
\vee\mathsf{R}_i\ (\mathsf{pass}\ f) & \overset{\circ}{=} \mathsf{pass}\ (\vee\mathsf{R}_i\ f) & (f : A' \mid \Gamma \vdash A) \\
\vee\mathsf{R}_i\ (\mathsf{IL}\ f) & \overset{\circ}{=} \mathsf{IL}\ (\vee\mathsf{R}_i\ f) & (f : - \mid \Gamma \vdash A) \\
\vee\mathsf{R}_i\ (\otimes\mathsf{L}\ f) & \overset{\circ}{=} \otimes\mathsf{L}\ (\vee\mathsf{R}_i\ f) & (f : A' \mid B', \Gamma \vdash A) \\
\vee\mathsf{R}_i\ (\wedge\mathsf{L}_i\ f) & \overset{\circ}{=} \wedge\mathsf{L}_i\ (\vee\mathsf{R}_i\ f) & (f : A' \mid \Gamma \vdash A) \\
\vee\mathsf{R}_i\ (\vee\mathsf{L}\ (f,g)) & \overset{\circ}{=} \vee\mathsf{L}\ (\vee\mathsf{R}_i\ f, \vee\mathsf{R}_i\ g) & (f : A' \mid \Gamma \vdash A, g : B' \mid \Gamma \vdash A)
\end{array} \tag{2}$$

The first four equations (η-conversions) characterize the ax rule for non-atomic formulae. The remaining equations are permutative conversions. The congruence $\overset{\circ}{=}$ has been carefully chosen to serve as the proof-theoretic counterpart of the equational theory of certain categories with skew structure, which we introduce in the next section.

3 Categorical Semantics

A *skew monoidal category* [19, 11, 12] is a category $\mathbb{C}$ with a unit object I, a functor $\otimes : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ and three natural transformations λ, ρ, α typed $\lambda_A : \mathsf{I} \otimes A \to A$, $\rho_A : A \to A \otimes \mathsf{I}$ and $\alpha_{A,B,C} : (A \otimes B) \otimes C \to A \otimes (B \otimes C)$, satisfying the following equations due to Mac Lane [14]:

$$\begin{tikzcd} & \mathsf{I}\otimes\mathsf{I} \arrow[dr, "\lambda_{\mathsf{I}}"] & \\ \mathsf{I} \arrow[ur, "\rho_{\mathsf{I}}"] \arrow[rr, equal] & & \mathsf{I} \end{tikzcd} \qquad \begin{tikzcd} (A\otimes\mathsf{I})\otimes B \arrow[r, "\alpha_{A,\mathsf{I},B}"] & A\otimes(\mathsf{I}\otimes B) \arrow[d, "A\otimes\lambda_B"] \\ A\otimes B \arrow[u, "\rho_A\otimes B"] \arrow[r, equal] & A\otimes B \end{tikzcd}$$

$$\begin{tikzcd} (\mathsf{I}\otimes A)\otimes B \arrow[rr, "\alpha_{\mathsf{I},A,B}"] \arrow[dr, "\lambda_A\otimes B"'] & & \mathsf{I}\otimes(A\otimes B) \arrow[dl, "\lambda_{A\otimes B}"] \\ & A\otimes B & \end{tikzcd} \qquad \begin{tikzcd} (A\otimes B)\otimes\mathsf{I} \arrow[rr, "\alpha_{A,B,\mathsf{I}}"] & & A\otimes(B\otimes\mathsf{I}) \\ & A\otimes B \arrow[ul, "\rho_{A\otimes B}"] \arrow[ur, "A\otimes\rho_B"'] & \end{tikzcd}$$

$$\begin{tikzcd} (A\otimes(B\otimes C))\otimes D \arrow[rr, "\alpha_{A,B\otimes C,D}"] & & A\otimes((B\otimes C)\otimes D) \arrow[d, "A\otimes\alpha_{B,C,D}"] \\ ((A\otimes B)\otimes C)\otimes D \arrow[u, "\alpha_{A,B,C}\otimes D"] \arrow[r, "\alpha_{A\otimes B,C,D}"'] & (A\otimes B)\otimes(C\otimes D) \arrow[r, "\alpha_{A,B,C\otimes D}"'] & A\otimes(B\otimes(C\otimes D)) \end{tikzcd}$$

A skew monoidal category with binary coproducts is *(binary) left-distributive* if the canonical morphism typed $(A \otimes C) + (B \otimes C) \to (A + B) \otimes C$ has an inverse $l : (A + B) \otimes C \to (A \otimes C) + (B \otimes C)$. We will be interested in skew monoidal categories with binary products and coproducts, which moreover are left-distributive. We simply call these distributive skew monoidal categories.

A *(strict) skew monoidal functor* $F : \mathbb{C} \to \mathbb{D}$ between skew monoidal categories $(\mathbb{C}, \mathsf{I}, \otimes)$ and $(\mathbb{D}, \mathsf{I}', \otimes')$ is a functor from $\mathbb{C}$ to $\mathbb{D}$ satisfying $F\mathsf{I} = \mathsf{I}'$ and $F(A \otimes B) = FA \otimes' FB$, also preserving the structural laws λ, ρ and α on the nose. This means that $F\lambda^{\mathbb{C}}_A = \lambda^{\mathbb{D}}_{FA}$, where $\lambda^{\mathbb{C}}$ and $\lambda^{\mathbb{D}}$ are left-unitors of $\mathbb{C}$ and $\mathbb{D}$ respectively, and similar equations hold for ρ and α. A skew monoidal functor is *distributive* if it also strictly preserves products, coproducts and (consequently also) left-distributivity.

The formulae, derivations and the equivalence relation $\stackrel{\circ}{=}$ of the sequent calculus determine a *syntactic* distributive skew monoidal category $\mathsf{FDSkM(At)}$ (an acronym for Free Distributive Skew Monoidal category on the set At). Its objects are formulae. The operations I and $\otimes$ are the logical connectives. The set of maps between objects A and B is the set of derivations $A \mid\ \vdash B$ quotiented by the equivalence relation $\stackrel{\circ}{=}$. The identity map on A is the equivalence class of ax_A, while composition is given by scut. The structural laws λ, ρ, α are all admissible. Products and coproducts are the additive connectives $\wedge$ and $\vee$. Left-distributivity follows from the logical rules of $\vee$ and $\otimes$.

Distributive skew monoidal categories form models of our sequent calculus. Moreover the sequent calculus, as a presentation of a distributive skew monoidal category, is the *initial* one among these models. Equivalently, $\mathsf{FDSkM(At)}$ is the *free* such category on the set At.

Theorem 3.1. *Let $\mathbb{D}$ be a distributive skew monoidal category. Given a function $F_{\mathsf{At}} : \mathsf{At} \to |\mathbb{D}|$ evaluating atomic formulae as objects of $\mathbb{D}$, there exists a unique distributive skew monoidal functor $F : \mathsf{FDSkM(At)} \to \mathbb{D}$ for which $FX = F_{\mathsf{At}}X$, for any atom X.*

The construction of the functor F and the proof of uniqueness proceed similarly to the proofs of Theorems 3.1 and 3.2 in [20].

4 A Focused Sequent Calculus with Tag Annotations

When oriented from left-to-right, the equations in (2) become a rewrite system, which is locally confluent and strongly normalizing, thus confluent with unique normal forms. Here we provide an explicit description of the normal forms of (1) wrt. this rewrite system.

For any sequent $S \mid \Gamma \vdash A$, a root-first proof search procedure can be defined as follows. First apply right invertible rules on the sequent until the principal connective of the succedent is non-negative, then apply left invertible rules until the stoup becomes either empty or non-positive. At this point, if we do not insist on focusing on a particular formula (either in the stoup or succedent, since no rule acts on formulae in context) as in Andreoli's focusing procedure [3], we obtain a sequent calculus with a reduced proof search space, that looks like this:

$$
\begin{array}{l}
\text{(right invertible)} \quad \dfrac{S \mid \Gamma \vdash_{\mathsf{RI}} A \quad S \mid \Gamma \vdash_{\mathsf{RI}} B}{S \mid \Gamma \vdash_{\mathsf{RI}} A \wedge B}\,\wedge\mathsf{R} \qquad \dfrac{S \mid \Gamma \vdash_{\mathsf{LI}} P}{S \mid \Gamma \vdash_{\mathsf{RI}} P}\,\mathsf{LI2RI} \\[2ex]
\text{(left invertible)} \quad \dfrac{- \mid \Gamma \vdash_{\mathsf{LI}} P}{\mathsf{I} \mid \Gamma \vdash_{\mathsf{LI}} P}\,\mathsf{IL} \qquad \dfrac{A \mid B, \Gamma \vdash_{\mathsf{LI}} P}{A \otimes B \mid \Gamma \vdash_{\mathsf{LI}} P}\,\otimes\mathsf{L} \\[2ex]
\qquad \dfrac{A \mid \Gamma \vdash_{\mathsf{LI}} P \quad B \mid \Gamma \vdash_{\mathsf{LI}} P}{A \vee B \mid \Gamma \vdash_{\mathsf{LI}} P}\,\vee\mathsf{L} \qquad \dfrac{T \mid \Gamma \vdash_{\mathsf{F}} P}{T \mid \Gamma \vdash_{\mathsf{LI}} P}\,\mathsf{F2LI} \\[2ex]
\text{(focusing)} \quad \dfrac{A \mid \Gamma \vdash_{\mathsf{LI}} P}{- \mid A, \Gamma \vdash_{\mathsf{F}} P}\,\mathsf{pass} \qquad \dfrac{}{X \mid \ \vdash_{\mathsf{F}} X}\,\mathsf{ax} \qquad \dfrac{}{- \mid \ \vdash_{\mathsf{F}} \mathsf{I}}\,\mathsf{IR} \\[2ex]
\dfrac{T \mid \Gamma \vdash_{\mathsf{RI}} A \quad - \mid \Delta \vdash_{\mathsf{RI}} B}{T \mid \Gamma, \Delta \vdash_{\mathsf{F}} A \otimes B}\,\otimes\mathsf{R} \qquad \dfrac{T \mid \Gamma \vdash_{\mathsf{RI}} A}{T \mid \Gamma \vdash_{\mathsf{F}} A \vee B}\,\vee\mathsf{R}_1 \qquad \dfrac{T \mid \Gamma \vdash_{\mathsf{RI}} B}{T \mid \Gamma \vdash_{\mathsf{F}} A \vee B}\,\vee\mathsf{R}_2 \\[2ex]
\qquad \dfrac{A \mid \Gamma \vdash_{\mathsf{LI}} P}{A \wedge B \mid \Gamma \vdash_{\mathsf{F}} P}\,\wedge\mathsf{L}_1 \qquad \dfrac{B \mid \Gamma \vdash_{\mathsf{LI}} P}{A \wedge B \mid \Gamma \vdash_{\mathsf{F}} P}\,\wedge\mathsf{L}_2
\end{array}
\tag{3}
$$

In the rules above, P is a non-negative formula, i.e. its principal connective is not $\wedge$, and T is a non-positive stoup (also called *irreducible*), i.e. it is not I and its principal connective is neither $\otimes$ nor $\vee$.

This calculus is too permissive. The same sequent $S \mid \Gamma \vdash_{\mathsf{RI}} A$ may have multiple derivations which correspond to $\stackrel{\circ}{=}$-related derivations in the original sequent calculus. This happens since certain sequents in phase $\vdash_{\mathsf{F}}$ can be alternatively proved by an application of a left non-invertible rule (pass, $\wedge\mathsf{L}_1$ and $\wedge\mathsf{L}_2$) or an application of a right non-invertible rule ($\otimes\mathsf{R}$, $\vee\mathsf{R}_1$ and $\vee\mathsf{R}_2$). As concrete examples, both sequents $- \mid X, Y \vdash_{\mathsf{F}} X \otimes Y$ and $X \wedge Y \mid \ \vdash_{\mathsf{F}} X \vee Y$ have multiple distinct proofs in this calculus, but their corresponding proofs in the original calculus are $\stackrel{\circ}{=}$-related.

In phase $\vdash_{\mathsf{F}}$, only non-invertible rules can be applied, so the question is: how to arrange the order between non-invertible rules without causing undesired non-determinism and losing completeness with respect to the sequent calculus in (1) and its equivalence relation $\stackrel{\circ}{=}$? Similarly to [20], our strategy is to prioritize left non-invertible rules over right ones, unless this does not lead to a valid derivation and the other way around is necessary. For example, consider the sequent $X \wedge Y \mid \ \vdash_{\mathsf{F}} (X \wedge Y) \vee Z$. Proof search fails if we apply $\wedge\mathsf{L}_i$ before $\vee\mathsf{R}_1$. A valid proof is obtained only when applying $\vee\mathsf{R}_1$ before $\wedge\mathsf{L}_i$. Rule sw is an abbreviation for the application of multiple consecutive phase switching rules.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{X \mid \ \vdash_{\mathsf{F}} X}\,\mathsf{ax}}{X \mid \ \vdash_{\mathsf{LI}} X}\,\mathsf{F2LI}}{X \wedge Y \mid \ \vdash_{\mathsf{F}} X}\,\wedge\mathsf{L}_1}{X \wedge Y \mid \ \vdash_{\mathsf{RI}} X}\,\mathsf{sw} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{Y \mid \ \vdash_{\mathsf{F}} Y}\,\mathsf{ax}}{Y \mid \ \vdash_{\mathsf{LI}} Y}\,\mathsf{F2LI}}{X \wedge Y \mid \ \vdash_{\mathsf{F}} Y}\,\wedge\mathsf{L}_2}{X \wedge Y \mid \ \vdash_{\mathsf{RI}} Y}\,\mathsf{sw}}{X \wedge Y \mid \ \vdash_{\mathsf{RI}} X \wedge Y}\,\wedge\mathsf{R}}{X \wedge Y \mid \ \vdash_{\mathsf{F}} (X \wedge Y) \vee Z}\,\vee\mathsf{R}_1
\tag{4}
$$

In this example it was possible to first apply $\vee\mathsf{R}_1$ since, after the application of $\wedge\mathsf{R}$, different left $\wedge$-rules are applied in different branches of the proof tree. If we would have applied the same rule $\wedge\mathsf{L}_1$ to both premises (imagine that $X = Y$ for this to be possible), then we could have obtained a $\stackrel{\circ}{=}$-equivalent derivation by moving the application of $\wedge\mathsf{L}_1$ to the bottom of the proof tree.

It is also possible that the two premises of $\wedge\mathsf{R}$ correspond to $\stackrel{\circ}{=}$-inequivalent derivations. For example, consider the following proof of sequent $- \mid \mathsf{I}\otimes X \vdash_{\mathsf{F}} ((I\otimes X)\wedge(\mathsf{I}\otimes X))\vee Y$:

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{IR}
\UnaryInfC{$- \mid\ \vdash_{\mathsf{F}} \mathsf{I}$}
\RightLabel{sw}
\UnaryInfC{$- \mid\ \vdash_{\mathsf{RI}} \mathsf{I}$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \mid\ \vdash_{\mathsf{F}} X$}
\RightLabel{$\mathsf{F2LI}$}
\UnaryInfC{$X \mid\ \vdash_{\mathsf{LI}} X$}
\RightLabel{pass}
\UnaryInfC{$- \mid X \vdash_{\mathsf{F}} X$}
\RightLabel{sw}
\UnaryInfC{$- \mid X \vdash_{\mathsf{RI}} X$}
\RightLabel{$\otimes\mathsf{R}$}
\BinaryInfC{$- \mid X \vdash_{\mathsf{F}} \mathsf{I}\otimes X$}
\RightLabel{$\mathsf{F2LI}$}
\UnaryInfC{$- \mid X \vdash_{\mathsf{LI}} \mathsf{I}\otimes X$}
\RightLabel{IL}
\UnaryInfC{$\mathsf{I} \mid X \vdash_{\mathsf{LI}} \mathsf{I}\otimes X$}
\RightLabel{$\otimes\mathsf{L}$}
\UnaryInfC{$\mathsf{I}\otimes X \mid\ \vdash_{\mathsf{LI}} \mathsf{I}\otimes X$}
\RightLabel{pass}
\UnaryInfC{$- \mid \mathsf{I}\otimes X \vdash_{\mathsf{F}} \mathsf{I}\otimes X$}
\RightLabel{sw}
\UnaryInfC{$- \mid \mathsf{I}\otimes X \vdash_{\mathsf{RI}} \mathsf{I}\otimes X$}
\AxiomC{}
\RightLabel{IR}
\UnaryInfC{$- \mid\ \vdash_{\mathsf{F}} \mathsf{I}$}
\RightLabel{sw}
\UnaryInfC{$- \mid\ \vdash_{\mathsf{RI}} \mathsf{I}$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \mid\ \vdash_{\mathsf{F}} X$}
\RightLabel{$\mathsf{F2LI}$}
\UnaryInfC{$X \mid\ \vdash_{\mathsf{LI}} X$}
\RightLabel{pass}
\UnaryInfC{$- \mid X \vdash_{\mathsf{F}} X$}
\RightLabel{$\mathsf{F2LI}$}
\UnaryInfC{$- \mid X \vdash_{\mathsf{LI}} X$}
\RightLabel{IL}
\UnaryInfC{$\mathsf{I} \mid X \vdash_{\mathsf{LI}} X$}
\RightLabel{$\otimes\mathsf{L}$}
\UnaryInfC{$\mathsf{I}\otimes X \mid\ \vdash_{\mathsf{LI}} X$}
\RightLabel{pass}
\UnaryInfC{$- \mid \mathsf{I}\otimes X \vdash_{\mathsf{F}} X$}
\RightLabel{sw}
\UnaryInfC{$- \mid \mathsf{I}\otimes X \vdash_{\mathsf{RI}} X$}
\RightLabel{$\otimes\mathsf{R}$}
\BinaryInfC{$- \mid \mathsf{I}\otimes X \vdash_{\mathsf{F}} \mathsf{I}\otimes X$}
\RightLabel{sw}
\UnaryInfC{$- \mid \mathsf{I}\otimes X \vdash_{\mathsf{RI}} \mathsf{I}\otimes X$}
\RightLabel{$\wedge\mathsf{R}$}
\BinaryInfC{$- \mid \mathsf{I}\otimes X \vdash_{\mathsf{RI}} (\mathsf{I}\otimes X)\wedge(\mathsf{I}\otimes X)$}
\RightLabel{$\vee\mathsf{R}_1$}
\UnaryInfC{$- \mid \mathsf{I}\otimes X \vdash_{\mathsf{F}} ((\mathsf{I}\otimes X)\wedge(\mathsf{I}\otimes X))\vee Y$}
\end{prooftree}
$$

In this case, the right non-invertible rule $\vee\mathsf{R}_1$ must be applied before the left non-invertible rule pass. This is because pass is used in the proof of the left branch of $\wedge\mathsf{R}$, but it is not used in the proof of the right branch, $\otimes\mathsf{R}$ is used instead. If both left and right proofs would have used pass (for example, they could have been the same exact proof), then it would have been possible to apply pass before $\vee\mathsf{R}_1$.

In general, a right non-invertible rule should be applied before a left non-invertible one if, after the possible application of some $\wedge\mathsf{R}$ rules, either: (i) a right non-invertible rule or the ax rule is applied to one of the premises; (ii) $\wedge\mathsf{L}_1$ and $\wedge\mathsf{L}_2$ are applied to different premises. Therefore, we have to make sure that in the focused sequent calculus, after the application of a right non-invertible rule, not all premises use the same left non-invertible rule, because in this case the latter rule could be applied first.

In order to keep track of this, we use a system of *tag annotations* and we introduce new phases of proof search where sequents are annotated by *list of tags*. There are four tags: $\mathbb{P},\mathbb{C}_1,\mathbb{C}_2,\mathbb{R}$. Intuitively, they respectively correspond to rules pass, $\wedge\mathsf{L}_1$, $\wedge\mathsf{L}_2$ and all the remaining non-invertible rules in phase $\vdash_{\mathsf{F}}$. A list of tags l is called *valid* if it is non-empty and either (i) $\mathbb{R}\in l$ or (ii) both $\mathbb{C}_1 \in l$ and $\mathbb{C}_2 \in l$.

Derivations in the focused sequent calculus with tag annotations are generated by the rules

(right invertible)
$$\frac{S \mid \Gamma \vdash_{\mathsf{RI}}^{l_1?} A \quad S \mid \Gamma \vdash_{\mathsf{RI}}^{l_2?} B}{S \mid \Gamma \vdash_{\mathsf{RI}}^{l_1?,l_2?} A\wedge B}\ \wedge\mathsf{R} \qquad \frac{S \mid \Gamma \vdash_{\mathsf{LI}}^{t?} P}{S \mid \Gamma \vdash_{\mathsf{RI}}^{t?} P}\ \mathsf{LI2RI}$$

(left invertible)
$$\frac{- \mid \Gamma \vdash_{\mathsf{LI}} P}{\mathsf{I} \mid \Gamma \vdash_{\mathsf{LI}} P}\ \mathsf{IL} \qquad \frac{A \mid B,\Gamma \vdash_{\mathsf{LI}} P}{A\otimes B \mid \Gamma \vdash_{\mathsf{LI}} P}\ \otimes\mathsf{L}$$
$$\frac{A \mid \Gamma \vdash_{\mathsf{LI}} P \quad B \mid \Gamma \vdash_{\mathsf{LI}} P}{A\vee B \mid \Gamma \vdash_{\mathsf{LI}} P}\ \vee\mathsf{L} \qquad \frac{T \mid \Gamma \vdash_{\mathsf{F}}^{t?} P}{T \mid \Gamma \vdash_{\mathsf{LI}}^{t?} P}\ \mathsf{F2LI}$$

(focusing)
$$\frac{A \mid \Gamma \vdash_{\mathsf{LI}} P}{- \mid A,\Gamma \vdash_{\mathsf{F}}^{\mathbb{P}?} P}\ \mathsf{pass} \qquad \frac{}{X \mid\ \vdash_{\mathsf{F}}^{\mathbb{R}?} X}\ \mathsf{ax} \qquad \frac{}{- \mid\ \vdash_{\mathsf{F}}^{\mathbb{R}?} \mathsf{I}}\ \mathsf{IR} \tag{5}$$
$$\frac{T \mid \Gamma \vdash_{\mathsf{RI}}^{l} A \quad - \mid \Delta \vdash_{\mathsf{RI}} B \quad l \text{ valid}}{T \mid \Gamma,\Delta \vdash_{\mathsf{F}}^{\mathbb{R}?} A\otimes B}\ \otimes\mathsf{R} \qquad \frac{T \mid \Gamma \vdash_{\mathsf{RI}}^{l} A \quad l \text{ valid}}{T \mid \Gamma \vdash_{\mathsf{F}}^{\mathbb{R}?} A\vee B}\ \vee\mathsf{R}_1 \qquad \frac{T \mid \Gamma \vdash_{\mathsf{RI}}^{l} B \quad l \text{ valid}}{T \mid \Gamma \vdash_{\mathsf{F}}^{\mathbb{R}?} A\vee B}\ \vee\mathsf{R}_2$$
$$\frac{A \mid \Gamma \vdash_{\mathsf{LI}} P}{A\wedge B \mid \Gamma \vdash_{\mathsf{F}}^{\mathbb{C}_1?} P}\ \wedge\mathsf{L}_1 \qquad \frac{B \mid \Gamma \vdash_{\mathsf{LI}} P}{A\wedge B \mid \Gamma \vdash_{\mathsf{F}}^{\mathbb{C}_2?} P}\ \wedge\mathsf{L}_2$$

We use l for lists of tags and t for single tags. The notation $l?$ indicates that the sequent is either untagged or assigned the list of tags l. Similarly for notation $t?$. We discuss the proof search procedures of untagged and tagged sequents separately. The proof search of a sequent $S \mid \Gamma \vdash_{\mathsf{RI}} A$ proceeds as follows:

($\vdash_{\mathsf{RI}}$) We apply the right invertible rule $\wedge\mathsf{R}$ eagerly to decompose the succedent until its principal connective is not $\wedge$, then we move to the left invertible phase $\vdash_{\mathsf{LI}}$ with an application of $\mathsf{LI2RI}$.

($\vdash_{\mathsf{LI}}$) We apply left invertible rules until the stoup becomes irreducible, then move to the focusing phase $\vdash_{\mathsf{F}}$ with an application of $\mathsf{F2LI}$.

($\vdash_{\mathsf{F}}$) We apply one of the remaining rules. Since the sequents are not marked by tags at this point, rules pass, ax , IR and $\wedge\mathsf{L}_i$ can be directly applied when stoups, contexts and succedents are of the appropriate form. If we decide to apply a right non-invertible rule, we need to come up with a valid list of tags l and subsequently continue proof search in tagged right invertible phase $\vdash^{l}_{\mathsf{RI}}$, which is described below. Notice that only the first premise of $\otimes\mathsf{R}$ is tagged, the second premise is not, i.e. its proof search continues in phase $\vdash_{\mathsf{RI}}$.

The proof search of a sequent $T \mid \Gamma \vdash^{l}_{\mathsf{RI}} A$ proceeds as follows (notice that at this point in proof search the stoup T is necessarily irreducible):

($\vdash^{l}_{\mathsf{RI}}$) We apply the $\wedge\mathsf{R}$ rule to decompose the succedent and split the list of tags carefully until the succedent becomes non-negative and the list of tags becomes a singleton t, then we move to phase $\vdash^{t}_{\mathsf{LI}}$ with an application of $\mathsf{LI2RI}$.

($\vdash^{t}_{\mathsf{LI}}$) Since the stoup is either empty or a negative formula, we immediately switch to phase $\vdash_{\mathsf{F}}$ with an application of $\mathsf{F2LI}$. This motivates why sequents in rules IL, $\otimes\mathsf{L}$, $\vee\mathsf{L}$ are not tagged.

($\vdash^{t}_{\mathsf{F}}$) If $t = \mathbb{R}$ we can apply either ax, IR or another right non-invertible rule. Again, when applying right non-invertible rules we need to come up with a new valid list of tags. Left non-invertible rules can be applied only when the tag is correct, i.e. pass with tag $\mathbb{P}$, $\wedge\mathsf{L}_1$ with tag $\mathbb{C}_1$, and $\wedge\mathsf{L}_2$ with tag $\mathbb{C}_2$.

The derivation in (4) can be reconstructed in the focused calculus with tag annotations.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{X \mid\ \vdash_{\mathsf{F}} X}\,\mathsf{ax}}{X \mid\ \vdash_{\mathsf{LI}} X}\,\mathsf{sw}}{X \wedge Y \mid\ \vdash^{\mathbb{C}_1}_{\mathsf{F}} X}\,\wedge\mathsf{L}_1}{X \wedge Y \mid\ \vdash^{\mathbb{C}_1}_{\mathsf{RI}} X}\,\mathsf{sw} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{Y \mid\ \vdash_{\mathsf{F}} Y}\,\mathsf{ax}}{Y \mid\ \vdash_{\mathsf{LI}} Y}\,\mathsf{sw}}{X \wedge Y \mid\ \vdash^{\mathbb{C}_2}_{\mathsf{F}} Y}\,\wedge\mathsf{L}_2}{X \wedge Y \mid\ \vdash^{\mathbb{C}_2}_{\mathsf{RI}} Y}\,\mathsf{sw}}{X \wedge Y \mid\ \vdash^{\mathbb{C}_1,\mathbb{C}_2}_{\mathsf{RI}} X \wedge Y}\,\wedge\mathsf{R}}{X \wedge Y \mid\ \vdash_{\mathsf{F}} (X \wedge Y) \vee Z}\,\vee\mathsf{R}_1 \tag{6}$$

Notice that the list of tags is not predetermined when a right non-invertible rule is applied, we have to come up with one ourselves. Practically, the list l can be computed by continuing proof search until, in each branch, we hit the first application of a rule in phase $\vdash_{\mathsf{F}}$, each with its own (necessarily uniquely determined) single tag t. Take l as the concatenation of the resulting ts and check whether it is valid. If it is not, backtrack and apply a left non-invertible rule instead.

Theorem 4.1. *The focused sequent calculus with tag annotations in (5) is sound and complete with respect to the sequent calculus in (1).*

Soundness is immediate because there exist functions $\mathsf{emb}_{ph} : S \mid \Gamma \vdash^{l?}_{ph} A \to S \mid \Gamma \vdash A$, for all $ph \in \{\mathsf{RI}, \mathsf{LI}, \mathsf{F}\}$, which erase all phase and tag annotations. Completeness follows from the fact that the following rules are all admissible:

$$\frac{- \mid \Gamma \vdash_{\mathsf{RI}} C}{\mathsf{I} \mid \Gamma \vdash_{\mathsf{RI}} C}\ \mathsf{IL}^{\mathsf{RI}} \quad \frac{A \mid B,\Gamma \vdash_{\mathsf{RI}} C}{A \otimes B \mid \Gamma \vdash_{\mathsf{RI}} C}\ \otimes\mathsf{L}^{\mathsf{RI}} \quad \frac{A \mid \Gamma \vdash_{\mathsf{RI}} C}{- \mid A,\Gamma \vdash_{\mathsf{RI}} C}\ \mathsf{pass}^{\mathsf{RI}} \quad \frac{}{A \mid \ \vdash_{\mathsf{RI}} A}\ \mathsf{ax}^{\mathsf{RI}} \quad \frac{}{- \mid \ \vdash_{\mathsf{RI}} \mathsf{I}}\ \mathsf{IR}^{\mathsf{RI}}$$

$$\frac{A \mid \Gamma \vdash_{\mathsf{RI}} C \quad B \mid \Gamma \vdash_{\mathsf{RI}} C}{A \vee B \mid \Gamma \vdash_{\mathsf{RI}} C}\ \vee\mathsf{L}^{\mathsf{RI}} \qquad \frac{S \mid \Gamma \vdash_{\mathsf{RI}} A \quad - \mid \Delta \vdash_{\mathsf{RI}} B}{S \mid \Gamma,\Delta \vdash_{\mathsf{RI}} A \otimes B}\ \otimes\mathsf{R}^{\mathsf{RI}} \tag{7}$$

$$\frac{A \mid \Gamma \vdash_{\mathsf{RI}} C}{A \wedge B \mid \Gamma \vdash_{\mathsf{RI}} C}\ \wedge\mathsf{L}_1^{\mathsf{RI}} \quad \frac{B \mid \Gamma \vdash_{\mathsf{RI}} C}{A \wedge B \mid \Gamma \vdash_{\mathsf{RI}} C}\ \wedge\mathsf{L}_2^{\mathsf{RI}} \quad \frac{S \mid \Gamma \vdash_{\mathsf{RI}} A}{S \mid \Gamma \vdash_{\mathsf{RI}} A \vee B}\ \vee\mathsf{R}_1^{\mathsf{RI}} \quad \frac{S \mid \Gamma \vdash_{\mathsf{RI}} B}{S \mid \Gamma \vdash_{\mathsf{RI}} A \vee B}\ \vee\mathsf{R}_2^{\mathsf{RI}}$$

The admissibility of the rules in (7), apart from the right non-invertible ones, is proved by structural induction on derivations. The same strategy cannot be applied to right non-invertible rules. For example, if the premise of $\vee\mathsf{R}_1^{\mathsf{RI}}$ ends with an application of $\wedge\mathsf{R}$, we get immediately stuck:

$$\frac{\dfrac{\overset{f}{S \mid \Gamma \vdash_{\mathsf{RI}} A'} \quad \overset{g}{S \mid \Gamma \vdash_{\mathsf{RI}} B'}}{S \mid \Gamma \vdash_{\mathsf{RI}} A' \wedge B'}\ \wedge\mathsf{R}}{S \mid \Gamma \vdash_{\mathsf{RI}} (A' \wedge B') \vee B}\ \vee\mathsf{R}_1^{\mathsf{RI}} \quad = \quad ??$$

The inductive hypothesis applied to f and g would produce wrong sequents for the target conclusion. This is fixed by proving the admissibility of more general rules. In order to state and prove this, we need to first introduce a few lemmata. The first one shows that applying several $\wedge\mathsf{R}$ rules in one step is admissible.

Let $\mathsf{conj}(A)$ be the list of formulae obtained by decomposing additive conjunctions $\wedge$ in the formula A. Concretely, $\mathsf{conj}(A) = \mathsf{conj}(A'),\mathsf{conj}(B')$ if $A = A' \wedge B'$ and $\mathsf{conj}(A) = A$ otherwise.

Lemma 4.2. *The following rules*

$$\frac{[T \mid \Gamma \vdash_{\mathsf{F}}^{t_i} P_i]_{i\in[1,\dots,n]}}{T \mid \Gamma \vdash_{\mathsf{RI}}^{l} A}\ \wedge\mathsf{R}_t^* \qquad \frac{[T \mid \Gamma \vdash_{\mathsf{LI}} P_i]_{i\in[1,\dots,n]}}{T \mid \Gamma \vdash_{\mathsf{RI}} A}\ \wedge\mathsf{R}^*$$

are admissible, where $\mathsf{conj}(A) = [P_1,\dots,P_n]$ *and* $l = [t_1,\dots,t_n]$.

Proof. We show the case of $\wedge\mathsf{R}_t^*$, the other one is similar. Let $fs : [T \mid \Gamma \vdash_{\mathsf{F}}^{t_i} P_i]_i$ be a list of derivations. The proof proceeds by induction on A.

- If $A \neq A' \wedge B'$, then fs consists of a single derivation f. Define $\wedge\mathsf{R}_t^*\ fs = \mathsf{F2LI}\ (\mathsf{LI2RI}\ f)$.
- If $A = A' \wedge B'$, then there exist lists of derivations $fs_1 : [T \mid \Gamma \vdash_{\mathsf{F}}^{t_i} A_i']_{i\in[1,\dots,m]}$ and $fs_2 : [T \mid \Gamma \vdash_{\mathsf{F}}^{t_i} B_i']_{i\in[m+1,\dots,n]}$, and lists of tags $l_1 = t_1,\dots,t_m$ and $l_2 = t_{m+1},\dots,t_n$, so that fs is the concatenation of fs_1 and fs_2 and l is the concatenation of l_1 and l_2. Apply $\wedge\mathsf{R}$ at the bottom, then proceed recursively:

$$\frac{\overset{fs}{[T \mid \Gamma \vdash_{\mathsf{F}}^{t_i} P_i]_{i\in[1,\dots,n]}}}{T \mid \Gamma \vdash^{l} A' \wedge B'}\ \wedge\mathsf{R}_t^* \quad = \quad \frac{\dfrac{\overset{fs_1}{[T \mid \Gamma \vdash_{\mathsf{F}}^{t_i} P_i]_{i\in[1,\dots,m]}}}{T \mid \Gamma \vdash_{\mathsf{RI}}^{l_1} A'}\ \wedge\mathsf{R}_t^* \quad \dfrac{\overset{fs_2}{[T \mid \Gamma \vdash_{\mathsf{F}}^{t_i} P_i]_{i\in[m+1,\dots,n]}}}{T \mid \Gamma \vdash_{\mathsf{RI}}^{l_2} B'}\ \wedge\mathsf{R}_t^*}{T \mid \Gamma \vdash_{\mathsf{RI}}^{l_1,l_2} A' \wedge B'}\ \wedge\mathsf{R}$$

□

The second lemma corresponds to the invertibility of phase $\vdash_{\mathsf{RI}}$.

Lemma 4.3. *Given $f : S \mid \Gamma \vdash_{\mathsf{RI}} A$, there is a list of derivations $fs : [S \mid \Gamma \vdash_{\mathsf{LI}} P_i]_{i\in[1,\dots,n]}$ with $f = \wedge\mathsf{R}^* fs$.*

Proof. The proof proceeds by structural induction on $f : S \mid \Gamma \vdash_{\mathsf{RI}} A$.

- If $f = \mathsf{LI2RI}\ f_1$, then A is non-negative. Take fs as the singleton list consisting exclusively of f_1.
- If $f = \wedge\mathsf{R}\ (f_1, f_2)$, then by inductive hypothesis we have $fs_1 : [S \mid \Gamma \vdash_{\mathsf{LI}} P_i]_{i\in[1,\dots,n]}$ and $fs_2 : [S \mid \Gamma \vdash_{\mathsf{LI}} P'_i]_{i\in[1,\dots,m]}$. Take fs as the concatenation of fs_1 and fs_2.

□

Proposition 4.4. *The following rules*

$$\frac{\overset{fs}{[S \mid \Gamma \vdash_{\mathsf{LI}} P_i]_{i\in[1,\dots,n]}}}{S \mid \Gamma \vdash_{\mathsf{LI}} A \vee B}\ \vee\mathsf{R}_1^{\mathsf{LI}} \qquad \frac{\overset{fs}{[S \mid \Gamma \vdash_{\mathsf{LI}} Q_i]_{i\in[1,\dots,m]}}}{S \mid \Gamma \vdash_{\mathsf{LI}} A \vee B}\ \vee\mathsf{R}_2^{\mathsf{LI}} \qquad \frac{\overset{fs}{[S \mid \Gamma \vdash_{\mathsf{LI}} P_i]_{i\in[1,\dots,n]}} \quad - \mid \Delta \vdash_{\mathsf{RI}} B'}{S \mid \Gamma, \Delta \vdash_{\mathsf{LI}} A \otimes B'}\ \otimes\mathsf{R}^{\mathsf{LI}}$$

are admissible, where $\mathsf{conj}(A) = [P_1, \dots, P_n]$ *and* $\mathsf{conj}(B) = [Q_1, \dots, Q_m]$.

Proof. The list of derivations fs is non-empty, so we let $fs = [f_1, fs']$. We proceed by induction on f_1. We only present the proof for $\vee\mathsf{R}_1^{\mathsf{LI}}$, the admissibility of $\vee\mathsf{R}_2^{\mathsf{LI}}$ and $\otimes\mathsf{R}^{\mathsf{LI}}$ is proved similarly.

If f_1 ends with the application of a left invertible rule, then all the derivations in fs' necessarily end with the same rule as well. Therefore, we permute this rule with $\vee\mathsf{R}_1^{\mathsf{LI}}$ and apply the inductive hypothesis.

If $f_1 = \mathsf{F2LI}\ f'_1$, then all the derivations in fs' necessarily end with $\mathsf{F2LI}$ as well. We generate a list of tags l by examining the shape of each derivation in fs: we add $\mathbb{P}$ for each pass, $\mathbb{C}_1$ for each $\wedge\mathsf{L}_1$, $\mathbb{C}_2$ for each $\wedge\mathsf{L}_2$ and $\mathbb{R}$ for the remaining rules. There are two possibilities:

- The resulting list l is valid. We switch to phase $\vdash_{\mathsf{F}}$ and apply $\vee\mathsf{R}_1^{\mathsf{LI}}$ followed by $\wedge\mathsf{R}_t^*$:

$$\frac{\dfrac{\overset{fs^*}{[T \mid \Gamma \vdash_{\mathsf{F}} P_i]_{i\in[1,\dots,n]}}}{[T \mid \Gamma \vdash_{\mathsf{LI}} P_i]_{i\in[1,\dots,n]}}\ [\mathsf{F2LI}]}{T \mid \Gamma \vdash_{\mathsf{LI}} A \vee B}\ \vee\mathsf{R}_1^{\mathsf{LI}} \quad = \quad \frac{\dfrac{\dfrac{\overset{fs^{*\prime}}{[T \mid \Gamma \vdash_{\mathsf{F}}^{t_i} P_i]_{i\in[1,\dots,n]}}}{T \mid \Gamma \vdash_{\mathsf{RI}}^{l} A}\ \wedge\mathsf{R}_t^*}{T \mid \Gamma \vdash_{\mathsf{F}} A \vee B}\ \vee\mathsf{R}_1}{T \mid \Gamma \vdash_{\mathsf{LI}} A \vee B}\ \mathsf{F2LI}$$

  A rule wrapped in square brackets, like $[\mathsf{F2LI}]$ above, denotes the application of the rule to the conclusion of each derivation in the list. The list of derivations fs^* is obtained from fs by applying $[\mathsf{F2LI}]$, i.e. $fs = [\mathsf{F2LI}]\, fs^*$, while $fs^{*\prime}$ is a list of derivations whose conclusions are tagged version of those in fs^*, which can be easily constructed from fs^*.

- The list l is invalid. In this case, all elements in fs end with the same left non-invertible rule, so we permute the rule down with $\vee\mathsf{R}_1^{\mathsf{LI}}$ and continue recursively. Here is an example where all derivations in fs end with an application of pass, i.e. $fs = [\mathsf{F2LI}]\,([\mathsf{pass}]\, fs^*)$:

$$\frac{\dfrac{\dfrac{\overset{fs^*}{[A' \mid \Gamma \vdash_{\mathsf{LI}} P_i]_{i\in[1,\dots,n]}}}{[- \mid A', \Gamma \vdash_{\mathsf{LI}} P_i]_{i\in[1,\dots,n]}}\ [\mathsf{pass}]}{[- \mid A', \Gamma \vdash_{\mathsf{LI}} P_i]_{i\in[1,\dots,n]}}\ [\mathsf{F2LI}]}{- \mid A', \Gamma \vdash_{\mathsf{LI}} A \vee B}\ \vee\mathsf{R}_1^{\mathsf{LI}} \quad = \quad \frac{\dfrac{\dfrac{\overset{fs^*}{[A' \mid \Gamma \vdash_{\mathsf{LI}} P_i]_{i\in[1,\dots,n]}}}{A' \mid \Gamma \vdash_{\mathsf{LI}} A \vee B}\ \vee\mathsf{R}_1^{\mathsf{LI}}}{- \mid A', \Gamma \vdash_{\mathsf{F}} A \vee B}\ \mathsf{pass}}{- \mid A', \Gamma \vdash_{\mathsf{LI}} A \vee B}\ \mathsf{F2LI}$$

□

Finally, a right non-invertible rule in (7) is defined as follows: first invert its premises (for $\otimes\mathsf{R}^{\mathsf{RI}}$, only the left premise) using Lemma 4.3. Then apply the corresponding generalized rule in Proposition 4.4.

We can construct a function $\mathsf{focus} : S \mid \Gamma \vdash A \to S \mid \Gamma \vdash_{\mathsf{RI}} A$ by structural recursion on the input derivation. Each inference rule in (1) is sent to the corresponding admissible rule in (7). For example, $\mathsf{focus}\ (\vee\mathsf{R}_1\ f) = \vee\mathsf{R}_1^{\mathsf{RI}}\ (\mathsf{focus}\ f)$. Furthermore, it can be shown that $\mathsf{emb}_{\mathsf{RI}}$ and focus are each other inverses, in the sense made precise by the following theorem.

Theorem 4.5. *The functions* $\mathsf{emb}_{\mathsf{RI}}$ *and* focus *define a bijective correspondence between the set of derivations of* $S \mid \Gamma \vdash A$ *quotiented by the equivalence relation* $\overset{\circ}{=}$ *and the set of derivations of* $S \mid \Gamma \vdash_{\mathsf{RI}} A$*:*

- *For all* $f, g : S \mid \Gamma \vdash A$*, if* $f \overset{\circ}{=} g$ *then* $\mathsf{focus}\ f = \mathsf{focus}\ g$.
- *For all* $f : S \mid \Gamma \vdash A$*,* $\mathsf{emb}_{\mathsf{RI}}\ (\mathsf{focus}\ f) \overset{\circ}{=} f$.
- *For all* $f : S \mid \Gamma \vdash_{\mathsf{RI}} A$*,* $\mathsf{focus}\ (\mathsf{emb}_{\mathsf{RI}}\ f) = f$.

Proof. The first bullet is proved by structural induction on the given equality proof $e : f \overset{\circ}{=} g$. The other bullets are proved by structural induction on f. See the associated Agda formalization for details. □

5 Extensions of the Logic

We now discuss some extensions of the sequent calculus and the focusing strategy.

5.1 Additive Units

The sequent calculus in (1) can be made "fully" additive by including two units $\top$ and $\bot$ (the latter named 0 in linear logic literature), two new introduction rules and two new generating equations:

$$\frac{}{S \mid \Gamma \vdash \top}\ \top\mathsf{R} \qquad \frac{}{\bot \mid \Gamma \vdash C}\ \bot\mathsf{L}$$

$$\begin{array}{lll} f & \overset{\circ}{=} \top\mathsf{R} & (f : S \mid \Gamma \vdash \top) \\ f & \overset{\circ}{=} \bot\mathsf{L} & (f : \bot \mid \Gamma \vdash C) \end{array}$$

In the focused sequent calculus we add $\top\mathsf{R}$ in phase $\vdash_{\mathsf{RI}}$ and $\bot\mathsf{L}$ in phase $\vdash_{\mathsf{LI}}$, so that they can be applied as early as possible. We include a new tag $\mathbb{T}$ for $\top$. The validity condition for lists of tags is updated as follows: (*i*) $\mathbb{T}$ or $\mathbb{R} \in l$ or (*ii*) both $\mathbb{C}_1 \in l$ and $\mathbb{C}_2 \in l$.

$$\frac{}{S \mid \Gamma \vdash_{\mathsf{RI}}^{\mathbb{T}?} \top}\ \top\mathsf{R} \qquad \frac{}{\bot \mid \Gamma \vdash_{\mathsf{LI}} P}\ \bot\mathsf{L}$$

Categorical models of the extended sequent calculus are the distributive monoidal categories of Section 3 with additionally a terminal and an initial object, which moreover satisfy a *(nullary) left-distributivity* (or *absorption*) condition: the canonical morphism typed $0 \to 0 \otimes C$ has an inverse $k : 0 \otimes C \to 0$. The latter is used in the interpretation of the rule $\bot\mathsf{L}$.

5.2 Skew Exchange

Following [24], we consider a "skew" commutative extension of the sequent calculus in (1) obtained by adding a rule swapping adjacent formulae in context:

$$\frac{S \mid \Gamma, A, B, \Delta \vdash C}{S \mid \Gamma, B, A, \Delta \vdash C}\ \mathsf{ex}$$

Note that exchanging the formula in the stoup, whenever the latter is non-empty, with a formula in context is not allowed. The new rule ex comes with additional generating equations for the congruence relation $\doteq$:

$$\begin{array}{rcll}
\mathsf{ex}_{B,A}(\mathsf{ex}_{A,B}f) & \doteq & f & (f : S \mid \Gamma,A,B,\Delta \vdash C) \\
\mathsf{ex}_{A,B}(\mathsf{ex}_{A,D}(\mathsf{ex}_{B,D}f)) & \doteq & \mathsf{ex}_{B,D}(\mathsf{ex}_{A,D}(\mathsf{ex}_{A,B}f)) & (f : S \mid \Gamma,A,B,D,\Delta \vdash C) \\
\wedge\mathsf{L}_i\ (\mathsf{ex}_{A,B}\ f) & \doteq & \mathsf{ex}_{A,B}\ (\wedge\mathsf{L}_i\ f) & (f : A' \mid \Gamma,A,B,\Delta \vdash C) \\
\wedge\mathsf{R}\ (\mathsf{ex}_{A,B}\ f,\mathsf{ex}_{A,B}\ g) & \doteq & \mathsf{ex}_{A,B}\ (\wedge\mathsf{R}\ (f,g)) & (f : S \mid \Gamma,A,B,\Delta \vdash A', g : S \mid \Gamma,A,B,\Delta \vdash B') \\
\vee\mathsf{L}\ (\mathsf{ex}_{A,B}\ f,\mathsf{ex}_{A,B}\ g) & \doteq & \mathsf{ex}_{A,B}\ (\vee\mathsf{L}\ (f,g)) & (f : A' \mid \Gamma,A,B,\Delta \vdash C, g : B' \mid \Gamma,A,B,\Delta \vdash C) \\
\vee\mathsf{R}_i\ (\mathsf{ex}_{A,B}\ f) & \doteq & \mathsf{ex}_{A,B}\ (\vee\mathsf{R}_i\ f) & (f : S \mid \Gamma,A,B \vdash A') \\
\mathsf{ex}_{A,B}(\mathsf{ex}_{A',B'}f) & \doteq & \mathsf{ex}_{A',B'}(\mathsf{ex}_{A,B}f) & (f : S \mid \Gamma,A,B,\Delta,A',B',\Lambda \vdash C)
\end{array}$$

The first equation states that swapping the same two formulae twice yields the same result as doing nothing. The second equation corresponds to the Yang-Baxter equation. The remaining equations are permutative conversions. We left out permutative conversions describing the relationship between ex and the rules pass, IL, ⊗L and ⊗R, which can be found in [24, Fig. 2].

The resulting sequent calculus enjoys categorical semantics in distributive skew *symmetric* monoidal categories, that possess a natural isomorphism $s_{A,B,C} : A \otimes (B \otimes C) \to A \otimes (C \otimes B)$ representing a form of "skew symmetry" involving three objects instead of two [6].

The focused sequent calculus is extended with a new phase $\vdash_{\mathsf{C}}$ (for 'context') where the exchange rule can be applied. Rule ⊗L has to be modified, since we need to give the possibility to move the formula B to a different position in the context.

$$\frac{S \mid \Gamma \,\vdots\, \Delta,A,\Lambda \vdash_{\mathsf{C}} C}{S \mid \Gamma,A \,\vdots\, \Delta,\Lambda \vdash_{\mathsf{C}} C}\ \mathsf{ex} \qquad \frac{S \mid \Gamma \vdash_{\mathsf{RI}} C}{S \mid \quad \vdots\, \Gamma \vdash_{\mathsf{C}} C}\ \mathsf{RI2C} \qquad \frac{A \mid B \,\vdots\, \Gamma \vdash_{\mathsf{C}} P}{A \otimes B \mid \Gamma \vdash_{\mathsf{LI}} P}\ \otimes\mathsf{L}$$

Root-first proof search now begins in the new phase $\vdash_{\mathsf{C}}$, where formulae in context are permuted. We start with a sequent $S \mid \Gamma \,\vdots\, \quad \vdash_{\mathsf{C}} C$ and end with a sequent $S \mid \quad \vdots\, \Gamma' \vdash_{\mathsf{C}} C$ where Γ' is a permutation of Γ. In the process, the context is divided into two parts $\Gamma \,\vdots\, \Delta$, where the formulae in Γ are ready to be moved while those in Δ have already been placed in their final position. Once all formulae in Γ have been moved, we switch to phase $\vdash_{\mathsf{RI}}$ with an application of rule RI2C. Note that sequents in phase $\vdash_{\mathsf{C}}$ are not marked by list of tags, since after the application of right non-invertible rules there is no need to further permute formulae in context. Moreover, no new formulae can appear in context via applications of rule ⊗L, since the stoup is irreducible at this point.

As already mentioned, rule ⊗L has been modified. Its premise is now a sequent in phase $\vdash_{\mathsf{C}}$, which allows a further application of ex for the relocation of the formula B to a different position in the context.

5.3 Linear implication

Finally, we consider a deductive system for a skew version of Lambek calculus with additive conjunction and disjunction. This is obtained by extending the sequent calculus in (1) with a linear implication $\multimap$ and two introduction rules:

$$\frac{- \mid \Gamma \vdash A \quad B \mid \Delta \vdash C}{A \multimap B \mid \Gamma,\Delta \vdash C}\ \multimap\mathsf{L} \qquad \frac{S \mid \Gamma,A \vdash B}{S \mid \Gamma \vdash A \multimap B}\ \multimap\mathsf{R}$$

The presence of $\multimap$ requires the extension of the congruence relation $\stackrel{\circ}{=}$ with additional generating equations: an η-conversion and more permutative conversions.

$$\begin{array}{rll}
\mathsf{ax}_{A\multimap B} & \stackrel{\circ}{=} {\multimap}\mathsf{R}\ ({\multimap}\mathsf{L}\ (\mathsf{pass}\ \mathsf{ax}_A, \mathsf{ax}_B)) & \\
\otimes\mathsf{R}\ ({\multimap}\mathsf{L}\ (f,g),h) & \stackrel{\circ}{=} {\multimap}\mathsf{L}\ (f, \otimes\mathsf{R}\ (g,h)) & (f : - \mid \Gamma \vdash A', g : B' \mid \Delta \vdash A, h : - \mid \Lambda \vdash B) \\
\mathsf{pass}\ ({\multimap}\mathsf{R}\ f) & \stackrel{\circ}{=} {\multimap}\mathsf{R}\ (\mathsf{pass}\ f) & (f : A' \mid \Gamma, A \vdash B) \\
\mathsf{IL}\ ({\multimap}\mathsf{R}\ f) & \stackrel{\circ}{=} {\multimap}\mathsf{R}\ (\mathsf{IL}\ f) & (f : - \mid \Gamma, A \vdash B) \\
\otimes\mathsf{L}\ ({\multimap}\mathsf{R}\ f) & \stackrel{\circ}{=} {\multimap}\mathsf{R}\ (\otimes\mathsf{L}\ f) & (f : A' \mid B', \Gamma, A \vdash B) \\
{\multimap}\mathsf{L}\ (f, {\multimap}\mathsf{R}\ g) & \stackrel{\circ}{=} {\multimap}\mathsf{R}\ ({\multimap}\mathsf{L}\ (f,g)) & (f : - \mid \Gamma \vdash A', g : B' \mid \Delta, A \vdash B) \\
\wedge\mathsf{L}_i\ ({\multimap}\mathsf{R}\ f) & \stackrel{\circ}{=} {\multimap}\mathsf{R}\ (\wedge\mathsf{L}_i\ f) & (f : A' \mid \Gamma, A \vdash B) \\
\vee\mathsf{L}\ ({\multimap}\mathsf{R}\ f, {\multimap}\mathsf{R}\ g) & \stackrel{\circ}{=} {\multimap}\mathsf{R}\ (\vee\mathsf{L}\ (f,g)) & (f : A' \mid \Gamma, A \vdash B, g : B' \mid \Gamma, A \vdash B) \\
\vee\mathsf{R}_i\ ({\multimap}\mathsf{L}\ (f,g)) & \stackrel{\circ}{=} {\multimap}\mathsf{L}\ (f, \vee\mathsf{R}_i\ g) & (f : - \mid \Gamma \vdash A, g : B \mid \Delta \vdash A')
\end{array}$$

The sequent calculus enjoys categorical semantics in skew monoidal categories with binary products and coproducts, which moreover are endowed with a *closed structure*, i.e. a functor $\multimap : \mathbb{C}^{\mathsf{op}} \times \mathbb{C} \to \mathbb{C}$ forming an adjunction $- \otimes B \dashv B \multimap -$ for all objects B [18]. There is no need to require left-distributivity, since this can now be proved using the adjunction and the universal property of coproducts.

Notice that, in non-commutative linear logic, there exist two distinct linear implications, also called left and right residuals [13]. Our calculus includes a single implication $\multimap$. We currently do not know whether the inclusion of the second implication to our logic is a meaningful addition nor whether it corresponds to some particular categorical notion.

We now discuss the extension of the focused sequent calculus. This is more complicated than the extensions considered in Sections 5.1 and 5.2. In order to understand the increased complexity, let us include the two new rules ${\multimap}\mathsf{R}$ and ${\multimap}\mathsf{L}$ in the "naive" focused sequent calculus in (3). The right $\multimap$-rule is invertible, so it belongs to phase $\vdash_{\mathsf{RI}}$, while the left rule is not, so it goes in phase $\vdash_{\mathsf{F}}$.

$$\frac{- \mid \Gamma \vdash_{\mathsf{RI}} A \quad B \mid \Delta \vdash_{\mathsf{LI}} P}{A \multimap B \mid \Gamma, \Delta \vdash_{\mathsf{F}} P}\,{\multimap}\mathsf{L} \qquad \frac{S \mid \Gamma, A \vdash_{\mathsf{RI}} B}{S \mid \Gamma \vdash_{\mathsf{RI}} A \multimap B}\,{\multimap}\mathsf{R}$$

As we know, this calculus is too permissive, and the inclusion of the above rules increases the non-deterministic choices in proof search even further. As a strategy for taming non-determinism, as before we decide to prioritize left non-invertible rules over right non-invertible ones. So we need to think of all possible situations when a right non-invertible rule must be applied before a left non-invertible one. The presence of $\multimap$ creates two new possibilities: (i) ${\multimap}\mathsf{L}$ splits the context differently in different premises, or (ii) left non-invertible rules manipulate formulae that have been moved to the context by applications of ${\multimap}\mathsf{R}$. To understand these situations, let us look at two examples.

As an example of situation (i), consider the sequent $\mathsf{I} \multimap \mathsf{I} \mid \mathsf{I}, Y \vdash_{\mathsf{F}} (\mathsf{I} \wedge \mathsf{I}) \otimes Y$ and the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{- \mid\ \vdash_{\mathsf{F}} \mathsf{I}}\,\mathsf{IR}}{- \mid\ \vdash_{\mathsf{LI}} \mathsf{I}}\,\mathsf{F2LI}}{\mathsf{I} \mid\ \vdash_{\mathsf{LI}} \mathsf{I}}\,\mathsf{IL}}{- \mid \mathsf{I} \vdash_{\mathsf{F}} \mathsf{I}}\,\mathsf{pass}}{- \mid \mathsf{I} \vdash_{\mathsf{RI}} \mathsf{I}}\,\mathsf{sw} \quad \dfrac{\dfrac{\dfrac{}{- \mid\ \vdash_{\mathsf{F}} \mathsf{I}}\,\mathsf{IR}}{- \mid\ \vdash_{\mathsf{LI}} \mathsf{I}}\,\mathsf{F2LI}}{\mathsf{I} \mid\ \vdash_{\mathsf{LI}} \mathsf{I}}\,\mathsf{IL}}{\mathsf{I} \multimap \mathsf{I} \mid \mathsf{I} \vdash_{\mathsf{F}} \mathsf{I}}\,{\multimap}\mathsf{L}}{\mathsf{I} \multimap \mathsf{I} \mid \mathsf{I} \vdash_{\mathsf{RI}} \mathsf{I}}\,\mathsf{sw} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{- \mid\ \vdash_{\mathsf{F}} \mathsf{I}}\,\mathsf{IR}}{- \mid\ \vdash_{\mathsf{RI}} \mathsf{I}}\,\mathsf{sw} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{- \mid\ \vdash_{\mathsf{F}} \mathsf{I}}\,\mathsf{IR}}{- \mid\ \vdash_{\mathsf{LI}} \mathsf{I}}\,\mathsf{F2LI}}{\mathsf{I} \mid\ \vdash_{\mathsf{LI}} \mathsf{I}}\,\mathsf{IL}}{- \mid \mathsf{I} \vdash_{\mathsf{F}} \mathsf{I}}\,\mathsf{pass}}{- \mid \mathsf{I} \vdash_{\mathsf{LI}} \mathsf{I}}\,\mathsf{F2LI}}{\mathsf{I} \mid \mathsf{I} \vdash_{\mathsf{LI}} \mathsf{I}}\,\mathsf{IL}}{\mathsf{I} \multimap \mathsf{I} \mid \mathsf{I} \vdash_{\mathsf{F}} \mathsf{I}}\,{\multimap}\mathsf{L}}{\mathsf{I} \multimap \mathsf{I} \mid \mathsf{I} \vdash_{\mathsf{RI}} \mathsf{I}}\,\mathsf{sw}}{\mathsf{I} \multimap \mathsf{I} \mid \mathsf{I} \vdash_{\mathsf{RI}} \mathsf{I} \wedge \mathsf{I}}\,\wedge\mathsf{R} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{Y \mid\ \vdash_{\mathsf{F}} Y}\,\mathsf{ax}}{Y \mid\ \vdash_{\mathsf{LI}} Y}\,\mathsf{F2LI}}{- \mid Y \vdash_{\mathsf{F}} Y}\,\mathsf{pass}}{- \mid Y \vdash_{\mathsf{RI}} Y}\,\mathsf{sw}}{\mathsf{I} \multimap \mathsf{I} \mid \mathsf{I}, Y \vdash_{\mathsf{F}} (\mathsf{I} \wedge \mathsf{I}) \otimes Y}\,\otimes\mathsf{R} \tag{8}$$

Here $\otimes\mathsf{R}$ must be applied first, before $\multimap\mathsf{L}$. In the proofs of the two premises of $\wedge\mathsf{R}$, which prove the same sequent $\mathsf{I} \multimap \mathsf{I} \mid \mathsf{I} \vdash_{\mathsf{RI}} \mathsf{I}$, rule $\multimap\mathsf{L}$ splits the context in different ways: in the left branch the unit I in context is sent to the left premise, while in the right branch it goes to the right premise. If the application of the rule $\multimap\mathsf{L}$ would have split the context in the same way, then we could have applied $\multimap\mathsf{L}$ before $\otimes\mathsf{R}$.

For (ii), consider sequents $- \mid Y \vdash_{\mathsf{F}} (X \multimap X) \otimes Y$ and $X \multimap Y \mid Z \vdash_{\mathsf{F}} (X \multimap Y) \otimes Z$ with proofs:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{X \mid\ \vdash_{\mathsf{F}} X}\,\mathsf{ax}}{X \mid\ \vdash_{\mathsf{LI}} X}\,\mathsf{F2LI}}{- \mid X \vdash_{\mathsf{F}} X}\,\mathsf{pass}}{- \mid X \vdash_{\mathsf{RI}} X}\,\mathsf{sw}}{- \mid\ \vdash_{\mathsf{RI}} X \multimap X}\,\multimap\mathsf{R} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{Y \mid\ \vdash_{\mathsf{F}} Y}\,\mathsf{ax}}{Y \mid\ \vdash_{\mathsf{LI}} Y}\,\mathsf{F2LI}}{- \mid Y \vdash_{\mathsf{F}} Y}\,\mathsf{pass}}{- \mid Y \vdash_{\mathsf{RI}} Y}\,\mathsf{sw}}{- \mid Y \vdash_{\mathsf{F}} (X \multimap X) \otimes Y}\,\otimes\mathsf{R} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{X \mid\ \vdash_{\mathsf{F}} X}\,\mathsf{ax}}{X \mid\ \vdash_{\mathsf{LI}} X}\,\mathsf{F2LI}}{- \mid X \vdash_{\mathsf{F}} X}\,\mathsf{pass}}{- \mid X \vdash_{\mathsf{RI}} X}\,\mathsf{sw} \quad \dfrac{\dfrac{}{Y \mid\ \vdash_{\mathsf{F}} Y}\,\mathsf{ax}}{Y \mid\ \vdash_{\mathsf{LI}} Y}\,\mathsf{F2LI}}{X \multimap Y \mid X \vdash_{\mathsf{F}} Y}\,\multimap\mathsf{L}}{X \multimap Y \mid X \vdash_{\mathsf{RI}} Y}\,\mathsf{sw}}{X \multimap Y \mid\ \vdash_{\mathsf{RI}} X \multimap Y}\,\multimap\mathsf{R} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{Z \mid\ \vdash_{\mathsf{F}} Z}\,\mathsf{ax}}{Z \mid\ \vdash_{\mathsf{LI}} Z}\,\mathsf{F2LI}}{- \mid Z \vdash_{\mathsf{F}} Z}\,\mathsf{pass}}{- \mid Z \vdash_{\mathsf{RI}} Z}\,\mathsf{sw}}{X \multimap Y \mid Z \vdash_{\mathsf{F}} (X \multimap Y) \otimes Z}\,\otimes\mathsf{R} \tag{9}$$

In the first derivation, rule pass in the left branch of $\otimes\mathsf{R}$ cannot be moved to the bottom of the proof tree, since formula X is not yet in context, it becomes available only after the application of $\multimap\mathsf{R}$. Analogously, in the second derivation, rule $\multimap\mathsf{L}$ in the left branch of $\otimes\mathsf{R}$ cannot be moved at the bottom, since the formula X that it sends to the left premise appears in context only after the application of $\multimap\mathsf{R}$.

This motivates the addition of two new tags, corresponding to the two situations previously discussed: on top of $\mathbb{P}, \mathbb{C}_1, \mathbb{C}_2$ and $\mathbb{R}$, a tag could either be of the form Γ, for each context Γ, or of the form $\bullet$. The validity condition for list of tags needs to be updated. A list of tags l is now valid if it is non-empty and either (i) $\mathbb{R} \in l$, (ii) both $\mathbb{C}_1 \in l$ and $\mathbb{C}_2 \in l$, (iii) there exist contexts Γ, Γ' such that $\Gamma \in l$, $\Gamma' \in l$ and $\Gamma \neq \Gamma'$, or (iv) $\bullet \in l$. Following [20], on top of tag annotations for sequents, we also require tag annotations for formulae in context. There is only one tag $\bullet$ for formulae. The tag on the formula $A^{\bullet}$ means that A has been previously moved to the context by an application of $\multimap\mathsf{R}$ in phase $\vdash^{l}_{\mathsf{RI}}$.

Here are the inference rules of the focused sequent calculus with linear implication:

$$\text{(right invertible)} \quad \dfrac{S \mid \Gamma \vdash^{l_1?}_{\mathsf{RI}} A \quad S \mid \Gamma \vdash^{l_2?}_{\mathsf{RI}} B}{S \mid \Gamma \vdash^{l_1?,l_2?}_{\mathsf{RI}} A \wedge B}\,\wedge\mathsf{R} \quad \dfrac{S \mid \Gamma, A^{\bullet?} \vdash^{l?}_{\mathsf{RI}} B}{S \mid \Gamma \vdash^{l?}_{\mathsf{RI}} A \multimap B}\,\multimap\mathsf{R} \quad \dfrac{S \mid \Gamma \vdash^{t?}_{\mathsf{LI}} P}{S \mid \Gamma \vdash^{t?}_{\mathsf{RI}} P}\,\mathsf{LI2RI}$$

$$\text{(left invertible)} \quad \begin{array}{c} \dfrac{- \mid \Gamma \vdash_{\mathsf{LI}} P}{\mathsf{I} \mid \Gamma \vdash_{\mathsf{LI}} P}\,\mathsf{IL} \qquad \dfrac{A \mid B, \Gamma \vdash_{\mathsf{LI}} P}{A \otimes B \mid \Gamma \vdash_{\mathsf{LI}} P}\,\otimes\mathsf{L} \\ \dfrac{A \mid \Gamma \vdash_{\mathsf{LI}} P \quad B \mid \Gamma \vdash_{\mathsf{LI}} P}{A \vee B \mid \Gamma \vdash_{\mathsf{LI}} P}\,\vee\mathsf{L} \qquad \dfrac{T \mid \Gamma \vdash^{t?}_{\mathsf{F}} P}{T \mid \Gamma \vdash^{t?}_{\mathsf{LI}} P}\,\mathsf{F2LI} \end{array}$$

$$\text{(focusing)} \quad \dfrac{A \mid \Gamma^{\circ} \vdash_{\mathsf{LI}} P \quad \text{if } A^{\bullet?} = A \text{ then } (t \text{ does not exist or } t = \mathbb{P}) \text{ else } t = \bullet}{- \mid A^{\bullet?}, \Gamma \vdash^{t?}_{\mathsf{F}} P}\,\mathsf{pass} \tag{10}$$

$$\dfrac{}{X \mid\ \vdash^{\mathbb{R}?}_{\mathsf{F}} X}\,\mathsf{ax} \qquad \dfrac{}{- \mid\ \vdash^{\mathbb{R}?}_{\mathsf{F}} \mathsf{I}}\,\mathsf{IR} \qquad \dfrac{A \mid \Gamma^{\circ} \vdash_{\mathsf{LI}} P}{A \wedge B \mid \Gamma \vdash^{\mathbb{C}_1?}_{\mathsf{F}} P}\,\wedge\mathsf{L}_1 \qquad \dfrac{B \mid \Gamma^{\circ} \vdash_{\mathsf{LI}} P}{A \wedge B \mid \Gamma \vdash^{\mathbb{C}_2?}_{\mathsf{F}} P}\,\wedge\mathsf{L}_2$$

$$\dfrac{T \mid \Gamma^{\circ} \vdash^{l}_{\mathsf{RI}} A \quad - \mid \Delta^{\circ} \vdash_{\mathsf{RI}} B \quad l \text{ valid}}{T \mid \Gamma, \Delta \vdash^{\mathbb{R}?}_{\mathsf{F}} A \otimes B}\,\otimes\mathsf{R} \quad \dfrac{T \mid \Gamma^{\circ} \vdash^{l}_{\mathsf{RI}} A \quad l \text{ valid}}{T \mid \Gamma \vdash^{\mathbb{R}?}_{\mathsf{F}} A \vee B}\,\vee\mathsf{R}_1 \quad \dfrac{T \mid \Gamma^{\circ} \vdash^{l}_{\mathsf{RI}} B \quad l \text{ valid}}{T \mid \Gamma \vdash^{\mathbb{R}?}_{\mathsf{F}} A \vee B}\,\vee\mathsf{R}_2$$

$$\dfrac{- \mid \Gamma, \Delta^{\circ} \vdash_{\mathsf{RI}} A \quad B \mid \Lambda^{\circ} \vdash_{\mathsf{LI}} P \quad \text{if } \Delta^{\bullet} \text{ is empty then } (t \text{ does not exist or } t = \Gamma) \text{ else } t = \bullet}{A \multimap B \mid \Gamma, \Delta^{\bullet}, \Lambda \vdash^{t?}_{\mathsf{F}} P}\,\multimap\mathsf{L}$$

Again P indicates a non-negative formula, which now means that its principal connective is neither $\wedge$ nor $\multimap$. The notation $\Gamma^{\bullet}$ means that all the formulae in Γ are tagged, while Γ° indicates that all the tags on formulae in Γ have been erased. We write $A^{\bullet ?}$ to denote A if the formula appears in an untagged sequent and $A^{\bullet}$ if it appears in a sequent marked with a list of tags l or a single tag t.

Tags of the form $t = \Gamma$ are used to record different splitting of context in applications of ${\multimap}\mathsf{L}$, while tag $t = \bullet$ marks when rule ${\multimap}\mathsf{L}$ sends tagged formulae to the left premise and when rule pass moves a tagged formula to the stoup.

Rule ${\multimap}\mathsf{R}$ moves a formula A from the succedent to the right end of the context. If its conclusion is marked by a list of tags l, then A is also tagged with $\bullet$.

The side condition in rule ${\multimap}\mathsf{L}$ should be read as follows. The tagged context $\Delta^{\bullet}$ starts with the leftmost tagged formula in the sequent. If $\Delta^{\bullet}$ is empty, then the sequent is either untagged (so there is no t) or the tag t is equal to Γ. If $\Delta^{\bullet}$ is non-empty, then $t = \bullet$. In particular, $\Delta^{\bullet}$ contains at least one tagged formula, which must have appeared in context from an application of ${\multimap}\mathsf{R}$. If $\Delta^{\bullet}$ is empty and $t = \Gamma$, no new (meaning: tagged with $\bullet$) formula is moved to the left premise. If $t = \Gamma$ then we are performing proof search inside the premise of a right non-invertible rule and t belongs to some valid list of tags l. List l could be valid because of a different branch in the proof tree where ${\multimap}\mathsf{L}$ is also applied but the context has been split differently (so its tag would be Γ' for some $\Gamma \neq \Gamma'$).

Rule pass has a similar side condition to ${\multimap}\mathsf{L}$. If A does not have a tag, then the sequent is also untagged or the tag t is equal to $\mathbb{P}$. If A has tag $\bullet$, then t must also be $\bullet$. In other words, if $t = \bullet$ then the formula that pass moves to the stoup must also be tagged with $\bullet$, i.e. must have been added to the context by an application of ${\multimap}\mathsf{R}$.

We can reconstruct the derivation in (8) within the focused sequent calculus with tags in (10).

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{-\mid\ \vdash_{\mathsf{F}}\mathsf{I}}\,\mathsf{IR}}{-\mid\ \vdash_{\mathsf{RI}}\mathsf{I}}\,\mathsf{sw} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{-\mid\ \vdash_{\mathsf{F}}\mathsf{I}}\,\mathsf{IR}}{-\mid\ \vdash_{\mathsf{LI}}\mathsf{I}}\,\mathsf{F2LI}}{\mathsf{I}\mid\ \vdash_{\mathsf{LI}}\mathsf{I}}\,\mathsf{IL}}{-\mid\mathsf{I}\vdash_{\mathsf{F}}\mathsf{I}}\,\mathsf{pass}}{-\mid\mathsf{I}\vdash_{\mathsf{LI}}\mathsf{I}}\,\mathsf{F2LI}}{\mathsf{I}\mid\mathsf{I}\vdash_{\mathsf{LI}}\mathsf{I}}\,\mathsf{IL}}{\mathsf{I}\multimap\mathsf{I}\mid\mathsf{I}\vdash^{[\,]}_{\mathsf{F}}\mathsf{I}}\,{\multimap}\mathsf{L}}{\mathsf{I}\multimap\mathsf{I}\mid\mathsf{I}\vdash^{[\,]}_{\mathsf{RI}}\mathsf{I}}\,\mathsf{sw} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{-\mid\ \vdash_{\mathsf{F}}\mathsf{I}}\,\mathsf{IR}}{-\mid\ \vdash_{\mathsf{LI}}\mathsf{I}}\,\mathsf{F2LI}}{\mathsf{I}\mid\ \vdash_{\mathsf{LI}}\mathsf{I}}\,\mathsf{IL}}{-\mid\mathsf{I}\vdash_{\mathsf{F}}\mathsf{I}}\,\mathsf{pass}}{-\mid\mathsf{I}\vdash_{\mathsf{RI}}\mathsf{I}}\,\mathsf{sw} \quad \dfrac{\dfrac{\dfrac{}{-\mid\ \vdash_{\mathsf{F}}\mathsf{I}}\,\mathsf{IR}}{-\mid\ \vdash_{\mathsf{LI}}\mathsf{I}}\,\mathsf{F2LI}}{\mathsf{I}\mid\ \vdash_{\mathsf{LI}}\mathsf{I}}\,\mathsf{IL}}{\mathsf{I}\multimap\mathsf{I}\mid\mathsf{I}\vdash^{[\mathsf{I}]}_{\mathsf{F}}\mathsf{I}}\,{\multimap}\mathsf{L}}{\mathsf{I}\multimap\mathsf{I}\mid\mathsf{I}\vdash^{[\mathsf{I}]}_{\mathsf{RI}}\mathsf{I}}\,\mathsf{sw}}{\mathsf{I}\multimap\mathsf{I}\mid\mathsf{I}\vdash^{[\,],[\mathsf{I}]}_{\mathsf{RI}}\mathsf{I}\wedge\mathsf{I}}\,\wedge\mathsf{R} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{Y\mid\ \vdash_{\mathsf{F}}Y}\,\mathsf{ax}}{Y\mid\ \vdash_{\mathsf{LI}}Y}\,\mathsf{F2LI}}{-\mid Y\vdash_{\mathsf{F}}Y}\,\mathsf{pass}}{-\mid Y\vdash_{\mathsf{RI}}Y}\,\mathsf{sw}}{\mathsf{I}\multimap\mathsf{I}\mid\mathsf{I},Y\vdash_{\mathsf{F}}(\mathsf{I}\wedge\mathsf{I})\otimes Y}\,\otimes\mathsf{R}}{\mathsf{I}\multimap\mathsf{I}\mid\mathsf{I},Y\vdash_{\mathsf{RI}}(\mathsf{I}\wedge\mathsf{I})\otimes Y}\,\mathsf{sw}
$$

The proofs with tags of the derivations in (9) are analogous to the ones described in [20].

Proving completeness of the extended focused sequent calculus is more involved than in the absence of implication. Concretely, the complication resides in stating and proving the analog of Proposition 4.4. First, define an operation $\mathsf{impconj}(A)$ which produces a list of pairs of lists of formulae and formulae as follows:

$$
\begin{array}{lll}
\mathsf{impconj}(A) & = \mathsf{impconj}(A'), \mathsf{impconj}(B') & \text{when } A = A' \wedge B' \\
\mathsf{impconj}(A) & = ((A',\Gamma'_1),B'_1),\ldots,((A',\Gamma'_n),B'_n) & \text{when } A = A' \multimap B' \text{ and} \\
 & & \mathsf{impconj}(B') = ([(\Gamma'_1,B'_1),\ldots,(\Gamma'_n,B'_n)]) \\
\mathsf{impconj}(A) & = ([\,],A) & \text{otherwise}
\end{array}
$$

For example, $\mathsf{impconj}(A \multimap (B \multimap (X \wedge (C \vee D) \wedge (Y \multimap Z)))) = [([A,B],X),([A,B],C\vee D),([A,B,Y],Z)]$.

The statement of Proposition 4.4 for the focused sequent calculus in (10) then becomes:

Proposition 5.1. *The following rules*

$$\frac{\overset{fs}{[S \mid \Gamma, \Gamma'_i \vdash_{\mathsf{LI}} P_i]_{i \in [1,\dots,n]}}}{S \mid \Gamma \vdash_{\mathsf{LI}} A \vee B} \vee\mathsf{R}_1^{\mathsf{LI}} \qquad \frac{\overset{fs}{[S \mid \Gamma, \Gamma''_i \vdash_{\mathsf{LI}} Q_i]_{i \in [1,\dots,m]}}}{S \mid \Gamma \vdash_{\mathsf{LI}} A \vee B} \vee\mathsf{R}_2^{\mathsf{LI}}$$

$$\frac{\overset{fs}{[S \mid \Gamma, \Gamma'_i \vdash_{\mathsf{LI}} P_i]_{i \in [1,\dots,n]}} \qquad - \mid \Delta \vdash_{\mathsf{RI}} B'}{S \mid \Gamma, \Delta \vdash_{\mathsf{LI}} A \otimes B'} \otimes\mathsf{R}^{\mathsf{LI}}$$

are admissible, where $\mathsf{impconj}(A) = [(\Gamma'_1, P_1), \dots, (\Gamma'_n, P_n)]$ *and* $\mathsf{impconj}(B) = [(\Gamma''_1, Q_1), \dots, (\Gamma''_m, Q_m)]$.

6 Conclusion

The paper presents a sequent calculus for a semi-associative and semi-unital logic, extending the system introduced in [23] with additive conjunction and disjunction. Categorical models of this calculus are skew monoidal categories with binary products and coproducts, and the tensor product preserves coproducts on the left: $(A+B) \otimes C \cong (A \otimes C) + (B \otimes C)$. Derivations in the sequent calculus are equated by a congruence relation $\overset{\circ}{=}$ and canonical representatives of each $\overset{\circ}{=}$-equivalence class can be computed in a separate sequent calculus of normal forms, that we dubbed "focused" due to its phase separation similar to the one in Andreoli's technique [3]. It should be remarked that, differently from Andreoli's focusing, and also the maximally multi-focused sequent calculus for skew monoidal closed categories by one of the authors [25], we do not insist on keeping the focus during the synchronous phase of proof search, and we always privilege the application of left non-invertible rules over right non-invertible ones. In order to achieve completeness wrt. the sequent calculus, the focused system employs a system of tag annotations providing explicit justifications for cases where right non-invertible rules must be applied before the left non-invertible ones.

The focused sequent calculus is a concrete presentation of the free distributive skew monoidal category on the set of atomic formulae. Therefore the normalization/focusing algorithm determines a procedure for solving the coherence problem of distributive skew monoidal categories.

In the final part of the paper, we have looked at extensions of the logic with additive units, a skew exchange rule in the style of Bourke and Lack [6], and linear implication. This section still needs to be formalized in Agda, which will be our forthcoming step.

This paper takes one step further in a large project aiming at modularly analyzing proof systems with categorical models given by categories with skew structure [26, 23, 22, 21, 24, 20]. We are interested in looking for applications of these systems to combinatorics and linguistics, following the initial investigation by Zeilberger [26] and Moortgat [15].

We are interested in using the techniques introduced in this paper to design a calculus of normal forms for the classical linear logic MALL. In the latter setting, the situation is more complicated than the skew one, since there could be more than two formulae that can be under focus in the synchronous phase. We expect our calculus to give an alternative presentation of the maximally multi-focused proofs of Chaudhuri et al. [8].

Acknowledgements This work was supported by the Estonian Research Council grant PSG749 and the ESF funded Estonian IT Academy research measure (project 2014-2020.4.05.19-0001).

References

[1] Vito Michele Abrusci (1990): *Non-commutative Intuitionistic Linear Logic*. Mathematical Logic Quarterly 36(4), pp. 297–318, doi:10.1002/malq.19900360405.

[2] Thorsten Altenkirch, James Chapman & Tarmo Uustalu (2015): *Monads Need Not Be Endofunctors*. Logical Methods in Computer Science 11(1):3, doi:10.2168/lmcs-11(1:3)2015.

[3] Jean-Marc Andreoli (1992): *Logic Programming with Focusing Proofs in Linear Logic*. Journal of Logic and Computation 2(3), pp. 297–347, doi:10.1093/logcom/2.3.297.

[4] John Bourke (2017): *Skew structures in 2-category theory and homotopy theory*. Journal of Homotopy and Related Structures 12(1), pp. 31–81, doi:10.1007/s40062-015-0121-z.

[5] John Bourke & Stephen Lack (2018): *Skew Monoidal Categories and Skew Multicategories*. Journal of Algebra 506, pp. 237–266, doi:10.1016/j.jalgebra.2018.02.039.

[6] John Bourke & Stephen Lack (2020): *Braided Skew Monoidal Categories*. Theory and Applications of Categories 35(2), pp. 19–63. Available at `http://www.tac.mta.ca/tac/volumes/35/2/35-02abs.html`.

[7] Michael Buckley, Richard Garner, Stephen Lack & Ross Street (2015): *The Catalan Simplicial Set*. Mathematical Proceedings of Cambridge Philosophical Society 158(2), pp. 211–222, doi:10.1017/s0305004114000498.

[8] Kaustuv Chaudhuri, Dale Miller & Alexis Saurin (2008): *Canonical Sequent Proofs via Multi-Focusing*. In Giorgio Ausiello, Juhani Karhumäki, Giancarlo Mauri & Luke Ong, editors: Proceedings of 5th IFIP International Conference on Theoretical Computer Science, TCS 2008, International Federation of Information Processing Series 273, Springer, pp. 383–396, doi:10.1007/978-0-387-09680-3_26.

[9] Jean-Yves Girard (1987): *Linear Logic*. Theoretical Computer Science 50, pp. 1–102, doi:10.1016/0304-3975(87)90045-4.

[10] Jean-Yves Girard (1991): *A New Constructive Logic: Classical Logic*. Mathematical Structures in Computer Science 1(3), pp. 255–296, doi:10.1017/s0960129500001328.

[11] Stephen Lack & Ross Street (2012): *Skew Monoidales, Skew Warpings and Quantum Categories*. Theory and Applications of Categories 26, pp. 385–402. Available at `http://www.tac.mta.ca/tac/volumes/26/15/26-15abs.html`.

[12] Stephen Lack & Ross Street (2014): *Triangulations, Orientals, and Skew Monoidal Categories*. Advances in Mathematics 258, pp. 351–396, doi:10.1016/j.aim.2014.03.003.

[13] Joachim Lambek (1958): *The Mathematics of Sentence Structure*. American Mathematical Monthly 65(3), pp. 154–170, doi:10.2307/2310058.

[14] Saunders Mac Lane (1963): *Natural Associativity and Commutativity*. Rice University Studies 49(4), pp. 28–46. Available at `http://hdl.handle.net/1911/62865`.

[15] Michael Moortgat (2020): *The Tamari order for D^3 and derivability in semi-associative Lambek-Grishin Calculus*. Talk at 16th Workshop on Computational Logic and Applications, CLA 2020. Slides available at: `http://cla.tcs.uj.edu.pl/history/2020/pdfs/CLA_slides_Moortgat.pdf`.

[16] Richard Moot & Christian Retoré (2012): *The Logic of Categorial Grammars - A Deductive Account of Natural Language Syntax and Semantics*. Lecture Notes in Computer Science 6850, Springer, doi:10.1007/978-3-642-31555-8.

[17] Gabriel Scherer & Ddier Rémy (2015): *Which Simple Types Have a Unique Inhabitant?* In: Proceedings of 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, ACM, pp. 243–255, doi:10.1145/2784731.2784757.

[18] Ross Street (2013): *Skew-Closed Categories*. Journal of Pure and Applied Algebra 217(6), pp. 973–988, doi:10.1016/j.jpaa.2012.09.020.

[19] Kornél Szlachányi (2012): *Skew-Monoidal Categories and Bialgebroids*. Advances in Mathematics 231(3–4), pp. 1694–1730, doi:10.1016/j.aim.2012.06.027.

[20] Tarmo Uustalu, Niccolò Veltri & Cheng-Syuan Wan (2022): *Proof Theory of Skew Non-Commutative MILL*. In Andrzej Indrzejczak & Michal Zawidzki, editors: *Proceedings of 10th International Conference on Non-classical Logics: Theory and Applications, NCL 2022*, *Electronic Proceedings in Theoretical Computer Science* 358, Open Publishing Association, pp. 118–135, doi:10.4204/eptcs.358.9.

[21] Tarmo Uustalu, Niccoló Veltri & Noam Zeilberger (2021): *Deductive Systems and Coherence for Skew Prounital Closed Categories*. In Claudio Sacerdoti Coen & Alwen Tiu, editors: *Proceedings of 15th Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2020*, *Electronic Proceedings in Theoretical Computer Science* 332, Open Publishing Association, pp. 35–53, doi:10.4204/eptcs.332.3.

[22] Tarmo Uustalu, Niccolò Veltri & Noam Zeilberger (2021): *Proof Theory of Partially Normal Skew Monoidal Categories*. In David I. Spivak & Jamie Vicary, editors: *Proceedings of 3rd Annual International Applied Category Theory Conference 2020, ACT 2020*, *Electronic Proceedings in Theoretical Computer Science* 333, Open Publishing Association, pp. 230–246, doi:10.4204/eptcs.333.16.

[23] Tarmo Uustalu, Niccolò Veltri & Noam Zeilberger (2021): *The Sequent Calculus of Skew Monoidal Categories*. In Claudio Casadio & Philip J. Scott, editors: *Joachim Lambek: The Interplay of Mathematics, Logic, and Linguistics*, *Outstanding Contributions to Logic* 20, Springer, pp. 377–406, doi:10.1007/978-3-030-66545-6_11.

[24] Niccolò Veltri (2021): *Coherence via Focusing for Symmetric Skew Monoidal Categories*. In Alexandra Silva, Renata Wassermann & Ruy de Queiroz, editors: *Proceedings of 27th International Workshop on Logic, Language, Information, and Computation, WoLLIC 2021*, *Lecture Notes in Computer Science* 13028, Springer, pp. 184–200, doi:10.1007/978-3-030-88853-4_12.

[25] Niccolò Veltri (2023): *Maximally Multi-focused Proofs for Skew Non-Commutative MILL*. In Helle Hvid Hansen, Andre Scedrov & Ruy J. G. B. de Queiroz, editors: *Proceedings of 29th International Workshop on Logic, Language, Information, and Computation, WoLLIC 2023*, *Lecture Notes in Computer Science* 13923, Springer, pp. 377–393, doi:10.1007/978-3-031-39784-4_24.

[26] Noam Zeilberger (2019): *A Sequent Calculus for a Semi-Associative Law*. *Logical Methods in Computer Science* 15(1):9, doi:10.23638/lmcs-15(1:9)2019.

Data Abstraction, Arrays, Maps, and Completeness, aka "Cell Morphing"

David Monniaux
Univ. Grenoble Alpes, CNRS, Grenoble INP[*], VERIMAG, 38000 Grenoble, France
David.Monniaux@univ-grenoble-alpes.fr

(Invited talk abstract. This discusses joint work with Laure Gonnord and Julien Braine.)

Arrays and array-like data structures (such as hash-tables) are widely used in software. Furthermore, many semantic aspects of programming, such as byte-addressed memory, data structure fields, etc., can be modeled as arrays. Static analyzers that aim at proving properties of programs thus often need to deal with arrays. An array, in our context, is a map a from an index set I (not necessarily an integer range) to a value set V, with "get" $a[i]$ and "set" $a[i \leftarrow v]$ operations satisfying the relations:

$$a[i \leftarrow v][i] = v$$

$$a[i \leftarrow v][i'] = t[i'] \text{ when } i' \neq i$$

Many interesting invariants about arrays are universally quantified over indices. For instance, "the array a contains 42 at all indices from 0 included to n excluded" is written $\forall k,\ 0 \leq k < n \Rightarrow a[k] = 42$"; "the array a is sorted from indices from 0 included to n excluded" can be written $\forall k_1 k_2,\ 0 \leq k_1 \leq k_2 < n \Rightarrow a[k_1] \leq a[k_2]$. More generally, we consider invariants of the form $\forall k_1, \dots, k_m P(k_1, \dots, k_m, a[k_1], \dots, a[k_m], v_1, \dots, v_{|V|})$ where a is an array and the v_i are other variables of the program, and P is an arbitrary predicate. We also consider the case of multiple arrays, so as to be able to express properties such as $\forall k,\ 0 \leq k < n \Rightarrow a[k] = b[k]$.

Numerous approaches have been proposed to automatically infer such invariants. We have proposed an approach [BGM21, BGM16, Bra22, MG16], that converts an invariant inference problem (expressed as a solution to a system of Horn clauses), constrained by a safety property and restricted to such universally quantified invariants into an invariant inference problem on ordinary invariants (without universal quantification), through suitable *quantifier instantiation*. By further processing, through Ackermannization, these problems can even be converted, without loss of precision, to invariant inference problems over purely scalar variables (no more arrays).

While that second step does not incur loss of precision, the first step (quantifier instantiation), is in general sound (if the transformed invariant problem has a solution, then so has the original problem) but *incomplete*: it may be the case that it converts an invariant inference problem (a system of Horn clauses) that has a solution among universally quantified array invariants into a problem that has no solution. We however show that under certain syntactic restrictions (mostly, that the original Horn clause problem should be *linear*, which includes all problems obtained from the inductiveness conditions of control-flow graphs), that transformation is complete.

We thus show that under these conditions, our approach for solving invariant inference problems within universally quantified array properties is thus relatively complete to our ability to solve the resulting array-free ordinary invariant inference problem, in general over non-linear Horn clauses. This is the main limitation to our approach, since solvers for such kinds of problems tend to have unpredictable performance.

[*]Institute of Engineering Univ. Grenoble Alpes

References

[BGM16] Julien Braine, Laure Gonnord & David Monniaux (2016): *Verifying Programs with Arrays and Lists*. Internship report, ENS Lyon. Available at `https://hal.archives-ouvertes.fr/hal-01337140`.

[BGM21] Julien Braine, Laure Gonnord & David Monniaux (2021): *Data Abstraction: A General Framework to Handle Program Verification of Data Structures*. In: *Static analysis (SAS)*, *Lecture Notes in Computer Science* 12913, Springer, pp. 215–235, doi:`10.1007/978-3-030-88806-0_11`. Available at `https://inria.hal.science/hal-03321868`.

[Bra22] Julien Braine (2022): *The Data-abstraction Framework: abstracting unbounded data-structures in Horn clauses, the case of arrays. (La Méthode Data-abstraction: une technique d'abstraction de structures de données non-bornées dans des clauses de Horn, le cas des tableaux)*. Ph.D. thesis, University of Lyon, France. Available at `https://tel.archives-ouvertes.fr/tel-03771839`.

[MG16] David Monniaux & Laure Gonnord (2016): *Cell Morphing: From Array Programs to Array-Free Horn Clauses*. In: *Static analysis*, *Lecture Notes in Computer Science* 9837, Springer Verlag, pp. 361–382, doi:`10.1007/978-3-662-53413-7_18`. arXiv:`1509.09092`.

CHC-COMP 2023: Competition Report

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

Hari Govind V K
University of Waterloo, Canada
hgvk94@gmail.com

CHC-COMP 2023 is the sixth edition of the Competition of Solvers for Constrained Horn Clauses. The competition was run in April 2023 and the results were presented at the 10th Workshop on Horn Clauses for Verification and Synthesis held in Paris, France, on April 23, 2023. This edition featured seven solvers (six competing and one hors concours) and six tracks, each of which dealing with a class of clauses. This report describes the organization of CHC-COMP 2023 and presents its results.

1 Introduction

Constrained Horn Clauses (CHCs) are a class of first-order logic formulas where the Horn clause format is extended with *constraints*, that is, formulas of an arbitrary, possibly non-Horn, background theory (such as linear integer arithmetic, arrays, and algebraic data types).

CHCs have gained popularity as a formalism well suited for automatic program verification [20, 5, 9]). Indeed, the last decade has seen impressive progress in the development of solvers for CHCs (CHC solvers), which can now be effectively used as back-end tools for program verification due to their ability to solve satisfiability problems dealing with a variety of background theories. A non-exhaustive list of solvers includes: ADTInd [40], ADTRem [11], Eldarica [25], FreqHorn [16], Golem [7], HSF [20], PCSat [39], RAHFT [27], RInGen [29], SPACER [28], Ultimate TreeAutomizer [13], and VeriMAP [10].

CHC-COMP is an annual competition that aims to evaluate state-of-the-art CHC solvers on realistic and publicly available benchmarks; it is open to proposals and contributions from users and developers of CHC solvers, as well as researchers working in the field of CHC solving foundations and its applications.

CHC-COMP 2023[1] is the 6th edition of the CHC-COMP, affiliated with the 10th Workshop on Horn Clauses for Verification and Synthesis (HCVS 2023[2]) held in Paris, France, on April 23, 2023. The deadline for submitting candidate benchmarks was March 24, 2023. The deadlines for submitting tools for the test (optional) and the competition runs were 31 March and 7 April 2023, respectively. The competition was run in the subsequent two weeks, and the results were announced at HCVS 2023. CHC-COMP 2023 featured 7 solvers (6 competing solvers and one hors concours), and 6 tracks, each of which dealing with a class of clauses consisting of linear and nonlinear CHCs with constraints over linear integer arithmetic, arrays, non-recursive/recursive algebraic data types, and a few combinations thereof.

This report is structured as follows. Section 2 presents the competition tracks, the technical resources used to run the competition, and the evaluation model adopted to rank the solvers. Section 3 presents the inventory of benchmarks and how the candidate benchmarks have been processed and selected for the competition runs. Sections 4 and 5 present the tools submitted to CHC-COMP 2023 and the results of the competition, respectively. Section 6 presents some closing remarks from the organizers and participants of CHC-COMP 2023. Section 7 collects the tool descriptions contributed by the participants. Finally, Appendix A includes the tables with the detailed results about the competition runs.

*The author is member of the INdAM Research Group GNCS.
[1]https://chc-comp.github.io/
[2]https://www.sci.unich.it/hcvs23/

Acknowledgements

We would also like to thank the HCVS 2023 Program Chairs, David Monniaux and Jose F. Morales, for hosting the competition this year as well, and all the HCVS attendees for the fruitful discussion we had after the presentation of the CHC-COMP report. A special thanks goes to Hossein Hojjat for presenting CHC-COMP 2023 at TOOLympics 2023[3].

CHC-COMP 2023 heavily built on the infrastructure developed by the organizers of the previous editions, that is, Grigory Fedyukovich, Arie Gurfinkel, and Philipp Rümmer, which also includes the contributions from Nikolaj Bjørner, Adrien Champion, and Dejan Jovanovic.

We are also extremely grateful to StarExec[4] [38] that continues to support the CHC community by providing the CHC-COMP the computing resources to run the competition. In particular, we would like to thank Aaron Stump for helping us in accessing and using the StarExec services.

2 Design and Organization

This section presents (i) the competition tracks, (ii) the technical resources used to run the solvers, (iii) the characteristics of the test and the competition runs, and (iv) the evaluation model used to rank the solvers in each track.

2.1 Tracks

CHC-COMP is organized in tracks, each of which deals with a class of CHCs. CHCs are classified according to the following features: (i) the background theory of the constraints, and (ii) the number of *uninterpreted atoms* (that is, atoms whose predicate symbols do not belong to the background theory) occurring in the premises of clauses. A clause with at most one uninterpreted atom in the premise is said to be *linear*, and *nonlinear* otherwise.

Solvers participating in the CHC-COMP 2023 could enter the competition in six tracks (one track was introduced in this edition, that is, ADT-LIA-nonlin, while the remaining tracks were inherited from the previous edition)[5]:

1. **LIA-lin**: Linear Integer Arithmetic - linear clauses
2. **LIA-nonlin**: Linear Integer Arithmetic - nonlinear clauses
3. **LIA-lin-Arrays**: Linear Integer Arithmetic & Arrays - linear clauses
4. **LIA-nonlin-Arrays**: Linear Integer Arithmetic & Arrays - nonlinear clauses
5. **LIA-nonlin-Arrays-nonrecADT**: Linear Integer Arithmetic & Arrays & nonrecursive Algebraic Data Types - nonlinear clauses
6. **ADT-LIA-nonlin**: Algebraic Data Types & Linear Integer Arithmetic - nonlinear clauses

[3]`https://tacas.info/toolympics2023.php`

[4]`https://www.starexec.org/`

[5]No solver requiring the syntactic restriction on the form of the clauses included in the LRA-TS track has been submitted in last two editions. Hence, as proposed in [15, 12], the LRA-TS and LRA-TS-par tracks have been discontinued. Similarly, by considering recent advances in solving techniques for CHCs including algebraic data types, the syntactic restriction on the constraints of the CHCs in the ADT-nonlin track, which requires to have all theory symbols encoded as ADTs (called "pure ADT" problems in [15]), was no longer needed. Hence, the track has been discontinued and replaced with a more general track combining LIA and ADTs (that is, ADT-LIA-nonlin).

In addition to the theories occurring in the above list (Linear Integer Arithmetic, Arrays, nonrecursive/recursive Algebraic Data Types, and combinations thereof), benchmarks in all tracks can also make use of the Bool theory.

Finally, in LIA constraints we allow the syntactic appearance of the function symbols $*$, *div*, *mod*, and *abs*. If these operations do appear, the benchmark is included/excluded from the set of LIA benchmarks according to the following rules: (i) if the second argument of any *div* and *mod* operation is not a constant term, the benchmark is excluded; (ii) if there is more than one non-constant term in any $*$ operation, the benchmark is excluded; (iii) otherwise, the operations are considered semantically linear and the benchmark is included.

2.2 Technical Resources

CHC-COMP 2023 was run, as well as in the previous editions, on the StarExec platform, but using different technical resources[12]. StarExec made available to the CHC community a queue, called `chc-seq.q`, consisting of 20 brand new nodes equipped with Intel(R) Xeon(R) Gold 6334 CPUs. The detailed specification of the machine is available on the StarExec webpage[6].

2.3 Test and Competition Runs

CHC solvers are evaluated by performing a *test* run and a *competition* run on the StarExec platform. A run involves submitting jobs to StarExec, that is, collections of ⟨solver-configuration, benchmark⟩ pairs.

The *test* run is used by the participants to get acquainted with the StarExec platform and test out their pre-submissions. Submitting a solver for *test* runs is optional. During this test phase, the organizers contact the participants if they find any issues with their submission so that the participants can fix it before their final submission. The participants are given a week in between the *test* and *competition* runs. In the *test* runs, a small set of randomly selected benchmarks is used, and each job is limited to 600s CPU time, 600s wall-clock time, and 64GB memory.

In *competition* runs, the final submissions of the solvers are evaluated to determine the outcome of the competition, that is, to rank the solvers that entered the competition. In these runs each job is limited to 1800s CPU time, 1800s wall-clock time, and 64GB memory.

Sometimes, the competition benchmarks expose soundness bugs in solvers. We catch these bugs if two solvers disagree on the satisfiability of a benchmark. At CHC-COMP, we keep things friendly by informing the participants about the inconsistency and giving them the benchmark to reproduce the issue. If we have time, we even give them a chance to fix the issue and resubmit their tool. If not, we disqualify the tool from the track.

The data gathered from the 'job information' CSV files produced by StarExec in the competition runs are used to rank the solvers. All 'job information' CSV files of the CHC-COMP 2023 runs are available on the StarExec space `CHC/CHC-COMP/CHC-COMP-23`[7].

2.4 Evaluation of the Competition Runs

The competing solvers were evaluated using the same approach as the 2022 edition [12].

[6] `https://www.starexec.org/starexec/public/machine-specs.txt`

[7] `https://www.starexec.org/starexec/secure/explore/spaces.jsp?id=538944`

The evaluation of the competition runs were done using the `summarize.py` script available at `https://github.com/chc-comp/scripts`; the script takes as input the 'job information' CSV file produced by StarExec at job completion, and produces a ranking of the solvers.

The ranking of solvers in each track is based on the score obtained by the solvers in the competition run for a track. The score is computed on the basis of the results provided by the solver on the benchmarks for that track. The result can be *sat*, *unsat*, or *unknown* (which includes solvers giving up, running out of resources, or crashing), and the score given by the number of *sat* or *unsat* results. In the case of ex-aequo, the ranking is determined by using the CPU time, which is the total CPU time needed by a solver to produce the results.

The tables in Appendix A also report in column '#unique' the number of *sat* or *unsat* results produced by a solver for benchmarks for which all other solvers returned *unknown*. The 'job information' files also include data about the space and memory consumption, which we consider less relevant and therefore are not reported in the tables (see also the CHC-COMP 2021 and CHC-COMP 2022 reports [15, 12]).

3 Benchmarks

3.1 Format

CHC-COMP accepts benchmarks in the SMT-LIB 2.6 format [2]. All benchmarks have to conform to the format described at `https://chc-comp.github.io/format.html`. This year, we updated the format to allow the declaration of ADTs using the `declare-datatypes` command. We support ADTs with any number of constructors and selectors as long as they are not parametric. Conformance is checked using the `format.py` script available at `https://github.com/chc-comp/scripts`.

3.2 Inventory

All benchmarks used for the competition are selected from repositories under `https://github.com/chc-comp`. Anyone can contribute benchmarks to this repository. This year, we got several new benchmarks for the track **ADT-LIA-nonlin**. Table 1 summarizes the number of benchmarks and unique benchmarks available in each repository. The organizers pick a subset of all available benchmarks for each year's competition. In the rest of this section, we explain the steps in this selection.

3.3 Processing Benchmarks

All benchmarks are processed using the `format.py` script, which is available at `https://github.com/chc-comp/scripts`. The command line for invoking the script is

```
> python3.9 format.py --out-dir <out-dir> --merge_queries True <smt-file>
```

The script attempts to put benchmark `<smt-file>` into CHC-COMP format. The `merge_queries` option merges multiple queries into a single query as discussed in previous editions of CHC-COMP [15]. In previous competitions, this script was not used in tracks containing ADTs because it did not print ADTs. This year, we updated the script to support printing ADTs in the SMT-LIB format using the `declare-datatypes` command. When printing ADTs are grouped as follows: if a constructor of ADT type a takes an argument of type ADT b, both a and b are grouped together. All ADTs in a group are declared together inside the same `declare-dataypes` command.

After formatting, benchmarks are categorized into one of the 6 competition tracks: LIA-lin, LIA-nonlin, LIA-lin-Arrays, LIA-nonlin-Arrays, ADT-LIA-nonlin, and LIA-nonlin-Arrays-nonrecADT. The scripts for categorizing benchmarks are available at `https://github.com/chc-comp/chc-tools/tree/master/format-checker`. This year, we added support for ADT tracks in the categorizing script. The script now checks for proper declaration of ADTs and proper usage of constructors, selectors, and recognizers. However, it does not check if a given ADT is recursive or not. Therefore, for the LIA-nonlin-Arrays-nonrecADT track, we manually verified that all ADTs are non-recursive. Benchmarks that could not be put in CHC-COMP compliant format and benchmarks that could not be categorized into any tracks are not used for the competition.

Repository	LIA-lin	LIA-nonlin	LIA-lin-Arrays	LIA-nonlin-Arrays	LIA-nonlin-Arrays-nonrecADT	ADT-LIA-nonlin
adtrem (*new*)						251/247
aeval	54/54					
aeval-unsafe	54/54					
chc-comp19			290/290			
eldarica-misc	149/136	69/66				
extra-small-lia	55/55					
hcai	101/87	133/131	39/39	25/25		
hopv	49/48	68/67				
jayhorn	75/73	7325/7224				
kind2		851/736				
ldv-ant-med			10/10	342/342		
ldv-arrays			3/2	821/546		
llreve	66/66	59/57	31/31			
quic3			43/43			
rust-horn (*new*)	11/11	6/6				56/56
seahorn	3379/2812	68/66				
solidity					2200/2174	
sv-comp	3150/2930	1643/1169	79/73	856/780		
synth/nay-horn		119/114				
synth/semgus				5371/4839		
tip-adt-lia (*new*)						320/320
tricera	405/405	4/4				
tricera/adt-arrays					156/156	
ultimate		8/8		23/23		
vmt	906/803					
total/**unique**	8454/**7534**	10353/**9648**	495/**488**	7438/**6555**	2356/**2330**	627/**623**

Table 1: Summary of benchmarks (total/unique).

3.4 Rating and Selection

This section describes the procedure used to select benchmarks for the competition.

We picked all unique benchmarks in the LIA-lin-Arrays track because of the scarcity of available benchmarks. In all other tracks, we followed a procedure similar to the past editions of the competition aiming at selecting a representative subset of the available benchmarks. In particular, we estimated how "easy" the benchmarks were and picked a mix of "easy" and "hard" instances. We say that a benchmark in a track is "easy" if it is solved by both the winner and the runner-up solvers in the corresponding track in CHC-COMP 2022, within a small time interval (30s).

Each benchmark was rated A/B/C based on how difficult the winner and the runner-up solvers found them. A rating of "A" is given if both solvers solved the benchmark, "B" if only one solver solved it, "C" if neither solved it, within the set timeout (30s). We ran all solvers using the same binaries and configurations submitted for CHC-COMP 2022.

Once we labeled each benchmark from a repository r, we decided the maximum number of instances, N_r, to take from the repository. N_r number was decided based on the total number of unique benchmarks and our knowledge about the benchmarks in repository r.

We picked at most $0.2 \cdot N_r$ benchmarks with rating A. Then, we picked at most $0.4 \cdot N_r$ benchmarks with rating B; namely, $0.2 \cdot N_r$ from those solved only by the winner solver and $0.2 \cdot N_r$ from those solved only by the runner-up solver. Finally, we picked at most $0.4 \cdot N_r$ benchmarks with rating C. If we did not find enough benchmarks with rating A, we picked the rest of the benchmarks with rating B (equally from those solved only by the winner and the runner-up). If we did not find enough benchmarks with rating B, we pick the remaining benchmarks from rating C.

This way, we obtained a mix of "easy" and "hard" benchmarks with a bias towards benchmarks that were not easily solved by either of the best solvers from the previous year's competition. The number of instances with each rating is given in Tables 2 and 3. The number of instances picked from each repository is given in Table 4. To pick `<num>` benchmarks of rating `<Y>`, we used the command

```
> cat <rating-Y-benchmark-list> | sort -R | head -n <num>
```

We were unable to run more than one solver for tracks containing ADTs (ADT-LIA-nonlin, LIA-nonlin-Arrays-nonrecADT). Only 3 solvers participated in tracks containing ADTs in CHC-COMP 2022: Spacer, Eldarica, and RInGen. RInGen does not support theories other than ADTs. The version of Eldarica submitted to CHC-COMP 2022 does not support the updated format of CHC-COMP 2023. Specifically, this version of Eldarica does not support the SMT-LIB syntax for recognizers[8]. Therefore, we were limited to using just one solver, Spacer, to select benchmarks for tracks containing ADTs. For each repository r, we decided a maximum number of instances N_r, ran Spacer on all benchmarks with the same timeout (30s), and picked $0.4 \cdot N_r$ benchmarks that Spacer solved (column B in Table 3) and $0.6 \cdot N_r$ benchmarks that Spacer did not solve (column C in Table 3).

The final set of benchmarks selected for CHC-COMP 2023 can be found in the github repository `https://github.com/chc-comp/chc-comp23-benchmarks`, and on StarExec in the public space `CHC/CHC-COMP-23/CHC-COMP-23-competition-runs`[9].

[8]since then, Eldarica has been updated to support recognizers. E.g. Eldarica v2.0.9 that participated in CHC-COMP 2023.

[9]`https://www.starexec.org/starexec/secure/explore/spaces.jsp?id=538230`

	LIA-lin				LIA-nonlin				LIA-nonlin-Arrays			
Repository	#A	#B		#C	#A	#B		#C	#A	#B		#C
		(*w*)	(*r*)			(*w*)	(*r*)			(*w*)	(*r*)	
aeval	12	9	4	29								
aeval-unsafe	17	0	12	25								
eldarica-misc	120	5	9	2	39	13	0	14				
extra-small-lia	30	13	8	4								
hcai	82	1	3	1	123	0	5	3	17	3	0	5
hopv	48	0	0	0	57	3	5	2				
jayhorn	73	0	0	0	3712	2275	1	1236				
kind2					650	70	0	16				
ldv-ant-med									0	128	0	214
ldv-arrays									7	195	0	344
llreve	61	0	5	0	48	4	2	3				
rust-horn	10	1	0	0	5	0	0	1				
seahorn	2089	65	69	589	60	1	2	3				
sv-comp	2854	1	74	1	1117	40	4	8	310	330	7	133
synth/nay-horn					70	20	4	20				
synth/semgus									737	2254	4	1844
tricera/svcomp20	43	7	4	351	4	0	0	0				
ultimate					0	1	0	7	0	0	0	23
vmt	711	31	7	54								
total	6150	133	195	1056	5885	2427	23	1313	1071	2910	11	2563

Table 2: The number of unique benchmarks with ratings A/B/C - Tracks: LIA-lin, LIA-nonlin, and LIA-nonlin-Arrays. B-rated benchmarks are reported in two sub-columns: (*w*) benchmarks solved only by the CHC-COMP 2022 winner, and (*r*) benchmarks solved only by the CHC-COMP 2022 runner-up solver.

	LIA-nonlin-Arrays-nonrecADT		ADT-LIA-nonlin	
Repository	#B	#C	#B	#C
adtrem			86	161
rust-horn			43	13
solidity	2109	65		
tip-adt-lia			39	281
tricera/adt-arrays	65	91		
total	2174	156	168	455

Table 3: The number of unique benchmarks with ratings B/C – Tracks: ADT-nonlin, and LIA-nonlin-Arrays-nonrecADT.

Repository	LIA-lin	LIA-nonlin	LIA-nonlin-Arrays	LIA-nonlin-Arrays-nonrecADT	ADT-LIA-nonlin
adtrem					125/125
aeval	30/30				
aeval-unsafe	30/30				
eldarica-misc	45/25	30/26			
extra-small-lia	30/22				
hcai	45/14	60/20	15/11		
hopv	30/6	30/16			
jayhorn	30/6	180/180			
kind2		90/52			
ldv-ant-med			60/60		
ldv-arrays			90/90		
llreve	30/11	45/18			
rust-horn					28/18
seahorn	90/90	45/15			
solidity				312/127	
sv-comp	90/38	90/48	135/135		
synth/nay-horn		60/48			
synth/semgus			135/135		
tip-adt-lia					160/160
tricera/svcomp20	60/60	3/0			
tricera/adt-arrays				156/122	
ultimate		6/5	15/15		
vmt	90/90				
total	600/**422**	639/**428**	450/**446**	468/**249**	313/**303**

Table 4: The number of benchmarks to select and the number of selected benchmarks from each repository.

4 Solvers

Seven solvers were submitted to CHC-COMP 2023: six competing solvers, and one solver *hors concours* (Spacer is co-developed by Hari Govind V K who is co-organizing the CHC-COMP 2023.).

Table 5 lists the submitted solvers together with the configurations used to run them on the competition tracks. Detailed descriptions of the solvers are provided in Section 7. The binaries of the solvers are available on the StarExec space `CHC/CHC-COMP/CHC-COMP-23-competitions-runs`.

Solver	LIA-lin	LIA-nonlin	LIA-lin-Arrays	LIA-nonlin-Arrays	LIA-nonlin-Arrays-nonrecADT	ADT-LIA-nonlin
Eldarica	`def`	`def`	`def`	`def`	`def`	`def`
Golem	`lia-lin`	`lia-nonlin`				
LoAT	`loat_horn`					
Theta	`fix`	`fix`	`fix`	`fix`		
Ultimate TreeAutomizer	`default`	`default`	`default`	`default`		
Ultimate Unihorn	`default`	`default`	`default`	`default`		
Spacer	`def`	`def`	`ARRAYS`	`ARRAYS`	`def`	`def`

Table 5: Solvers and configurations used in the tracks; an empty entry denotes that the solver did not enter the competition in that track. The configuration names have been taken as is from solver submissions.

5 Results

The results of the CHC-COMP 2023 are reported in Table 6. Detailed results are provided in Appendix A. All the data gathered from the execution of the StarExec jobs created for the competition run are available on the StarExec space `CHC/CHC-COMP/CHC-COMP-23-competitions-runs`.

	LIA-lin	LIA-nonlin	LIA-lin-Arrays	LIA-nonlin-Arrays	LIA-nonlin-Arrays-nonrecADT	ADT-LIA-nonlin
Winner	**Golem**	**Eldarica**	**Eldarica**	**Eldarica**	**Eldarica**	**Eldarica**
2nd place	Eldarica	Golem	Theta	Ultimate Unihorn		
3rd place	Theta	Ultimate Unihorn	Ultimate Unihorn	Theta		

Table 6: Results of CHC-COMP 2023. Spacer, which entered the competition as hors concours solver, placed in the first position of the LIA-lin, LIA-nonlin, LIA-lin-Arrays, and LIA-nonlin-Arrays tracks.

5.1 Observed Issues and Fixes during the Competition runs

This section describes the issues we have run across when using the tools entered in the competition and how we worked with the teams to overcome them.

Ultimate TreeAutomizer and Ultimate Unihorn Due to issues in building a version of Z3 that is able to run on StarExec, the final submission for the competition run of the solvers Ultimate TreeAutomizer and Ultimate Unihorn were completed on 14 April, 2023.

Theta In the competition runs of the LIA-nonlin-Arrays track we detected one inconsistent result: Theta (Theta-default in Table 7) reported *unsat* on one benchmark, while other solvers reported *sat*. The inconsistency was detected on April 14, and we informed the team on the same day by sending them the benchmark on which the issue was detected. The team submitted an updated version of Theta on April 15. Due to a configuration problem, the updated version of Theta reported *unknown* on all benchmarks. We informed the team on April 16, who provided an updated version of the solver (Theta-fix in Table 7) on the same day.

In the competition runs of the LIA-nonlin track we detected one inconsistent result: Theta-fix reported *sat*, while other solvers reported *unsat*. The Theta team was informed on April 19 by sending them the benchmark on which the issue was detected. The team submitted a fixed version (Thetafix-fix in Table 7) on April 19 that produced no inconsistent results.

The results presented in this report were produces using the fixed version. In Table 7 we report the results before and after the fixes.

Theta version	LIA-lin		LIA-nonlin		LIA-lin-Arrays		LIA-nonlin-Arrays	
	#sat	*#unsat*	*#sat*	*#unsat*	*#sat*	*#unsat*	*#sat*	*#unsat*
Theta-default	129	53	12	21	148	50	52	40
Theta-fix	121	49	9	20	135	50	45	39
Thetafix-fix	122	48	8	30	134	50	45	40

Table 7: Results produced by Theta before and after the fixes.

6 Conclusions and Final Remarks

We would like to congratulate the winners of the CHC-COMP 2023 (in alphabetical order): **Eldarica** (winner of the following tracks: LIA-nonlin, LIA-lin-Arrays, LIA-nonlin-Arrays, LIA-nonlin-Arrays-nonrecADT, and LIA-nonlin-Arrays), and **Golem** (winner of the LIA-lin track).

In organizing this edition of the competition we did our best to address some open issues discussed in the report of the CHC-COMP 2022 [12]. In particular, we have replaced the ADT-nonlin track with a more general track dealing with the combined theory of LIA and ADTs (ADT-LIA-nonlin), and we have extended the CHC format and the tools for processing and selecting the benchmarks to deal with ADTs. Moreover, as mentioned in the previous reports [15, 12], we have discontinued the obsolete tracks LRA-TS and LRA-TS-par. Finally, we have made a small change to the candidate benchmarks rating process by increasing the timeout used to evaluate their "hardness" (see Section 3.4). Ideally, we would have run the solvers with the same timeout as used in the competition (20 minutes). However, there are over 7500 benchmarks to pick from and we expect several timeouts irrespective of the time limit. Hence, for practical reasons, we set the timeout to 30 seconds for all solvers (previous editions had lower values that were dependent on the solver used to rate the benchmarks).

Below, we report the still open issues that should be further discussed for future editions, and the proposal for new tracks that emerged from the follow-up discussion we had after the presentation of the competition report at HCVS.

- **Validation of results** (also discussed in the previous editions [15, 12]). The ability of solvers to generate a witnesses (models or counter-examples) to support their results is a recurrent request by our community members. Several solvers have support for generating a witness. However, the witness is used mainly for debugging by the developers and having a common format for them is still a work in progress. As an additional issue, it is often the case that these witnesses are not for the original CHCs but for those obtained after many layers of pre-processing. Transforming these "internal" witnesses into a witness for the original problem is also a work in progress. While reaching a consensus on a common format for their encoding would require a thoughtful discussion involving all members of the CHC community, we could begin, as already proposed in the previous reports, by introducing in the CHC-COMP new tracks where the ability of producing a witness is taken into consideration in the computation of the score.
- **Status of benchmarks** (from [12]). In order to assess the correctness of the result provided by the solvers, each submitted benchmark should explicitly declare the expected result of the satisfiability problem. We propose to use the `( set-info` ⟨*keyword*⟩ ⟨*attr-value*⟩ `)` command with the `:status` as *keyword*, and either `sat` or `unsat` as *attr-value*.
- **Parallel tracks**. (Thanks to *Martin Blicha* for having sent us this note.) We propose a parallel version for each (or some) of the existing tracks. Instead of putting a limit on the CPU time, only a limit on the wall-clock time would be imposed in the parallel version. Parallel tracks can be implemented in two ways: either use the solvers' configuration submitted for the classical tracks, or allow a separate submission for the parallel tracks.

Finally, we would to stress once again that **a bigger set of benchmarks are needed**. Besides submitting their tools, all participants are invited to contribute with new benchmarks.

7 Solver Descriptions

The tool descriptions in this section were contributed by the participants, and the copyright on the texts remains with the individual authors.

7.1 Eldarica v2.0.9

Hossein Hojjat
University of Tehran, Iran

Philipp Rümmer
University of Regensburg, Germany and Uppsala University, Sweden

Algorithm. Eldarica [25] is a Horn solver applying classical algorithms from model checking: predicate abstraction and counterexample-guided abstraction refinement (CEGAR). Eldarica can solve Horn clauses over linear integer arithmetic, arrays, algebraic data-types, bit-vectors, and the theory of heaps. It can process Horn clauses and programs in a variety of formats, implements sophisticated algorithms to solve tricky systems of clauses without diverging, and offers an elegant API for programmatic use.

Architecture and Implementation. Eldarica is entirely implemented in Scala, and only depends on Java or Scala libraries, which implies that Eldarica can be used on any platform with a JVM. For computing abstractions of systems of Horn clauses and inferring new predicates, Eldarica invokes the SMT solver Princess [34] as a library.

Configuration in CHC-COMP 2023. Eldarica is in the competition run with the option `-portfolio`, which enables a simple portfolio mode. Four instances of the solver are run in parallel, with the following options:

1. `-splitClauses:0 -abstract:off`,
2. `-splitClauses:1 -abstract:off -stac`,
3. `-splitClauses:1 -abstract:off`,
4. `-splitClauses:1 -abstract:relEqs` (the default options).

`https://github.com/uuverifiers/eldarica`
BSD licence

7.2 Golem

Martin Blicha
Università della Svizzera italiana, Switzerland
Konstantin Britikov
Università della Svizzera italiana, Switzerland

Algorithm. Golem is a CHC solver under active development that provides several backend engines implementing various SMT- and interpolation-based model-checking algorithms. It supports the theory of Linear Real or Integer Arithmetic and it is able to provide witnesses for both satisfiable and unsatisfiable CHC systems. Several back-end engines are implemeted in Golem:

- `lawi` is our re-implementation of the IMPACT algorithm [32]
- `spacer` is our re-implementation of the SPACER algorithm [28] and allows Golem to solve non-linear systems.
- `tpa` is our new model-checking algorithm based on doubling abstractions using Craig interpolants [7, 6].
- `bmc` implements the standard algorithm of Bounded Model Checking [4]
- `kind` implements a basic variant of k-induction [35]
- `imc` is our implementation of McMillan's first interpolation-based model-checking algorithm [31]

Architecture and Implementation. Golem is implemented in C++ and built on top of the interpolating SMT solver OPENSMT [26] which is used for both satisfiability solving and interpolation. The only dependencies are those inherited from OPENSMT: Flex, Bison and GMP libraries.

New Features in CHC-COMP 2023. Compared to the previous year, Golem has three new back-end engines: `bmc`, `kind` and `imc`. However, these engines support only transition systems and did not participate in the competition for this reason. Additionally, the preprocessing of the input system has improved significantly, without losing the ability to produce witnesses.

Configuration in CHC-COMP 2023. For LIA-nonlin track we used only `spacer` engine; the other engines cannot handle nonlinear system yet.

```
$ golem --engine spacer
```

For LIA-lin track, we used a trivial portfolio of `lawi`, `spacer` and `tpa` (in `split-tpa` mode) running independently.

```
$ golem --engine=spacer,lawi,split-tpa
```

`https://github.com/usi-verification-and-security/golem`
MIT LICENSE

7.3 LoAT chc-comp-2023

Florian Frohn
LuFG Informatik 2, RWTH Aachen University, Germany

Jürgen Giesl
LuFG Informatik 2, RWTH Aachen University, Germany

Algorithm. The *Loop Acceleration Tool* (LoAT) [18] is based on *Acceleration Driven Clause Learning* (ADCL) [19], a novel calculus for analyzing satisfiability of CHCs. LoAT's implementation of ADCL is based on a calculus for modular *loop acceleration* [17]. It can analyze linear Horn clauses over integer arithmetic. While ADCL can also prove satisfiability of CHCs, LoAT is currently restricted to proving unsatisfiability. Besides unsatisfiability of CHCs, LoAT can also prove non-termination and lower bounds on the worst-case runtime complexity of transition systems.

Architecture and Implementation. LoAT is implemented in C++. It uses the SMT solvers Z3 [33] and Yices [14], the recurrence solver PURRS [1], and the automata library libFAUDES [30].

New Features in CHC-COMP 2023. LoAT participates in the competition for the first time. Earlier version of LoAT could not analyze CHCs, but only transition systems.

Configuration in CHC-COMP 2023. At the competition, LoAT is run with the following arguments:

`--mode reachability` for proving reachability for transition systems or unsatisfiability of CHCs, respectively

`--format horn` for specifying that the input problem is given in the SMT-LIB-format for Horn clauses

`https://loat-developers.github.io/LoAT/`
GPL licence

7.4 Theta v4.2.3

Márk Somorjai
Mihály Dobos-Kovács
Levente Bajczi
András Vörös

Department of Measurement and Information Systems
Budapest University of Technology and Economics, Hungary

Algorithm. THETA decides the satisfiability of Constrained Horn Clauses by transforming it to a formal verification problem and employing an abstraction-based model checking technique. The input set of CHCs are transformed into a formal program representation named *Control Flow Automata (CFA)* [3] in a way that the unsatisfiability of the CHC problem is equivalent to the reachability of erroneous locations in the CFA. A bottom-up transformation is used for linear CHCs while a top-down transformation is done to nonlinear CHCs [36]. The erroneous state reachability of the created CFA is then checked using *CounterExample-Guided Abstraction Refinement (CEGAR)* [8], an iterative abstraction-based model checking algorithm.

Architecture and Implementation. THETA is a highly configurable model checking framework implemented in Java [21]. It supports various formalisms for the verification programs, engineering models and timed systems, among others. Verification is done by the main CEGAR engine, which utilizes SMT solvers through an SMTLIB interface to calculate interpolants and check the feasibility of paths. The CEGAR engine can be configured to use different abstraction domains and interpolation techniques. The framework offers a number of command line tools equipped with frontends that parse the input problem into a formalism. The bottom-up and top-down transformations from CHCs to CFA are implemented as a frontends for the `xcfa-cli` tool.

Configuration in CHC-COMP 2023. THETA is run with a sequential portfolio of 3 configurations listed below, using explicit value tracking, split predicate or cartesian predicate abstraction. Interpolation was set to backwards binary interpolation or sequential interpolation, calculated by Z3 [10] as the underlying SMT solver.

1. `--domain PRED_SPLIT --refinement BW_BIN_ITP --predsplit WHOLE`
2. `--domain PRED_CART --refinement BW_BIN_ITP --predsplit WHOLE`
3. `--domain EXPL --refinement SEQ_ITP`

THETA detects whether the input CHCs are linear or not and employs a bottom-up transformation for the former and a top-down transformation for the latter. The submitted Theta version and run scripts are available in the competition archive [37].

`https://github.com/ftsrg/theta`
Apache License 2.0

[10]`https://github.com/Z3Prover/z3`

7.5 Ultimate TreeAutomizer 0.2.3-dev-ac87e89

Matthias Heizmann
University of Freiburg, Germany
Daniel Dietsch
University of Freiburg, Germany
Jochen Hoenicke
University of Freiburg, Germany
Alexander Nutz
University of Freiburg, Germany
Andreas Podelski
University of Freiburg, Germany
Frank Schüssele
University of Freiburg, Germany

Algorithm. The ULTIMATE TREEAUTOMIZER solver implements an approach that is based on tree automata [13]. In this approach potential counterexamples to satisfiability are considered as a regular set of trees. In an iterative CEGAR loop we analyze potential counterexamples. Real counterexamples lead to an *unsat* result. Spurious counterexamples are generalized to a regular set of spurious counterexamples and subtracted from the set of potential counterexamples that have to be considered. In case we detected that all potential counterexamples are spurious, the result is *sat*. The generalization above is based on tree interpolation and regular sets of trees are represented as tree automata.

Architecture and Implementation. TREEAUTOMIZER is a toolchain in the ULTIMATE framework. This toolchain first parses the CHC input and then runs the `treeautomizer` plugin which implements the above mentioned algorithm. We obtain tree interpolants from the SMT solver SMTInterpol[11] [24]. For checking satisfiability, we use the and Z3 SMT solver[12]. The tree automata are implemented in ULTIMATE's automata library[13]. The ULTIMATE framework is written in Java and build upon the Eclipse Rich Client Platform (RCP). The source code is available at GitHub[14].

Configuration in CHC-COMP 2023. Our StarExec archive for the competition is shipped with the `bin/starexec_run_default` shell script calls the ULTIMATE command line interface with the `TreeAutomizer.xml` toolchain file and the `TreeAutomizerHopcroftMinimization.epf` settings file. Both files can be found in toolchain (resp. settings) folder of ULTIMATE's repository.

`https://www.ultimate-pa.org/`
LGPLv3 with a linking exception for Eclipse RCP

[11]`https://ultimate.informatik.uni-freiburg.de/smtinterpol/`
[12]`https://github.com/Z3Prover/z3`
[13]`https://www.ultimate-pa.org/?ui=tool&tool=automata_library`
[14]`https://github.com/ultimate-pa/`

7.6 Ultimate Unihorn 0.2.3-dev-ac87e89

Matthias Heizmann
University of Freiburg, Germany
Daniel Dietsch
University of Freiburg, Germany
Jochen Hoenicke
University of Freiburg, Germany
Alexander Nutz
University of Freiburg, Germany
Andreas Podelski
University of Freiburg, Germany
Frank Schüssele
University of Freiburg, Germany

Algorithm. ULTIMATE UNIHORN reduces the satisfiability problem for a set of constraint Horn clauses to a software verfication problem. In a first step UNIHORN applies a yet unpublished translation in which the constraint Horn clauses are translated into a recursive program that is nondeterministic and whose correctness is specified by an assert statement The program is correct (i.e., no execution violates the assert statement) if and only if the set of CHCs is satisfiable. For checking whether the recursive program satisfies its specification, Unihorn uses ULTIMATE AUTOMIZER [22] which implements an automata-based approach to software verification [23].

Architecture and Implementation. ULTIMATE UNIHORN is a toolchain in the ULTIMATE framework. This toolchain first parses the CHC input and then runs the `chctoboogie` plugin which does the translation from CHCs into a recursive program. We use the Boogie language to represent that program. Afterwards the default toolchain for verifying a recursive Boogie programs by ULTIMATE AUTOMIZER is applied. The ULTIMATE framework shares the libraries for handling SMT formulas with the SMTInterpol SMT solver. While verifying a program, ULTIMATE AUTOMIZER needs SMT solvers for checking satisfiability, for computing Craig interpolants and for computing unsatisfiable cores. The version of UNIHORN that participated in the competition used the SMT solvers SMTInterpol[15] and Z3[16]. The ULTIMATE framework is written in Java and build upon the Eclipse Rich Client Platform (RCP). The source code is available at GitHub[17].

Configuration in CHC-COMP 2023. Our StarExec archive for the competition is shipped with the `bin/starexec_run_default` shell script calls the ULTIMATE command line interface with the `AutomizerCHC.xml` toolchain file and the `chccomp-Unihorn_Default.epf` settings file. Both files can be found in toolchain (resp. settings) folder of ULTIMATE's repository.

`https://www.ultimate-pa.org/`
LGPLv3 with a linking exception for Eclipse RCP

[15]`https://ultimate.informatik.uni-freiburg.de/smtinterpol/`
[16]`https://github.com/Z3Prover/z3`
[17]`https://github.com/ultimate-pa/`

References

[1] Roberto Bagnara, Andrea Pescetti, Alessandro Zaccagnini & Enea Zaffanella (2005): *PURRS: Towards Computer Algebra Support for Fully Automatic Worst-Case Complexity Analysis*, doi:10.48550/arXiv.cs/0512056.

[2] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2016): *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org.

[3] Dirk Beyer & M. Erkan Keremoglu (2011): *CPAchecker: A Tool for Configurable Software Verification*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 184–190, doi:10.1007/978-3-642-22110-1_16.

[4] Armin Biere, Alessandro Cimatti, Edmund Clarke & Yunshan Zhu (1999): *Symbolic Model Checking without BDDs*. In W. Rance Cleaveland, editor: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 193–207, doi:10.1007/3-540-49059-0_14.

[5] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan & Andrey Rybalchenko (2015): *Horn Clause Solvers for Program Verification*. In Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner & Wolfram Schulte, editors: *Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, Springer International Publishing, Cham, pp. 24–51, doi:10.1007/978-3-319-23534-9_2.

[6] Martin Blicha, Grigory Fedyukovich, Antti E. J. Hyvärinen & Natasha Sharygina (2022): *Split Transition Power Abstractions for Unbounded Safety*. In Alberto Griggio & Neha Rungta, editors: *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design – FMCAD 2022*, TU Wien Academic Press, pp. 349–358.

[7] Martin Blicha, Grigory Fedyukovich, Antti E. J. Hyvärinen & Natasha Sharygina (2022): *Transition Power Abstractions for Deep Counterexample Detection*. In Dana Fisman & Grigore Rosu, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer International Publishing, Cham, pp. 524–542, doi:10.1007/978-3-030-99524-9_29.

[8] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu & Helmut Veith (2003): *Counterexample-Guided Abstraction Refinement for Symbolic Model Checking*. *J. ACM* 50(5), p. 752–794, doi:10.1145/876638.876643.

[9] Emanuele De Angelis, Fabio Fioravanti, John P. Gallagher, Manuel V. Hermenegildo, Alberto Pettorossi & Maurizio Proeitti (2021): *Analysis and Transformation of Constrained Horn Clauses for Program Verification*. *Theory and Practice of Logic Programming*, p. 1–69, doi:10.1017/S1471068421000211.

[10] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi & Maurizio 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.

[11] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi & Maurizio Proietti (2022): *Satisfiability of constrained Horn clauses on algebraic data types: A transformation-based approach*. *J. Log. Comput.* 32(2), pp. 402–442, doi:10.1093/logcom/exab090.

[12] Emanuele De Angelis & Hari Govind V K (2022): *CHC-COMP 2022: Competition Report*. *Electronic Proceedings in Theoretical Computer Science* 373, pp. 44–62, doi:10.4204/eptcs.373.5.

[13] Daniel Dietsch, Matthias Heizmann, Jochen Hoenicke, Alexander Nutz & Andreas Podelski (2019): *Ultimate TreeAutomizer (CHC-COMP Tool Description)*. In Emanuele De Angelis, Grigory Fedyukovich, Nikos Tzevelekos & Mattias Ulbrich, editors: *Proceedings of the Sixth Workshop on Horn Clauses for Verification and Synthesis and Third Workshop on Program Equivalence and Relational Reasoning, HCVS/PERR@ETAPS 2019, Prague, Czech Republic, 6-7th April 2019*, *EPTCS* 296, pp. 42–47, doi:10.4204/EPTCS.296.7.

[14] Bruno Dutertre (2014): *Yices 2.2*. In Armin Biere & Roderick Bloem, editors: *CAV '14*, Springer International Publishing, pp. 737–744, doi:10.1007/978-3-319-08867-9_49.

[15] Grigory Fedyukovich & Philipp Rümmer (2021): *Competition Report: CHC-COMP-21*. In Hossein Hojjat & Bishoksan Kafle, editors: *Proceedings 8th Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2021, Virtual, 28th March 2021*, *EPTCS* 344, Open Publishing Association, pp. 91–108, doi:10.4204/EPTCS.344.7.

[16] Grigory Fedyukovich, Yueling Zhang & Aarti Gupta (2018): *Syntax-Guided Termination Analysis*. In Hana Chockler & Georg Weissenbacher, editors: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, *Lecture Notes in Computer Science* 10981, Springer, pp. 124–143, doi:10.1007/978-3-319-96145-3_7.

[17] Florian Frohn (2020): *A Calculus for Modular Loop Acceleration*. In Armin Biere & David Parker, editors: *TACAS '20*, Springer International Publishing, pp. 58–76, doi:10.1007/978-3-030-45190-5_4.

[18] Florian Frohn & Jürgen Giesl (2022): *Proving Non-Termination and Lower Runtime Bounds with LoAT (System Description)*. In Jasmin Blanchette, Laura Kovács & Dirk Pattinson, editors: *IJCAR '22*, Springer International Publishing, pp. 712–722, doi:10.1007/978-3-031-10769-6_41.

[19] Florian Frohn & Jürgen Giesl (2023): *ADCL: Acceleration Driven Clause Learning for Constrained Horn Clauses*, doi:10.48550/arXiv.2303.01827.

[20] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea & Andrey Rybalchenko (2012): *Synthesizing software verifiers from proof rules*. In Jan Vitek, Haibo Lin & Frank Tip, editors: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, ACM, pp. 405–416, doi:10.1145/2254064.2254112.

[21] Ákos Hajdu & Zoltán Micskei (2020): *Efficient Strategies for CEGAR-Based Model Checking*. *Journal of Automated Reasoning* 64(6), pp. 1051–1091, doi:10.1007/s10817-019-09535-x.

[22] Matthias Heizmann, Max Barth, Daniel Dietsch, Leonard Fichtner, Jochen Hoenicke, Dominik Klumpp, Mehdi Naouar, Tanja Schindler, Frank Schüssele & Andreas Podelski (2023): *Ultimate Automizer and the CommuHash Normal Form - (Competition Contribution)*. In Sriram Sankaranarayanan & Natasha Sharygina, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*, *Lecture Notes in Computer Science* 13994, Springer, pp. 577–581, doi:10.1007/978-3-031-30820-8_39.

[23] Matthias Heizmann, Jochen Hoenicke & Andreas Podelski (2013): *Software Model Checking for People Who Love Automata*. In: *CAV*, *Lecture Notes in Computer Science* 8044, Springer, pp. 36–52, doi:10.1007/978-3-642-39799-8_2.

[24] Jochen Hoenicke & Tanja Schindler (2018): *Efficient Interpolation for the Theory of Arrays*. In: *IJCAR*, *Lecture Notes in Computer Science* 10900, Springer, pp. 549–565, doi:10.1007/978-3-319-94205-6_36.

[25] Hossein Hojjat & Philipp Rümmer (2018): *The ELDARICA Horn Solver*. In: *2018 Formal Methods in Computer Aided Design, FMCAD*, pp. 1–7, doi:10.23919/FMCAD.2018.8603013.

[26] Antti E. J. Hyvärinen, Matteo Marescotti, Leonardo Alt & Natasha Sharygina (2016): *OpenSMT2: An SMT Solver for Multi-core and Cloud Computing*. In Nadia Creignou & Daniel Le Berre, editors: *Theory and Applications of Satisfiability Testing – SAT 2016*, Springer International Publishing, Cham, pp. 547–553, doi:10.1007/978-3-319-40970-2_35.

[27] Bishoksan Kafle, John P. Gallagher & José F. Morales (2016): *RAHFT: A Tool for Verifying Horn Clauses Using Abstract Interpretation and Finite Tree Automata*. In Swarat Chaudhuri & Azadeh Farzan, editors: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, *Lecture Notes in Computer Science* 9779, Springer, pp. 261–268, doi:10.1007/978-3-319-41528-4_14.

[28] Anvesh Komuravelli, Arie Gurfinkel & Sagar 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`.

[29] Yurii Kostyukov, Dmitry Mordvinov & Grigory Fedyukovich (2021): *Beyond the Elementary Representations of Program Invariants over Algebraic Data Types*. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, ACM, p. 451–465, doi:`10.1145/3453483.3454055`.

[30] *libFAUDES Library*. Available at `https://fgdes.tf.fau.de/faudes/index.html`.

[31] Kenneth L. McMillan (2003): *Interpolation and SAT-Based Model Checking*. In Warren A. Hunt & Fabio Somenzi, editors: *Computer Aided Verification*, Springer, Berlin Heidelberg, pp. 1–13, doi:`10.1007/978-3-540-45069-6_1`.

[32] Kenneth L. McMillan (2006): *Lazy Abstraction with Interpolants*. In Thomas Ball & Robert B. Jones, editors: *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 123–136, doi:`10.1007/11817963_14`.

[33] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *TACAS '08*, Springer Berlin Heidelberg, pp. 337–340, doi:`10.1007/978-3-540-78800-3_24`.

[34] Philipp Rümmer (2008): *A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic*. In: *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, *LNCS* 5330, Springer, pp. 274–289, doi:`10.1007/978-3-540-89439-1_20`.

[35] Mary Sheeran, Satnam Singh & Gunnar Stålmarck (2000): *Checking Safety Properties Using Induction and a SAT-Solver*. In Warren A. Hunt & Steven D. Johnson, editors: *Formal Methods in Computer-Aided Design*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 127–144, doi:`10.1007/3-540-40922-X_8`.

[36] Márk Somorjai, Mihály Dobos-Kovács, Zsófia Ádám, Levente Bajczi & András Vörös (2023): *Bottoms Up for CHCs: Novel Transformation of Linear Constrained Horn Clauses to Software Verification*. *Electronic Proceedings in Theoretical Computer Science*.

[37] Márk Somorjai, Mihály Dobos-Kovács, Levente Bajczi & András Vörös (2023): *Tool Archive of Theta for CHC-COMP 2023*, doi:`10.5281/zenodo.7954684`.

[38] Aaron Stump, Geoff Sutcliffe & Cesare Tinelli (2014): *StarExec: A Cross-Community Infrastructure for Logic Solving*. In Stéphane Demri, Deepak Kapur & Christoph Weidenbach, editors: *Automated Reasoning*, Springer International Publishing, Cham, pp. 367–373, doi:`10.1007/978-3-319-08587-6_28`.

[39] Hiroshi Unno, Tachio Terauchi & Eric Koskinen (2021): *Constraint-Based Relational Verification*. In Alexandra Silva & K. Rustan M. Leino, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 742–766, doi:`10.1007/978-3-030-81685-8_35`.

[40] Weikun Yang, Grigory Fedyukovich & Aarti Gupta (2019): *Lemma Synthesis for Automating Induction over Algebraic Data Types*. In Thomas Schiex & Simon de Givry, editors: *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, *Lecture Notes in Computer Science* 11802, Springer, pp. 600–617, doi:`10.1007/978-3-030-30048-7_35`.

A Detailed results

Solver	**Score**	**#sat**	**#unsat**	**CPU time/s**	**Wall-clock/s**	**#unique**
Spacer	265	199	66	274397	138310	43
Golem	229	148	81	368980	129633	8
Eldarica	219	160	59	385851	112832	23
Theta	170	122	48	426006	370425	0
U. Unihorn	103	72	31	449683	384389	0
U. TreeAutomizer	81	50	31	537858	517349	0
LoAT	50	0	50	287878	287841	4

Table 8: Solver performance on LIA-lin track

Solver	**Score**	**#sat**	**#unsat**	**CPU time/s**	**Wall-clock/s**	**#unique**
Spacer	384	235	149	90842	50781	38
Eldarica	330	185	145	218944	79522	9
Golem	310	178	132	248569	248578	3
U. Unihorn	121	72	49	470768	389915	0
Theta	38	8	30	687374	666145	0
U. TreeAutomizer	34	5	29	569895	531158	0

Table 9: Solver performance on LIA-nonlin track

Solver	**Score**	**#sat**	**#unsat**	**CPU time/s**	**Wall-clock/s**	**#unique**
Spacer	281	212	69	359439	187454	81
Eldarica	220	150	70	478284	166185	15
Theta	184	134	50	285884	271624	0
U. Unihorn	164	122	42	242113	206799	1
U. TreeAutomizer	131	96	35	239591	229783	0

Table 10: Solver performance on LIA-lin-Arrays track

Solver	Score	#sat	#unsat	CPU time/s	Wall-clock/s	#unique
Spacer	258	148	110	290925	156914	75
Eldarica	206	122	84	454921	184851	26
U. Unihorn	96	37	59	234519	199416	0
Theta	85	45	40	588095	569760	4
U. TreeAutomizer	56	6	50	276025	250747	0

Table 11: Solver performance on LIA-nonlin-Arrays track

Solver	Score	#sat	#unsat	CPU time/s	Wall-clock/s	#unique
Eldarica	176	85	91	114521	42212	57
Spacer	120	59	61	195321	107046	1

Table 12: Solver performance on LIA-nonlin-Arrays-nonrecADT track

Solver	Score	#sat	#unsat	CPU time/s	Wall-clock/s	#unique
Eldarica	58	22	36	433561	150012	30
Spacer	30	3	27	440259	290358	2

Table 13: Solvers performance on ADT-LIA-nonlin track

Bottoms Up for CHCs: Novel Transformation of Linear Constrained Horn Clauses to Software Verification

Márk Somorjai Mihály Dobos-Kovács Zsófia Ádám
Levente Bajczi András Vörös
vori@mit.bme.hu
Department of Measurement and Information Systems
Budapest University of Technology and Economics

Constrained Horn Clauses (CHCs) have conventionally been used as a low-level representation in formal verification. Most existing solvers use a diverse set of specialized techniques, including direct state space traversal or under-approximating abstraction, necessitating purpose-built complex algorithms. Other solvers successfully simplified the verification workflow by translating the problem to inputs for other verification tasks, leveraging the strengths of existing algorithms. One such approach transforms the CHC problem into a recursive program roughly emulating a *top-down* solver for the deduction task; and verifying the reachability of a safety violation specified as a control location. We propose an alternative *bottom-up* approach for linear CHCs, and evaluate the two options in the open-source model checking framework THETA on both synthetic and industrial examples. We find that there is a more than twofold increase in the number of solved tasks when the novel *bottom-up* approach is used in the verification workflow, in contrast with the *top-down* technique.

1 Introduction

Constraint Horn Clauses (CHCs) are widely used in the field of formal verification both as a means for an intermediate representation [6, 10, 14] and as a specification language [1]. Conventionally, CHCs allow the specification of *deduction* problems using implication, allowing the formalization of rules that govern how atomic facts lead to more complex (*deduced*) information.

A CHC problem can be solved in many different ways. SPACER [9] in Z3 [15] uses a solver based on automatic under-approximating abstraction; ELDARICA [13] uses a direct abstract state space traversal over the CHC formulae; and UNIHORN [1] uses a translation to recursive Boogie [3] code before applying a conventional software verification workflow to achieve a result. While the former approaches in SPACER and ELDARICA work well as demonstrated by their performance in previous years' CHC-COMP [1], a competition for CHC solvers, they require purpose-built solvers, thus incurring additional effort when developing new algorithms.

In contrast, the approach utilized by UNIHORN relies on existing algorithms, taking advantage of the tool being part of the ULTIMATE framework with proven and efficient algorithms for tackling software verification tasks [12]. By complementing the framework with a new front-end for parsing and transforming CHC formulae, the same verification workflows can be applied to the CHC-based problems as well, enabling their efficient verification.

The transformation step used by UNIHORN creates Boogie code that roughly emulates a program capable of deducing the existence of facts necessary to reach some end goal (e.g., a safety violation). We refer to this approach as *top-down* [18] or *backward*.

In this paper, we introduce an alternative to the *backward* method, which creates a program that emulates a *bottom-up* solver [18] (i.e., starting from nondeterministic facts and trying to deduce a safety violation using the formulae). We implement this *forward* transformation to another formal representation of programs, the Control Flow Automaton (CFA), alongside with a *backward* transformation alternative, in THETA [11]. Our benchmarks show that using the proposed approach increased the number of successfully solved CHCs more than twofold on linear CHC verification tasks from the CHC-COMP benchmark suite [1].

This paper is structured as follows. In Section 2, we introduce the necessary background concepts. Then, in Section 3, we present our proposed *forward* transformation and the accompanying verification workflow, as well as the theory behind proof- and counterexample-generation. Finally, in Section 4, we present our experimental results comparing the effect of using the existing *backward* transformation versus the novel *forward* transformation on the performance of the verification workflow.

2 Background

In this section, we introduce the theoretical background for the paper, including *software verification*, *control flow automata* (CFAs), and *Counterexample-Guided Abstraction Refinement* (CEGAR).

2.1 Formal Software Verification

The goal of software verification is to mathematically prove certain properties of a program. One such property is the reachability of labelled control locations. A program is *unsafe* if such a location can be reached from the initial location of the program using a finite number of transitions; otherwise, it is *safe*. Due to the uncertainties and complexity of dealing with high-level programming languages, the input is first transformed into a formal representation [2]. *Model checking* is then often employed [8], which explores the state space of the program, thus verifying the reachability of the error states. While generally this problem is undecidable [17], and enumerating the state space naively is infeasible in practice [5], there exist efficient algorithms for solving a subset of the input tasks, such as the Counterexample-Guided Abstraction Refinement (CEGAR) technique [4].

2.1.1 Control Flow Automata

A *Control Flow Automaton* represents a program as a directed graph. Formally, a control flow automaton is a tuple $CFA = (V, L, l_0, E)$, where:

- V: A set of *variables*, where each $v \in V$ can have values from its domain D_v.
- L: A set of *locations*, where each *location* can be interpreted as a possible value of the program counter.
- $l_0 \in L$: The *initial location*, that is active at the start of the program.
- $E \subseteq L \times Ops \times L$: A set of transitions, where a transition is a directed edge going from one location in L to another, with a label $op \in Ops$, where Ops is a set of operations that can be executed as the program advances from one location to another. An $op \in Ops$ can be one of the following:
  - $v = expr$: An assignment of a variable, where the value of $v \in V$ becomes the evaluation of the right-hand side $expr$.
  - $havoc\ v$: A non-deterministic assignment of a variable, after which the value of $v \in V$ can be anything from its domain D_v.
  - $[cond]$: A *guard* operation, where $cond$ is an expression that evaluates to a boolean value. The transition can only be executed if the $cond$ in the *guard* evaluates to $true$.

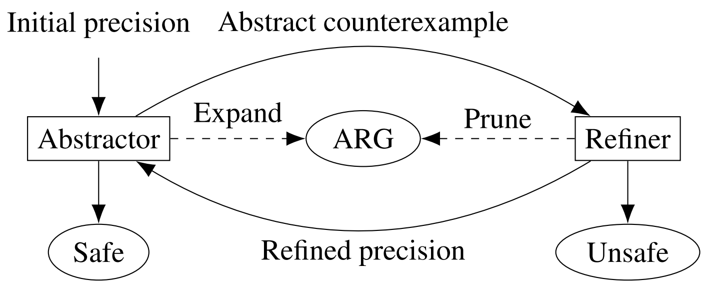


Figure 1: The CEGAR loop

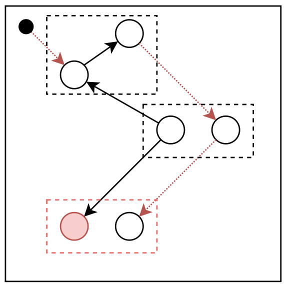


Figure 2: Abstract state space

In formal software verification, it is also useful to distinguish *error locations*, which are locations where the program would behave in an undesirable way, as well as *final locations*, which have no *outgoing transitions*.

The representation of program execution on the CFA consists of an alternating sequence of locations and operations, where at each location, the *state* of the CFA can be described as $S = (l_S, d_1, d_2, ..., d_n)$, where:

- $l \in L$ is the current location of the program,
- $d_1, d_2, ..., d_n$ are the values of all variables, that is $v_i = d_i, v_i \in V, d_i \in D_{v_i}$, for every $1 \leq i \leq |V|$.

All possible states of the CFA make up the *state space* of the program. The operations in an alternating sequence (representing an execution of the program) can then be interpreted as *transitions* in the state-space of the program.

2.2 Counterexample-Guided Abstraction Refinement

Counterexample-Guided Abstraction Refinement (CEGAR) [4] is an abstraction-based model checking algorithm.

The core of the algorithm is the CEGAR-loop (Figure 1), made up of two main parts: the *abstractor* and the *refiner*. The abstractor builds the *Abstract Reachability Graph* (ARG, a directed and acyclic graph containing abstract states and interconnecting transitions) using the *expand* operation and *covering* relation on abstract states. A parameter of abstraction is precision, which describes how much information about a concrete state is abstracted in the abstract state. An abstract state is an overapproximation of the possible concrete states (as seen in Figure 2), consequently, if no abstract error-state is reachable, then no concrete error-state is reachable, meaning the program is *safe*.

On the other hand, if an abstract error-state is reachable, the abstractor produces an *abstract counterexample*, starting at the initial abstract state and ending in an abstract error state. The refiner then decides whether a concrete error state is reachable in the abstract error state. If it can be reached, then the program is *unsafe*, and the path from the initial location of the CFA to a concrete error state is presented as a counterexample.

However, if a concrete error-state is not reachable, then the reachability of the abstract error state is a result of the overapproximation of abstraction, as demonstrated in Figure 2. Thus, the abstraction needs to be *refined* so that the abstract error state does not contain the unreachable concrete error state. This results in a refined precision, which is passed back to the abstractor after all unreachable abstract states are removed (*pruned*) from the abstract state-space.

The CEGAR loop is repeated until it either finds a concrete counterexample to the safety of the program or proves that no abstract error-state is reachable, that is, all nodes in the ARG are either expanded or covered. In the first case, the program is *unsafe*, while in the latter, it is *safe*.

3 Transforming Constrained Horn Clauses to Control Flow Automata

In this section, we present a novel approach of CHC to CFA transformation. The goal of this transformation is to create a CFA from a linear CHC in a way that turns the SMT problem of satisfiability in a CHC into a software verification question of erroneous state reachability in the CFA, so that model checking techniques can be used to decide both. More specifically, an erroneous state in a CFA should be reachable if, and only if the CHC is unsatisfiable. In this case, a refutation of the satisfiability should be given; otherwise a satisfying model ought to be generated. The approach is summarized in Figure 3.

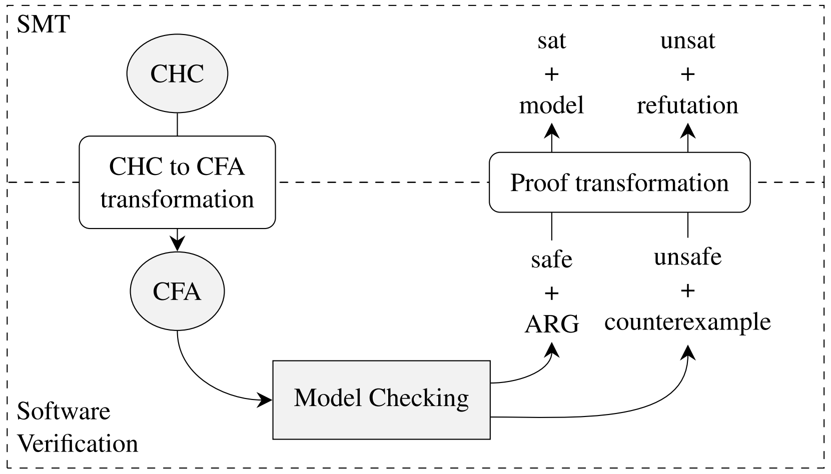


Figure 3: Overview of the presented work.

The transformation consists of two parts: the mapping of CHCs to CFAs, and the generation of a model/refutation from the output of model checking. These are represented in Figure 3 by the boxes *CHC to CFA transformation* and *Proof transformation*, respectively, and are not to be confused with *forward* and *backward* transformations described later on. As seen in the figure, proof transformation requires the utilized model checking algorithm to provide a counterexample when the CFA is deemed unsafe, and to produce an ARG when the CFA is safe.

The main idea behind the CHC to CFA transformation is to represent the uninterpreted functions as locations in the CFA, map CHCs to edges guarded by the conditions in the CHC, and use local variables to model the implications of deductions. The deducibility of a predicate with certain parameters can then be represented by the corresponding location's reachability during verification, with the given parameters as the variables' values. The source of the edges of fact CHCs can be the initial location of a CFA, since these do not have any preconditional predicates in their bodies. The target of the edge of a query CHC can then be an error location, which can only be reached if the conditions on an incoming edge are satisfied, similarly to how $\perp$ is deduced. If the error location can be reached from the initial location, then the counterexample contains the path of edges to it, which can then be mapped to their CHCs to show a sequence of CHCs that deduce $\perp$ from facts. On the other hand, if the error location is unreachable, then the explored abstract states can be used to define the uninterpreted functions to provide a satisfying model.

One way of approaching the problem of CHC satisfiability is to start with the facts, and try to apply the induction and query CHCs to deduce $\bot$. This is called the *forward* or *bottom-up* approach, which is what our main contribution, the *forward transformation* employs. Another approach is to recursively check what would be required to satisfy the body of the query CHC, stopping only when all requirements are satisfied by facts. We refer to this as a *backward* or *top-down* approach, which is used by ULTIMATE UNIHORN [1] to transform CHCs into program code.

An example CHC problem will be used throughout the chapter to demonstrate the transformations.

Example 1 *Consider the following CHC problem within integer arithmetic:*

$$A(n) \leftarrow n > 0 \land n < 100 \tag{1}$$
$$B(n,x) \leftarrow A(n) \land x > 0 \tag{2}$$
$$C(y,x) \leftarrow B(n,x) \land y = n - x \land y > 0 \tag{3}$$
$$A(n) \leftarrow C(y,x) \land n = y + (y \bmod x) \tag{4}$$
$$\bot \leftarrow A(n) \land n \geq 100 \tag{5}$$

The fact states that $A(n)$ needs to evaluate to true for $0 < n < 100$, while the satisfiability of the query depends on $A(n)$ being false for $n \geq 100$ and $n \leq 0$. What makes this problem non-trivial is the cyclic deductions between the predicates A, B and C: B can be deduced from A, C can be deduced from B, and A can be deduced from C under certain conditions. Trying a naive, manual deduction approach becomes a bit cumbersome here, due to the possibility of an infinite deduction cycle and the high number of combinations possible between the variables' values.

One may notice that n can not increase in the cycle since no matter what the subtracted x is, it will always be larger than the $y \bmod x$ that is added to n in a cycle. In the following, it will be shown that the problem is indeed satisfiable, by transforming it into a software verification problem and synthesizing a satisfying model from its proof. The CFA resulting from the transformation can be seen on Figure 4. The effect of each step on the CFA is explained as the steps are introduced.

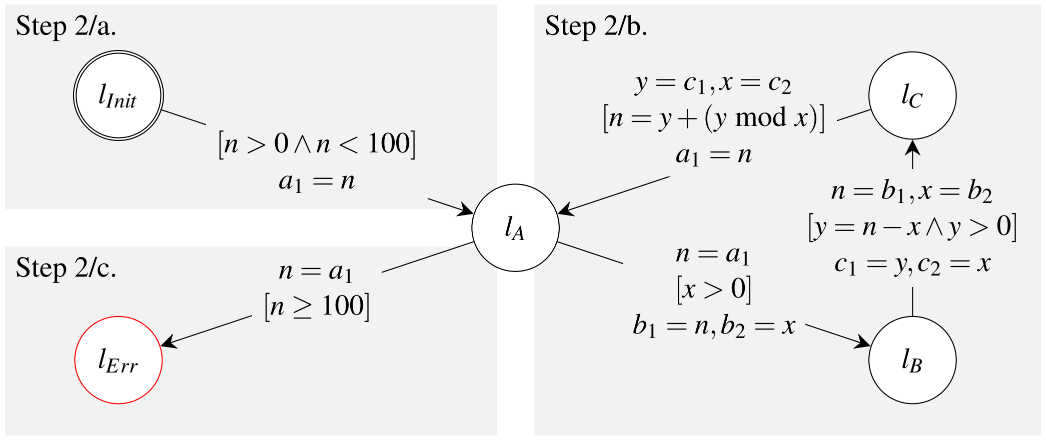


Figure 4: CFA of Example 1 after forward transformation.

3.1 Constrained Horn Clause Transformation

The transformation first creates the locations and variables of the CFA, then maps the CHCs to edges in different ways for fact, induction and query CHCs.

Consider the linear CHC problem with CHC set $\{C_1, C_2, \dots, C_k\}$ over the following uninterpreted functions:

$$B_1(b^1_1, b^1_2, \dots, b^1_{m_1}), B_2(b^2_1, b^2_2, \dots, b^2_{m_2}), \dots, B_n(b^n_1, b^n_2, \dots, b^n_{m_n})$$

That is, each CHC $C_l, \forall l \in \{1,2,\dots,k\}$ takes one of the following three forms for some $i, j \in \{1,2,\dots,k\}$:

$$\begin{aligned} B_i(x_1, x_2, \dots, x_{m_i}) &\leftarrow \varphi_l, \\ B_i(x_1, x_2, \dots, x_{m_i}) &\leftarrow B_j(y_1, y_2, \dots, y_{m_j}) \land \varphi_l, \\ \bot &\leftarrow B_j(y_1, y_2, \dots, y_{m_j}) \land \varphi_l, \end{aligned}$$

where φ_l is the interpreted formula in the body of C_l. As before, CHCs in these forms are referred to as facts, inductions and queries, respectively.

Step 1. Create CFA locations and variables

The uninterpreted functions $B_1(b^1_1, b^1_2, \dots, b^1_{m_1})$, $\dots$, $B_n(b^n_1, b^n_2, \dots, b^n_{m_n})$ are mapped to the $CFA = (V, L, l_{Init}, E)$, where:

- $V = \{b^i_j \mid \forall i \in \{1,2,\dots,n\} : \forall j \in \{1,2,\dots,m_i\}\}$,
- $L = \{l_{Init}, l_{Err}, l_1, l_2, \dots, l_n\}$,
- l_{Init},
- $E = \varnothing$.

Semantically, a new location is created for each uninterpreted function, along with an initial location l_{Init} and a distinguished error location l_{Err}. In addition, a unique variable is created for each parameter in every predicate. It is worth noting that the edge set is empty at this point, because edges are added in the next step of the transformation.

The motivation behind creating a location and variables for every uninterpreted function is that this way, a location's reachability with certain variable values can be directly mapped to the predicate's evaluation with said variable values as parameters: if a location l_i representing C_i is reachable with some values for variables $b^i_1, b^i_2, \dots, b^i_{m_i}$, then $C_i(b^i_1, b^i_2, \dots, b^i_{m_i})$ should evaluate to true. On the other hand, if l_i can not be reached with variables $b^i_1, b^i_2, \dots, b^i_{m_i}$, then $C_i(b^i_1, b^i_2, \dots, b^i_{m_i})$ ought to evaluate to false.

Example 2 *From Example 1, the first step of the forward transformation would create the $CFA = (V, L, l_{Init}, \varnothing)$, with locations $L = \{l_{Init}, l_{Err}, l_A, l_B, l_C\}$ and variables $V = \{a_1, b_1, b_2, c_1, c_2\}$. The created locations can be seen in white on the CFA in Figure 4.*

Step 2. Create CFA edges

In this step, each CHC is transformed into an edge in the CFA created in Step 1. Each kind of CHC (fact, induction, query) is treated differently, as described in the following subsections. The goal of this mapping is for the transition on the edge to only be possible, when the head of the CHC is deducible from the body of it.

Step 2/a. Create fact edges

For each fact CHC $C_l : B_i(x_1, x_2, \dots, x_{m_i}) \leftarrow \varphi_l$ where $i \in \{1,2,\dots,n\}$, an edge is created from the initial location l_{Init} to l_i, the location representing B_i. The labels on the created edge consist of the following, in the specified order:

- φ_l, the interpreted formula in the CHC's body as a guard,
- $b^i_1 = x_1, b^i_2 = x_2, \ldots, b^i_{m_i} = x_{m_i}$, assignment of the passed values to the variables corresponding to the input parameters.

Fact CHCs are named facts because they can be deduced just from the background theory $\top$, when the interpreted formula φ_l is true. The created edge from the initial location mimics this, since the target of an edge will be reachable from the initial location when the guard φ is true.

To put it more formally, the head of a fact CHC $B_i(x_1,x_2,\ldots,x_{m_i})$ is only deducible when its body, the interpreted formula φ_l is true. Similarly, the location l_i is only reachable from the initial location l_{Init} of the CFA using the created edge, when its guard φ_l evaluates to true. Furthermore, the parameters $x_1,x_2,\ldots,x_{m_i}$ are assigned to $b^i_1,b^i_2,\ldots,b^i_{m_i}$, meaning that the constraints of φ_l on the parameters are applied to the variables related to the location, just as they are applied when deducing $B_i(x_1,x_2,\ldots,x_{m_i})$. Thus, we can conclude that l_i is only reachable using the created edge with variables $b^i_1,b^i_2,\ldots,b^i_{m_i}$ valued $x_1,x_2,\ldots,x_{m_i}$, when $B_i(x_1,x_2,\ldots,x_{m_i})$ is deducible using C_l.

Example 3 *In Example 1, the second step of the forward transformation for fact CHCs would create the edge $e = (l_{Init}, op, l_A)$ from Equation 1, where the guard of op would be $n > 0 \land n < 100$, and the assignments would consist of $a_1 = n$, since a_1 is the variable corresponding to the first (and only) parameter of the predicate A. The created edge can be seen in the top-left gray rectangle on the CFA in Figure 4.*

Step 2/b. Create induction edges

For each induction CHC $C_l : B_i(x_1,x_2,\ldots,x_{m_i}) \leftarrow B_j(y_1,y_2,\ldots,y_{m_j}) \land \varphi_l$ where $i,j \in \{1,2,\ldots,n\}$, an edge is created from l_j (the location representing B_j) to l_i (the location representing B_i). The labels on the created edge consist of the following, in the specified order:

- $y_1 = b^j_1, y_2 = b^j_2, \ldots, y_{m_j} = b^j_{m_j}$, assignment of the variables corresponding to the input parameters of B_j to the passed values,
- φ_l, the interpreted formula in the CHC's body as a guard,
- $b^i_1 = x_1, b^i_2 = x_2, \ldots, b^i_{m_i} = x_{m_i}$, assignment of the passed values to the variables corresponding to the input parameters of B_i.

In addition to the first assignments, $x_1,x_2,\ldots,x_{m_i}$ and all variables in φ_l need to be uninitialized with a *havoc* statement to ensure that the semantics of $\forall$ in the CHCs are kept. However, the *havoc* statements are omitted from the examples for ease of readability. The order of instructions is also important: the assignments from the source location's variables need to happen before φ_l is evaluated.

Induction CHCs embody deductions from their bodies to their heads with some conditions φ_l. Assuming that l_j could have only been reached if it is deducible with some parameters, then this edge resembles the same: one can only go to l_i from l_j when φ_l is true.

More formally, the head of an induction CHC $B_i(x_1,x_2,\ldots,x_{m_i})$ is only deducible, when φ_l is true and $B_j(y_1,y_2,\ldots,y_{m_j})$ is deducible. Similarly, the location l_i can only be reached from l_j once l_j has been reached and the guard φ_l evaluates to true. Furthermore, the variables $b^j_1,b^j_2,\ldots,b^j_{m_j}$ are assigned to $y_1,y_2,\ldots,y_{m_j}$ and the parameters $x_1,x_2,\ldots,x_{m_i}$ are assigned to $b^i_1,b^i_2,\ldots,b^i_{m_i}$, meaning that the constraints of φ_l are applied to the y parameters and the b^i variables related to the location l_i, just as they are applied when deducing $B_i(x_1,x_2,\ldots,x_{m_i})$ from $B_j(y_1,y_2,\ldots,y_{m_j})$. Thus, we can conclude that l_i is only reachable using the created edge with variables $b^i_1,b^i_2,\ldots,b^i_{m_i}$ valued $x_1,x_2,\ldots,x_{m_i}$ from l_j with variables $b^j_1,b^j_2,\ldots,b^j_{m_j}$ valued $y_1,y_2,\ldots,y_{m_j}$, when $B_i(x_1,x_2,\ldots,x_{m_i})$ is deducible from $B_j(y_1,y_2,\ldots,y_{m_j})$ using C_l.

Example 4 *From Example 1, the second step of the forward transformation for induction CHCs would create three edges from Equation 2, 3 and 4:*

- $e_1 = (l_A, op_1, l_B)$ *for* $B(n,x) \leftarrow A(n) \wedge x > 0$*, where* op_1 *consists of the assignment* $n = a_1$*, then the guard* $x > 0$*, and the assignments* $b_1 = n, b_2 = x$ *at last,*
- $e_2 = (l_B, op_2, l_C)$ *for* $C(y,x) \leftarrow B(n,x) \wedge y = n - x \wedge y > 0$*, where* op_2 *consists of the assignments* $n = b_1, x = b_2$*, then the guard* $y = n - x \wedge y > 0$*, and the assignments* $c_1 = y, c_2 = x$ *at last,*
- $e_3 = (l_C, op_3, l_A)$ *for* $A(n) \leftarrow C(y,x) \wedge n = y + (y \bmod x)$*, where* op_3 *consists of the assignments* $y = c_1, x = c_2$*, then the guard* $n = y + (y \bmod x)$*, and the assignment* $a_1 = n$ *at last.*

The created edges can be seen in the right-hand side gray rectangle on the CFA in Figure 4.

Step 2/c. Create query edges

For each query CHC $C_l : \perp \leftarrow B_j(y_1, y_2, \ldots, y_{m_j}) \wedge \varphi_l$ where $j \in \{1, 2, \ldots, n\}$ an edge is created to the error location l_{Err} from l_j, the location representing B_j. The labels on the created edge consist of the following, in the specified order:

- $y_1 = b_1^j, y_2 = b_2^j, \ldots, y_{m_j} = b_{m_j}^j$, assignment of the variables corresponding to the input parameters to the passed values,
- φ_l, the interpreted formula in the CHC's body as a guard.

The bodies of CHC queries should not be deducible, otherwise $\perp$ can be deduced and the problem is unsatisfiable. This behaviour is captured by the created edge: if the edge's source is reachable with values that make the guard of the edge true, then the error location is reachable, making the program unsafe.

In a formal way, the head of the query CHC $\perp$ is only deducible when both $B_j(y_1, y_2, \ldots, y_{m_j})$ is deducible, and φ_l is true. Similarly, the error location l_{Err} can only be reached from l_j once l_j has been reached and the guard φ_l evaluates to true. Furthermore, the variables $b_1^j, b_2^j, \ldots, b_{m_j}^j$ are assigned to $y_1, y_2, \ldots, y_{m_j}$, meaning that the constraints of φ_l are applied to the y parameters, just as they are applied when deducing $\perp$ from $B_j(y_1, y_2, \ldots, y_{m_j})$. Thus, we can conclude that l_{Err} is only reachable using the created edge from l_j with variables $b_1^j, b_2^j, \ldots, b_{m_j}^j$ valued $y_1, y_2, \ldots, y_{m_j}$, when $\perp$ is deducible from $B_j(y_1, y_2, \ldots, y_{m_j})$ using C_l.

Example 5 *In Example 1, the second step of the forward transformation for query CHCs would create the edge* $e = (l_A, op, l_{Err})$ *from Equation 5, where op would consist of the assignment* $n = a_1$ *and the guard* $n \geq 100$*. The created edge can be seen in the bottom-left gray rectangle on the CFA in Figure 4.*

To summarize, first a location l_i and variables $b_1^i, b_2^i, \ldots, b_{m_i}^i$ are created for each uninterpreted function $B_i(b_1^i, b_2^i, \ldots, b_{m_i}^i)$, then all CHCs are transformed into edges. Since the edges are created in a way that l_i can only be reached with the corresponding variables $b_1^i, b_2^i, \ldots, b_{m_i}^i$ valued $x_1, x_2, \ldots, x_{m_i}$ if, and only if $B_i(x_1, x_2, \ldots, x_{m_i})$ can be deduced, we can conclude that the described transformation successfully converts the problem of satisfiability into a question of error location reachability. Thus, using a model checker to decide the latter will yield a result for the former as well: if the CFA is *unsafe*, the CHC problem is *unsatisfiable*; if the CFA is *safe*, the CHC problem is *satisfiable*.

It is worth to consider what the transformation results in, when there is no fact or query CHC in the set of CHCs. In the former case, there will not be any outgoing edges from the initial location of the

CFA. As a result, none of the locations will be reachable, meaning the predicates need not be true for any input, which can be expressed as $B_i \equiv false, \forall i \in \{1,2,\dots,n\}$.

In the latter case, there will not be any edges going to the error location of the CFA. As a result, all locations are reachable in the abstract state $\top$, meaning the predicates can be true for any input, which can be expressed as $B_i \equiv true, \forall i \in \{1,2,\dots,n\}$.

3.2 Proof Transformation

Proof transformation is the step of converting the result of the model checking algorithm to an answer to the CHC problem. This consists of two parts, depending on the result: the generation of a satisfying model from the ARG built during verification, or the creation of a refutation from the counterexample provided by the model checking algorithm.

3.2.1 Satisfying Model Generation

An SMT problem is called *satisfiable*, when a *model* (i.e., an assignment to constants) fulfilling all constraints exists. In the case of a CHC problem this means the definition of all uninterpreted functions $B_1(b^1_1, b^1_2, \dots, b^1_{m_1})$, $B_2(b^2_1, b^2_2, \dots, b^2_{m_2})$, …, $B_n(b^n_1, b^n_2, \dots, b^n_{m_n})$ present in the set of CHCs, that satisfy all of the CHCs.

The transformation in Subsection 3.1 ensures that a location l_i in the CFA can only be reached with the corresponding variables $b^i_1, b^i_2, \dots, b^i_{m_i}$ valued $x_1, x_2, \dots, x_{m_i}$ if, and only if $B_i(x_1, x_2, \dots, x_{m_i})$ can be deduced. If a node $S_j = (l_i, L^j_1, \dots, L^j_{k_j})$ is present in the ARG, it means l_i has been reached under the condition $L^j_1 \wedge \dots \wedge L^j_{k_j}$. Consequently, it is guaranteed that B_i can be deducted under the condition $L^j_1 \wedge \dots \wedge L^j_{k_j}$. This is true for all $S^i = \{S_j \,|\, S_j = (l_i, L^j_1, \dots, L^j_{k_j})\}$ nodes in the ARG, therefore B_i needs to evaluate to true under either of their conditions, which can be represented by concatenating them with $\vee$. This gives the following the definition for $B_i, \forall i \in \{1,2,\dots,n\}$:

$$B_i(b^i_1, b^i_2, \dots, b^i_{m_i}) = \bigvee_{S_j=(l_i, L^j_1, \dots, L^j_{k_j})}^{S^i} \left(L^j_1 \wedge \dots \wedge L^j_{k_j} \right) \tag{6}$$

At the end of verification of a safe CFA, the ARG is fully expanded, i.e., all reachable abstract states have been visited and none are in an erroneous location. Furthermore, no erroneous state can be reached from any of the nodes in the ARG. Therefore the definitions provided by Equation 6 guarantee that there can not be a deduction to $\bot$, meaning they satisfy the CHC problem.

The type of information present in any L^j needs to be taken into consideration when defining the function. If L^j contains information about any other variable x then the variables $b^i_1, b^i_2, \dots, b^i_{m_i}$ representing the input parameters of B_i, then unless some information about a b^i is dependent on x (e.g. $b^i_1 > x$), L^j can be left out. If there is a dependent b^i, then x needs to be defined with a universal quantifier inside the function ($\forall x$).

Example 6 *Applying model checking with predicate abstraction to the CFA in Figure 4 may result in the Abstract Reachability Graph (ARG) seen in Figure 5. The regular arrows represent transitions between abstract states, while the dotted arrow denotes that the source abstract state is covered by the target abstract state.*

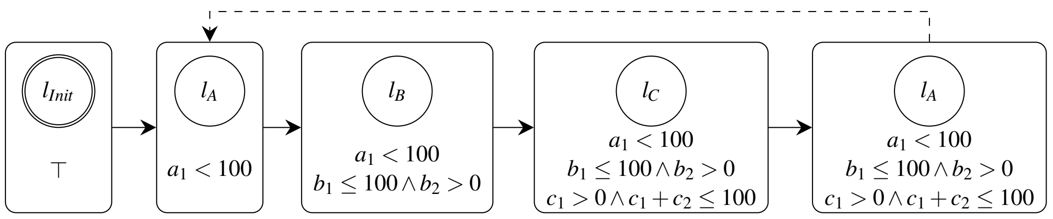

l_{Init}
$\top$
l_A
$a_1 < 100$
l_B
$a_1 < 100$
$b_1 \leq 100 \wedge b_2 > 0$
l_C
$a_1 < 100$
$b_1 \leq 100 \wedge b_2 > 0$
$c_1 > 0 \wedge c_1 + c_2 \leq 100$
l_A
$a_1 < 100$
$b_1 \leq 100 \wedge b_2 > 0$
$c_1 > 0 \wedge c_1 + c_2 \leq 100$

Figure 5: ARG resulting from the model checking of the CFA in Figure 4.

As described in Example 2, the uninterpreted function $A(n)$ corresponds to the location l_A and the variable a_1. Therefore its definition depends on the predicates of the abstract states that are in l_A, more specifically $(l_A, a_1 < 100)$ and $(l_A, a_1 < 100 \wedge b_1 \leq 100 \wedge b_2 > 0 \wedge c_1 > 0 \wedge c_1 + c_2 \leq 100)$. Using these states, we can define $A(n)$ as the disjunction of the predicates by converting a_i to n: $A(n) = n < 100 \vee (n < 100 \wedge b_1 \leq 100 \wedge b_2 > 0 \wedge c_1 > 0 \wedge c_1 + c_2 \leq 100), \forall b_1, b_2, c_1, c_2$. Since predicates of n do not depend on other variables, they can be left out, leading to $A(n) = n < 100 \vee n < 100 = n < 100$.

Similarly, $B(n,x)$ can be defined using abstract states that are in l_B, namely the single abstract state $(l_B, a_1 < 100 \wedge b_1 \leq 100 \wedge b_2 > 0)$. Converting b_1 and b_2 back to n and x gives $B(n,x) = a_1 < 100 \wedge n \leq 100 \wedge x > 0, \forall a_1$, which can also be simplified to $B(n,x) = n \leq 100 \wedge x > 0$ by omitting unused variables.

Lastly, $C(y,x)$ is defined using the abstract state $(l_C, a_1 < 100 \wedge b_1 \leq 100 \wedge b_2 > 0 \wedge c_1 > 0 \wedge c_1 + c_2 \leq 100)$. Converting c_1 and c_2 back to y and x results in $C(y,x) = a_1 < 100 \wedge b_1 \leq 100 \wedge b_2 > 0 \wedge y > 0 \wedge y + x \leq 100, \forall a_1, b_1, b_2$, which leads to the definition of $C(y,x) = y > 0 \wedge y + x \leq 100$ after getting rid of unused variables.

While it may not be trivial to see why this definition is a good model of the CHC problem, part of the reasoning is that using the definition of $A(n) = n < 100$, the query CHC Equation 5 takes the form $\perp \leftarrow n < 100 \wedge n \geq 100$. The body of this CHC is clearly unsatisfiable, thus, $\perp$ cannot be deduced.

3.2.2 Refutation Creation

When a CHC problem is unsatisfiable, a deduction can be found from the facts to $\perp$ that is always valid, regardless of how the uninterpreted functions are defined. The refutation is then a series of applications of the CHCs in the CHC set that start with a fact CHC and end with a satisfiable query CHC.

The counterexample provided by the model checker is an alternating sequence of concrete states of the CFA and edges. It starts at the initial location of the CFA with some values assigned to the variables and ends in the error location. The transformation described in Subsection 3.1 ensures that a location l_i in the CFA can only be reached with the related variables $b^i_1, b^i_2, \ldots, b^i_{m_i}$ valued $x_1, x_2, \ldots, x_{m_i}$ if, and only if $B_i(x_1, x_2, \ldots, x_{m_i})$ can be deduced. Consequently, all predicates corresponding to the locations of the concrete states in the counterexample are deducible, with the valuations present in the concrete states as parameters. The transformation also creates a one-to-one mapping of CHCs and edges. Thus, mapping the edges in the counterexample back to their CHCs, with the values of variables in the concrete states substituted as parameters, amounts to a valid refutation of the CHC problem's satisfiability.

Example 7 *Since the motivating Example 1 is satisfiable, consider a modified version of it, in which the only fact is replaced with $A(n) \leftarrow n > 0 \wedge n \leq 100$. The forward generated CFA would be similar to the one in Figure 4, with the exception of the edge going from l_{Init} to l_A having $n \leq 100$ instead of $n < 100$ in its guard.*

The model checking algorithm would return the following counterexample, with the irrelevant variable values omitted:

$$\begin{aligned}&(l_{Init}, n = 100)\\&(l_{Init}, ([n > 0, n \leq 100], a_1 = n), l_A)\\&(l_A, n = 100, a_1 = 100)\\&(l_A, (n = a_1, [n >= 100]), l_{Err})\\&(l_{Err}, n = 100, a_1 = 100)\end{aligned}$$

This could be mapped to the refutation below:

$$\begin{aligned}A(n) &\leftarrow (n > 0 \wedge n <= 100) \wedge n = 100\\ \bot &\leftarrow (A(n) \wedge n \geq 100) \wedge n = 100\end{aligned}$$

Since all variables have values assigned to them, it is trivial to check that this is indeed unsatisfiable.

4 Evaluation

Implementation The CHC to CFA transformation steps were implemented as ANTLR frontends [16] in the open-source model checking framework THETA [11]. The implementation is able to check the satisfiability of a CHC problem; however the generation of refutations and proofs is not implemented yet. Backward transformation was also implemented in a similar manner in the tool for comparison.

Goals and Design The aim of this evaluation is to show the effectiveness of the bottom-up approach by comparing it to the top-down approach. It also aims to study the performance of the approach with different configurations of CEGAR, e.g., different abstract domains.

The main comparison was done inbetween configurations of THETA only. Thus we were able to compare the different transformation approaches while the verification process was the same. Additionally, we also compared THETA to other state-of-the-art CHC solvers.

The implementation was evaluated on 585 linear CHCs over the background theory of linear integer arithmetic from the LIA-Lin track of the CHC-COMP21 benchmark repository[1]. The benchmarks were run on machines with 8 logical CPU cores and 16 GB of memory, with a timeout of 300 seconds.

domain	interpolation	pred-split	transformation	
			BACKWARD	FORWARD
EXPL	NWT_IT_WP	-	77	138
EXPL	NWT_WP_LV	-	82	137
EXPL	SEQ_ITP	-	81	175
PRED_BOOL	BW_BIN_ITP	WHOLE	110	288
PRED_CART	BW_BIN_ITP	WHOLE	141	302
PRED_SPLIT	SEQ_ITP	ATOMS	131	310
PRED_SPLIT	SEQ_ITP	WHOLE	142	318
PRED_SPLIT	BW_BIN_ITP	ATOMS	83	291
PRED_SPLIT	BW_BIN_ITP	WHOLE	114	328

Table 1: Number of solved tasks by certain configurations.

[1] https://github.com/chc-comp/chc-comp21-benchmarks

THETA (FW)	**328**
THETA (BW)	142
ELDARICA	337
UNIHORN	380
Z3	437

Table 2: Comparison to other tools.

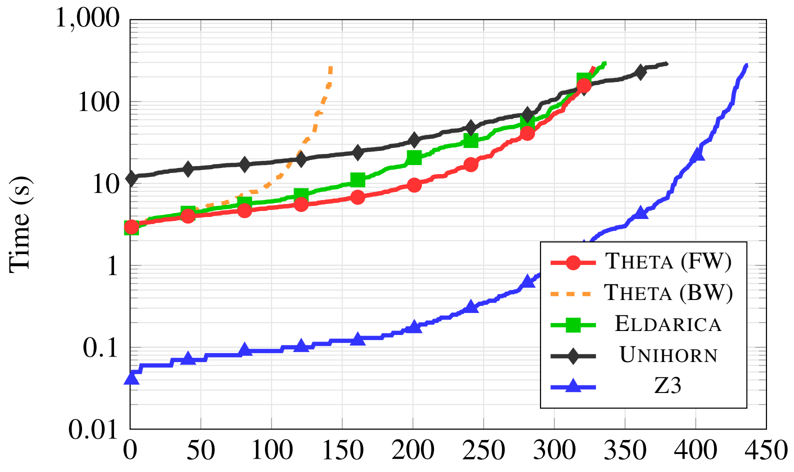


Figure 6: Number of solved tasks by tools under a certain time.

Results Table 1 shows the results of THETA with the different configuration options of THETA [11]. The results of the tool were either correct or timeout for all of the tasks.

Forward transformation proved to be far more effective than backward transformation in all configurations. The configurations using boolean predicate based abstraction with sub-state splitting (`PRED_SPLIT`) performed the best, with the other predicate based abstraction methods not too far behind.

The same benchmarks were also run with the top solvers of the LIA-Lin track from CHC-COMP21 [7], namely Z3, UNIHORN and ELDARICA. These solvers were run using their default configuration and with the same constraints as THETA. Table 2 shows the number of solved tasks compared to the best-performing configuration of THETA. Although THETA performs worse than the other solvers, its performance is comparable to ELDARICA's.

A quantile plot of the tools' performances can be seen on Figure 6. THETA performs better than both UNIHORN and ELDARICA for easier tasks, but it starts to get slower at a faster pace for tougher tasks than the other tools.

Conclusion As shown in Table 1, the performance of THETA was greatly improved by the forward transformation process for all of the tested configurations. This improvement gains even more significance when compared to other tools: just by changing the transformation method, THETA becomes a relevant competitor for some of the best linear CHC solvers of CHC-COMP. Based on our findings, we propose that tools employing software verification techniques for CHC solving implement our novel approach, to potentially significantly increase the number of successfully solved CHC problems.

Acknowledgement. We want to thank Martin Blicha for the help and fruitful discussion regarding the topic.

Funding. This research was partially funded by the ÚNKP-22-{2,3}-I New National Excellence Program and the 2019-1.3.1-KK-2019-00004 project from the National Research, Development and Innovation Fund of Hungary.

References

[1] Emanuele De Angelis & Hari Govind V K (2022): *CHC-COMP 2022: Competition Report*. *Electronic Proceedings in Theoretical Computer Science* 373, pp. 44–62, doi:10.4204/eptcs.373.5.

[2] Levente Bajczi, Zsófia Ádám & Vince Molnár (2022): *C for Yourself: Comparison of Front-End Techniques for Formal Verification*. In: *2022 IEEE/ACM 10th International Conference on Formal Methods in Software Engineering (FormaliSE)*, doi:10.1145/3524482.3527646.

[3] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs & K. Rustan M. Leino (2006): *Boogie: A Modular Reusable Verifier for Object-Oriented Programs*. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf & Willem-Paul de Roever, editors: *Formal Methods for Components and Objects*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 364–387, doi:10.1007/11804192_17.

[4] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu & Helmut Veith (2003): *Counterexample-Guided Abstraction Refinement for Symbolic Model Checking*. *J. ACM* 50(5), p. 752–794, doi:10.1145/876638.876643.

[5] Edmund M. Clarke, William Klieber, Miloš Nováček et al. (2012): *Model checking and the state explosion problem*. *Lecture Notes in Computer Science*, p. 1–30, doi:10.1007/978-3-642-35746-6_1.

[6] Zafer Esen & Philipp Ruemmer (2022): *TRICERA: Verifying C Programs Using the Theory of Heaps*. In: *CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN–FMCAD 2022*, TU Wien Academic Press, pp. 360–391, doi:10.34727/2022/isbn.978-3-85448-053-2_45.

[7] Grigory Fedyukovich & Philipp Rümmer (2021): *Competition Report: CHC-COMP-21*. In Hossein Hojjat & Bishoksan Kafle, editors: *Proceedings 8th Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2021, Virtual, 28th March 2021, EPTCS* 344, pp. 91–108, doi:10.4204/EPTCS.344.7.

[8] Orna Grumberg, Doron A Peled & EM Clarke (1999): *Model checking*. MIT press Cambridge.

[9] Arie Gurfinkel (2022): *Program Verification with Constrained Horn Clauses (Invited Paper)*. In Sharon Shoham & Yakir Vizel, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 19–29, doi:10.1007/978-3-031-13185-1_2.

[10] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli & Jorge A. Navas (2015): *The SeaHorn Verification Framework*. In Daniel Kroening & Corina S. Păsăreanu, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 343–361, doi:10.1007/978-3-319-21690-4_20.

[11] Ákos Hajdu & Zoltán Micskei (2020): *Efficient Strategies for CEGAR-Based Model Checking*. *Journal of Automated Reasoning* 64(6), pp. 1051–1091, doi:10.1007/s10817-019-09535-x.

[12] Matthias Heizmann, Jochen Hoenicke & Andreas Podelski (2009): *Refinement of Trace Abstraction*. In Jens Palsberg & Zhendong Su, editors: *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, *Lecture Notes in Computer Science* 5673, Springer, pp. 69–85, doi:10.1007/978-3-642-03237-0_7.

[13] Hossein Hojjat & Philipp Rümmer (2018): *The ELDARICA Horn Solver*. In: *2018 Formal Methods in Computer Aided Design (FMCAD)*, pp. 1–7, doi:10.23919/FMCAD.2018.8603013.

[14] Yusuke Matsushita, Takeshi Tsukada & Naoki Kobayashi (2021): *RustHorn: CHC-Based Verification for Rust Programs*. *ACM Trans. Program. Lang. Syst.* 43(4), doi:10.1145/3462205.

[15] Leonardo 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*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[16] Terence Parr (2013): *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, doi:10.5555/2501720.

[17] A. M. Turing (1937): *On Computable Numbers, with an Application to the Entscheidungsproblem*. *Proceedings of the London Mathematical Society* s2-42(1), pp. 230–265, doi:10.1112/plms/s2-42.1.230.

[18] Jeffrey D. Ullman (1989): *Bottom-Up Beats Top-Down for Datalog*. In Avi Silberschatz, editor: *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania, USA*, ACM Press, pp. 140–149, doi:10.1145/73721.73736.

An Encoding for CLP Problems in SMT-LIB

Daneshvar Amrollahi
University of Tehran
d.amrollahi@ut.ac.ir

Hossein Hojjat
TeIAS, Khatam University
University of Tehran
hojjat@ut.ac.ir

Philipp Rümmer
University of Regensburg
Uppsala University
philipp.ruemmer@ur.de

The input language for today's CHC solvers are commonly the standard SMT-LIB format, borrowed from SMT solvers, and the Prolog format that stems from Constraint-Logic Programming (CLP). This paper presents a new front-end of the Eldarica CHC solver that allows inputs in the Prolog language. We give a formal translation of a subset of Prolog into the SMT-LIB commands. Our initial experiments show the effectiveness of the approach and the potential benefits to both the CHC solving and CLP communities.

1 Introduction

Over the last years, a growing number of solvers for Constrained Horn Clauses (CHC) have been developed; for instance, Spacer [8], Eldarica [5], Golem [1], and RInGEN [2]. There are two main languages used to interface such solvers: the SMT-LIB language [3], in which Horn clauses can either be expressed using quantified assertions, or using the rule-based notation that was introduced by Z3; and dialects of Prolog [6] as a language in Constraint-Logic Programming (CLP). The former language is designed primarily for machine-generated input to solvers, as it is simple to parse, strongly typed, and has unambiguous semantics. The latter language is more concise and convenient for handwritten programs. There is, unfortunately, no exact match between the theories considered in CLP and SMT-LIB, and some Prolog language features have semantics that are not straightforward to model; for instance, the default absence of the "occurs-check" in equations.

This paper presents an ongoing effort to add a comprehensive Prolog/CLP front-end to the CHC solver Eldarica [5]. Eldarica has been able to process clauses in Prolog format since the beginning of its development; however, the existing Prolog front-end of Eldarica is restricted to the parsing of clauses over integers, and it is planned to replace it with a front-end with more extensive support for the different Prolog features through this project. To document the applied interpretation of Prolog, we define a formal translation of a fragment of Prolog to the SMT-LIB language. In the scope of this paper, we focus on three key features of Prolog: functions, defining a Herbrand universe of values; lists; and integer numbers as introduced by CLP($\mathbb{Z}$). We model the semantics of those language features using a mapping to SMT-LIB data-types (i.e., algebraic data-types with free constructors) and SMT-LIB integers.

Example To illustrate the complementarity of CHC and CLP, we start with a simple routing example including integer arithmetic, lists, and functions in Prolog. Figure 1 shows a CLP($\mathbb{Z}$) program to compute the distance between cities. The program consists of 9 facts and 2 rules. A fact of the form `distance(`X`, `Y`, `Z`)` states that the direct distance between cities X and Y is Z. The rule in line 12 implies that distance is symmetric. The meaning of `path(`X`, `Y`, `Z`, `L`)` is that there exists a path of length Z from X to Y where L represents the path as a list of points in the format `waypoint(`C`, `D`)`. This shows that city C lies on the path from X to Y with a distance of D from the starting point, which is X. The fact `path(A, A, 0, [waypoint(A, 0)]).` states that for any city `A`, there is a path of length 0 to itself and the list

```
1  :- use_module(library(clpz)).  % or clpfd
2
3  distance(tehran,   vienna, 31).
4  distance(vienna,   paris,  10).
5  distance(vienna,   munich, 3).
6  distance(paris,    munich, 10).
7  distance(paris,    rome,   11).
8  distance(lausanne, rome,   6).
9  distance(lausanne, munich, 4).
10 distance(tehran,   paris,  42).
11
12 distance(A, B, D) :- distance(B, A, D).
13
14 path(A, A, 0, [waypoint(A, 0)]).
15 path(A, C, D, [waypoint(C, D) | N]) :- path(A, B, P, N), distance(B, C, Q),
16                                         D #= P + Q.
17
18 ?- path(tehran, munich, D, X), D #< 40.
```

Figure 1: Prolog Program for distance between cities in CLP($\mathbb{Z}$)

presenting the path is the city `A` itself. Finally, the rule on line 15 implies that if there is path `N` of length `P` from `A` to `B`, and the direct distance from `B` to `C` is `Q` and `D` is equal to `P + Q`, then we have found a path from `A` to `C`, which is the previously found path (`N`) extended with the city `C`. The query `path(tehran, munich, D, X), D #< 40.` searches for a path from `munich` to `tehran` of length less than 40. A possible answer to this query is `D = 34` and `X = [waypoint(munich,34), waypoint(vienna,31), waypoint(tehran,0)]`. This means that there is a path from `tehran` to `munich` of distance 34, with `vienna` being on the way with distance `31` from `tehran`.

Note that this Prolog program, while intuitive, is also rather inefficient, and interpreting it using a CLP engine might lead to non-termination due the recursion on lines 12,15. The program can be rewritten to a more operationally oriented (and harder to read) set of clauses to be terminating and efficient, in particular preventing cyclic paths from being explored, and inlining the length bound to allow branches without solutions to be pruned. Alternatively, tabling [4] could be used for the predicate `path`[1].

CHC solvers are generally less efficient than Prolog engines for solving constraint satisfaction problems. However, the abstraction techniques in CHC solvers are naturally able to cope with clauses written in declarative and otherwise inefficient style. They can easily find a path from `tehran` to `munich` of length `40` in Figure 1. CHC solvers are also able to determine that no such path exists for lengths less than `34`, a task more challenging for at least some CLP solvers. More generally, CHC solvers are agnostic of the order of clauses, and the order of literals in clauses, and process clauses focusing more on their logical content than syntactic features. In this sense, CHC solvers can be a useful debugging tool, and have a complementary performance profile to CLP systems. The goal of our work is to simplify the integration of CLP and CHC methods, by providing a translation of a subset of Prolog into SMT-LIB, the common input language of CHC solvers.

Organization of the paper: In Section 2 we overview the CLP and the SMT-LIB input languages. Section 3 discusses translation rules for converting Prolog to SMT-LIB. Section 4 translates the introductory example to SMT-LIB. We conclude the paper and present directions of future research in Section 5.

[1]Which, however, led to an error in experiments with SWI-Prolog [11].

⟨*Database*⟩ ::= ⟨*Clause*⟩*

⟨*Clause*⟩ ::= ⟨*Predicate*⟩ ‘`.`’
  | ⟨*Predicate*⟩ ‘`:-`’ ⟨*BodyItem*⟩* ‘`.`’
  | ‘`?-`’ ⟨*BodyItem*⟩* ‘`.`’

⟨*BodyItem*⟩ ::= ⟨*Predicate*⟩
  | ⟨*Constraint*⟩

⟨*Predicate*⟩ ::= ⟨*Atom*⟩
  | ⟨*Atom*⟩ ‘`(`’ ⟨*Term*⟩* ‘`)`’

⟨*Term*⟩ ::= ⟨*Variable*⟩
  | ⟨*Atom*⟩
  | ⟨*Atom*⟩ ‘`(`’ ⟨*Term*⟩* ‘`)`’
  | ⟨*List*⟩
  | ⟨*Integer*⟩

⟨*List*⟩ ::= ‘`[`’ ⟨*Term*⟩* ‘`]`’
  | ‘`[`’ ⟨*Term*⟩* ‘`|`’ ⟨*Term*⟩ ‘`]`’

Figure 2: A simplified grammar for describing the syntax of Prolog programs.

2 Preliminaries

2.1 Constraint Logic Programming (CLP)

Constraint Logic Programming (CLP) [10], first introduced by Jaffar and Lassez in 1987 [7], is a programming paradigm that combines the benefits of constraint programming and logic programming. It enables the modeling of complicated real-world problems with variables and constraints, and uses logical and constraint reasoning to find a solution. CLP expresses the connections between variables as constraints and looks for a derivation of queries from given clauses.

Prolog is the main language of CLP, where the constraints are equations over the algebra of terms. CLP problems may in addition contain symbols with pre-defined meanings, defined by a *theory*. CLP($\mathbb{Z}$) [9] refers to CLP over the theory of integer arithmetic. Figure 2 is a simplified grammar that is able to parse Prolog programs with functions, lists, and integer arithmetic. An extended version of it is used in our actual implementation.

As shown in Figure 2, every Prolog program consists of a set of *Clause*s. A clause has a *head* and a *body*. The head can have zero or one predicates, and the body is a list of predicates and constraints. A clause is either a *Fact*, a *Rule*, or a *Query*. For example, `man(tom)` is a fact, and `friends(X, Y) :- likes(X, Y), likes(Y, X)` is a rule with `friends(X, Y)` as the head and `likes(X, Y), likes(Y, X)` as the body. Queries always start with a `?-`. For instance, `?- likes(X, tom)` would search for an assignment for X such that `likes(X, tom)` can be derived from the set of given clauses. *Term*s can be variables, atoms, compound terms, lists, and integers.

Structured data is represented using compound terms. A compound term consists of a function and a sequence of one or more sub-terms. A function is characterized by its name, which is an atom, and its arity. For instance, in the fact `man(father(claire))`, the term `father(claire)` is a compound term.

In Prolog, there are different kinds of built-in equality operators (`#=, =, =:=, is`) with different semantics. The `=/2` predicate, which accepts two arguments, is used to ensure that its two arguments are syntactically equal through unification. For instance, the response to the query `?- father(X) = father(john)` will be positive as the sub-terms can be unified by setting `X = john`. By default, Prolog does not apply any “occurs-check”, however, which can sometimes cause non-termination. An example of this is `?- X = father(X)`. The unification algorithm decides to unify `X` with the right-hand-side, which is `father(X)`. But still there is an `X` left in this term; interpretation can get stuck in this loop of replacing `X` by `father(X)` and never terminate.

*List*s are terms representing sequences of elements; for instance, `[a, [b, c], 7]` is a list of terms. A non-empty list can be also be thought of as having two parts: *head* as the first element of the list, and *tail* as the remainder of the list. Both representations can be parsed using the grammar rules in Figure 2.

The body of the clauses in a Prolog program may also include *constraint*s, in addition to predicates. An example of a constraint in CLP($\mathbb{Z}$), Constraint Logic Programming over the domain of integers, is `X + Y #>= 3 * Z - 17`. Some other operators for constraints in CLP($\mathbb{Z}$) include `#=, #>, #>=, #<, #=<, +, -, *, /, mod`. Throughout the rest of this paper, whenever we refer to constraints, we refer to constraints in CLP($\mathbb{Z}$).

2.2 SMT-LIB

SMT-LIB, or the Satisfiability Modulo Theories Library [3], is a standardized format for specifying logical formulas modulo various background theories. It is widely used in applications like software verification, hardware design, and automated reasoning, and is also used as a standard input format of CHC solvers. The SMT-LIB language supports a range of theories, including arithmetic, bit-vectors, and arrays, allowing users to express a wide variety of constraints.

An SMT-LIB script consists of a list of commands to be processed by an SMT solver. Important SMT-LIB commands are:

- `(set-logic` L`)`: Set the logic, i.e., the combination of theories that will be used. For CHC solvers, typically $L =$ `HORN`.
- `(declare-fun` f `(`T_1 `...` T_n`)` T`)`: Declaration of a function or predicate f with the given argument types $T_1 \dots T_n$ and result type T. Types in SMT-LIB include the types provided by background theories (e.g., `Int` for integers, `Bool` for Booleans), as well as user-defined types. When interfacing CHC solvers, typically only predicates (i.e., Boolean-valued functions) are declared.
- `(declare-datatype` T `(`C_1 `...` C_n`))`: Declaration of an algebraic data-type T with constructors $C_1 \dots C_n$. Data-types can be recursive, and are the main modelling technique in SMT-LIB to represent structured data. For every constructor C, the data-type declaration will also introduce a tester `(_ is` C`)` to identify terms constructed using C, as well as selectors for the arguments. An extended version of the command, `declare-datatypes`, exists to define several mutually recursive data-types. We will see various examples of algebraic data-types in this article.
- `(assert` ϕ`)`: Assert a constraint ϕ, which can be formulated using the standard operators of first-order logic, the functions and predicates provided by background theories, and declared symbols. CHC solvers require that asserted constraints ϕ are *constrained Horn clauses,* which means that they fall into one of the following classes of formulas:

Facts:	`(`$p\ t_1 \dots t_k$`)`
Quantified facts:	`(forall (`$X_1 \dots X_n$`) (`$p\ t_1 \dots t_k$`))`
Rules:	`(forall (`$X_1 \dots X_n$`) (=> (and` $\psi_1 \dots \psi_m$`) (`$p\ t_1 \dots t_k$`)))`
Queries:	`(forall (`$X_1 \dots X_n$`) (=> (and` $\psi_1 \dots \psi_m$`) false)`

- `(check-sat)`: Instruct the SMT solver to check whether the constraints asserted up to this point are satisfiable.

Throughout the paper, only a small subset of the SMT-LIB commands are used. For a full definition of SMT-LIB, we refer to the standard documentation [3].

```
(declare-datatype U (          (declare-fun man      (assert
    (claire)                       (U)                   (man (father claire))
    (father (father_1 U))          Bool              )
    )                          )
)
```

Figure 3: SMT-LIB equivalent of `man(father(claire))`

3 Translating CLP to SMT-LIB

In this section, we explain our translation from CLP problems in Prolog to SMT-LIB. We begin by a motivating example as a warm-up. After that, we present the translation rules for Prolog facts, rules, lists, and CLP($\mathbb{Z}$) constraints. We remark that our implementation does not explicitly translate input problems to SMT-LIB, but instead directly constructs clauses using Eldarica's internal data structures; the translation to SMT-LIB is presented for documentation purposes, and reflects the semantics that is applied.

3.1 A Motivating Example

Consider the Prolog fact `man(father(claire))`. This fact consists of the function `father` applied to an atomic term `claire`. To model this term, one could consider declaring corresponding uninterpreted functions in SMT-LIB. However, with uninterpreted functions, `claire` and `father(claire)` might be assigned the same value, despite being syntactically different terms: the equation `(= claire (father claire))` is satisfiable in SMT-LIB. This is inconsistent with Prolog semantics, in which `claire` and `father(claire)` are different and non-unifiable terms, and it is guaranteed that they stand for distinct elements of the Herbrand universe. In addition, most CHC solvers do not support uninterpreted functions.

We therefore propose to treat `claire` and `father` as constructors of an algebraic data-type. To translate the mentioned fact to SMT-LIB, the first step is to introduce a new algebraic data-type for all of the terms occurring in a program. The data-type will be called `U`, standing for *U*niversal. The constructors of `U` will be all atoms and functions appearing in the clauses. Figure 3 shows the full translation of the example. The constructor `father` is unary, and its definition also adds a selector `father_1` to retrieve the sub-term. We also define the predicate `man`, as a Boolean function with a single argument of type `U`. Finally, we add the fact `man(father(claire))` to the set of our clauses using the `assert` command. Section 3.2 provides further details.

It has to be noted that our encoding of terms does not exactly model Prolog semantics. As explained in Section 2, the unification algorithm of Prolog will not detect the non-unifiability of a query `?- X = father(X)` due to the missing occurs-check. In contrast, elements of SMT-LIB data-types are finite constructor terms, which implies that the equation `(= X (father X))` is not satisfiable in SMT-LIB. However, we believe that the finite-term semantics is usually the intended semantics of a Prolog program. It remains to be investigated in future work whether there is a way to represent Prolog semantics more closely in SMT-LIB.

3.2 Algebraic Data Types (ADTs)

We define a single algebraic data-type `U` for representing the type of all terms in a program. The constructors of `U` will be all functions and atoms appearing in the clauses. As this global type `U` is defined

$$
collectFunctions(t) = \begin{cases} \{(atom, 0)\}, & \text{if } t = atom \\ \{(c,n)\} \cup \bigcup_{i=1}^{i=n} collectFunctions(a_i), & \text{if } t = c(a_1, \cdots, a_n) \\ \emptyset, & \text{otherwise} \end{cases}
$$

Figure 4: Function *collectFunctions* collecting functions in the terms

```
(declare-datatype U
(
    (f_1 (f_1^1 U) ... (f_1^{a_1} U))
    ...
    (f_n (f_n^1 U) ... (f_n^{a_n} U))
)
```

Figure 5: Definition of the universal type U for functions $\{(f_1,a_1),\dots,(f_n,a_n)\}$

only once, we need to iterate through all clauses and collect the functions and atoms appearing in them. We define a recursive function responsible for this matter in Figure 4. This function is applied to a term t, and recursively collects the atoms and functions occurring in t, as well as their arity. For instance, the value of $collectFunctions(a(X,b,c(d,Y,e)))$ is $\{(e,0),(d,0),(c,3),(b,0),(a,3)\}$.

Given that the set collected using *collectFunctions* is $\{(f_1,a_1),\dots,(f_n,a_n)\}$, the universal algebraic data type U should be defined as in Figure 5.

3.3 Translation of Facts

Section 3.1 shows an example of translating a fact to SMT-LIB. We now introduce the general case of this translation, and add rules for lists and integer arithmetic in Sections 3.5 and 3.6. We refer to triplets of the form $\texttt{s} \rhd \texttt{s}' : \Phi$ as *translation judgements*. The meaning of a judgement is that the Prolog term s can be translated to an SMT-LIB term s′ under a set of side conditions Φ. Side conditions are mainly needed to capture typing requirements. For instance, Prolog (CLP($\mathbb{Z}$)) raises an error when encountering

$$\frac{}{a \rhd a : \emptyset}\ \text{a is an atom} \qquad \frac{}{V \rhd V : \emptyset}\ \text{V is a variable}$$

$$\frac{\{t_i \rhd t_i' : \Phi_i\}_{i=1}^{n}}{\texttt{f}(t_1,\dots,t_n) \rhd (\texttt{f}\ t_1'\ \dots\ t_n') : \bigcup_{i=1}^{n} \Phi_i}\ \texttt{f} \text{ is a function or predicate}$$

$$\frac{s \rhd s' : \Phi_1 \quad t \rhd t' : \Phi_2}{s\ \texttt{=}\ t \rhd (\texttt{=}\ s'\ t') : \Phi_1 \cup \Phi_2} \qquad \frac{s \rhd s' : \Phi}{\texttt{\textbackslash+}s \rhd (\texttt{not}\ s') : \Phi}$$

$$\frac{s \rhd s' : \Phi_1 \quad t \rhd t' : \Phi_2}{s\ \texttt{=\textbackslash=}\ t \rhd (\texttt{not}\ (\texttt{=}\ s'\ t')) : \Phi_1 \cup \Phi_2}$$

Figure 6: Basic rules for translating Prolog to SMT-LIB

```
(declare-fun p
    (U ... U)
    Bool
)
```

```
(assert
    (forall ( (X_1 U) ... (X_m U) )
        (=> (and Φ_1 ... Φ_n) (p t'_1 ... t'_n))
    )
)
```

Figure 7: Translation of a fact $p(t_1,\dots,t_n)$. We assume that the elements of a set Φ_i are implicitly conjoined.

```
(declare-fun likes
    (U U)
    Bool
)

(declare-fun friends
    (U U)
    Bool
)
```

```
(assert
    (forall ( (X U) (Y U) )
        (=>
            (and (likes X Y) (likes Y X))
            (friends X Y)
        )
    )
)
```

Figure 8: Translation of a rule in SMT-LIB

the expression `a + b`, where `a` or `b` are not integers. Thus, when translating `a + b` to SMT-LIB, we will later add side conditions that ensure the correct type of `a` and `b`. More details concerning the use of side conditions are given in Sections 3.5 and 3.6.

Figure 6 shows the basic translation rules. Each rule derives a translation judgement, the conclusion under the bar, from premises shown above the bar. The rules in Figure 6 are mostly self-explanatory, and recursively translate a given term or constraint to SMT-LIB. The rules assume, for sake of presentation, that atoms, variables, and functions are translated to SMT-LIB symbols with the same name.

The SMT-LIB translation of a Prolog fact $p(t_1,\dots,t_n)$ is shown in Figure 7 in terms of expressions $t'_1,\dots,t'_n$ and side conditions $\Phi_1,\dots,\Phi_n$, which are defined as follows:

- For all $i \in \{1,\dots,n\}$ it holds that $t_i \triangleright t'_i \;:\; \Phi_i$.
- $X_1,\dots,X_m$ are all the variables appearing in the terms $t_1,\dots,t_n$.

The fact is captured using an SMT-LIB assertion in Figure 7, listing the side conditions required by the translation as assumptions.

3.4 Translation of Rules

Before presenting the translation of rules, we begin with a concrete example, using the Prolog rule `friends(X, Y) :- likes(X, Y), likes(Y, X).` This rule will be turned into a universally quanti-

```
(assert
    (forall ( (X_1 U) ... (X_k U) )
        (=> (and ψ'_1 ... ψ'_m Φ_0 ... Φ_m) ψ'_0)
    )
)
```

Figure 9: Translation of a rule ψ_0 `:-` $\psi_1,\dots,\psi_m$

```
(declare-datatypes () (
        (U
            (aList (theList L))
            ...
        )
        (L
            nil
            (cons (head U) (tail L))
        )
    )
)
```

Figure 10: Algebraic data-types including lists

```
list_concat([],L,L).
list_concat([X1|L1],L2,[X1|L3]) :- list_concat(L1,L2,L3).
```

Figure 11: A Prolog program concatenating lists

fied implication in SMT-LIB. Given that we have already defined our universal data-type `U` as explained in Section 3.2, all we need is to declare functions in SMT-LIB for `likes` and `friends`, and create the rule using the universal quantifier. The full translation is illustrated in Figure 8.

The general schema for expressing a rule ψ_0 :- $\psi_1, \ldots, \psi_m$ in SMT-LIB is shown in Figure 9, assuming that:

- For all $i \in \{0,\ldots,m\}$ it holds that $\psi_i \triangleright \psi_i' : \Phi_i$.
- $X_1,\ldots,X_k$ are all the variables appearing in the rule.

The assertion representing the rule in Figure 9 has a similar shape as the assertion for facts in Figure 7.

3.5 Translation of Lists

We now add lists to the picture. The absence of static typing in Prolog makes it a-priori impossible to tell whether a Prolog variable will represent a list or a term constructed using some function (or possibly both). We therefore include lists as one case in our universal data-type `U`, and ensure the correct typing of terms through side-conditions in our translation rules.

```
(declare-fun list_concat (U U U) Bool)

(assert (forall ((X U)) (list_concat (aList nil) X X)))

(assert (forall ((X1 U) (L1 U) (L2 U) (L3 U))
    (=> (and (list_concat L1 L2 L3) ((_ is aList) L1) ((_ is aList) L3))
        (list_concat
          (aList (cons X1 (theList L1))) L2 (aList (cons X1 (theList L3))))))
)
```

Figure 12: SMT-LIB representation of the program in Figure 11

$$\frac{}{\texttt{[]} \triangleright \texttt{(aList nil)} : \emptyset}$$

$$\frac{\{t_i \triangleright t_i' \,:\, \Phi_i\}_{i=1}^{n}}{\texttt{[}t_1\texttt{,}\ldots\texttt{,}t_n\texttt{]} \;\triangleright\; \texttt{(aList (cons } t_1' \texttt{ (cons } t_2' \texttt{ (... cons } t_n' \texttt{ nil))...))} \triangleright \bigcup_{i=1}^{n} \Phi_i}$$

$$\frac{h \triangleright h' \triangleright \Phi_1 \quad t \triangleright t' \triangleright \Phi_2}{\texttt{[}h\texttt{|}t\texttt{]} \triangleright \texttt{(aList (cons } h' \texttt{ (theList } t'\texttt{)))} \;:\; \Phi_1 \cup \Phi_2 \cup \{\texttt{((\_ is aList) } t'\texttt{)}\}}$$

Figure 13: Translation rules for lists

```
(declare-datatypes () (
        (U
            (anInt (theInt Int))
            ...
        )
    )
)
```

Figure 14: Modified declaration of the universal ADT for integer arithmetic

Lists can be viewed as an algebraic data-type (call it L) with two constructors nil and cons. The constructor nil has zero arguments and represents empty lists, and cons has two arguments: one referring to the head of the list (a term of type U), and one of type L referring to the tail of the list.

To implement lists in SMT-LIB, it is therefore natural to declare a new data-type L with the nil and cons constructors. This new data-type will have a mutual dependency on our universal data-type U, as defined before, and therefore has to be declared along with U as shown Figure 10. Note that in Figure 10, there may be other constructors for U coming from the collected functions in Section 3.2, denoted by three dots. The aList constructor is introduced to wrap terms of type L as a term of the universal type U. An example of a Prolog program involving lists is presented in Figure 11. The translation of the program in Figure 11 to SMT-LIB is presented in Figure 12.

A set of inference rules for translating lists in Prolog to SMT-LIB is presented in Figure 13. The first rule in Figure 13 translates the empty list. The second rule represents the translation of a list with explicitly enumerated elements. In this case, the list can be constructed in SMT-LIB as a simple iterated application of cons. The final rule directly matches the functional definition of a list. A list in Prolog with h as the head, and t as the tail, can be constructed using the cons constructor in SMT-LIB. This construction is type-correct only if t is again a list; thus, the rule adds the side condition ((_ is aList) t'), and unwraps the tail using the theList selector. In all the rules in Figure 13, the result is finally wrapped using the aList constructor and turned into a U-term.

3.6 Integer Arithmetic

We finally introduce inference rules for converting constraints in the background theory of integer arithmetic (e.g., 2*X + 7 #> Y) to their corresponding expressions in SMT-LIB. To represent integers, sim-

$$\frac{s \rhd s' : \Phi_1 \quad t \rhd t' : \Phi_2}{s \star t \rhd \texttt{(anInt (}op\texttt{ (theInt }s'\texttt{) (theInt }t'\texttt{)))} : \Phi_1 \cup \Phi_2 \cup \{\texttt{(\_ is anInt) }s'\texttt{)}, \texttt{(\_ is anInt) }t'\texttt{)}\}}$$

where $(\star, op) \in \{$`(+,+), (-,-),(*,*),(/,div),(mod,mod)`$\}$

$$\frac{s \rhd s' : \Phi_1 \quad t \rhd t' : \Phi_2}{s \circ t \rhd \texttt{(}op\texttt{ (theInt }s'\texttt{) (theInt }t'\texttt{))} : \Phi_1 \cup \Phi_2 \cup \{\texttt{(\_ is anInt) }s'\texttt{)}, \texttt{(\_ is anInt) }t'\texttt{)}\}}$$

where $(\circ, op) \in \{$`(#=,=), (#>,>),(#>=,>=),(#<,<),(#=<,<=)`$\}$

$$\frac{}{I \rhd \texttt{(anInt }I\texttt{)} : \emptyset} \quad \text{I is a decimal non-negative integer}$$

$$\frac{}{\texttt{-}I \rhd \texttt{(anInt (- }I\texttt{))} : \emptyset} \quad \text{I is a decimal non-negative integer}$$

Figure 15: Translation rules for expressions and contraints in CLP($\mathbb{Z}$)

ilarly as with lists we add a new constructor to `U`, so that we can wrap integers into a term of type `U`. We call this new constructor `anInt`, and define it in Figure 14. We once again note that `U` is a universal type and may include other constructors.

The translation rules for integer expressions and Boolean constraints over them are presented in Figure 15. The first rule translates integer-valued functions by recursively translating the sub-terms, unwrapping the result, and applying the corresponding SMT-LIB function. The result is finally wrapped again using `anInt`. The second rule performs the corresponding translation for integer predicates. The other two rules take care of the translation of integer literals.

The translation can be modified easily to model bounded integers, which are used by some Prolog implementations. In this case, instead of `Int` in Figure 14, a bit-vector type like `(_ BitVec 64)` has to be used, and in Figure 15 the corresponding bit-vector operations have to be applied. Bit-vector support in CHC solvers is much less mature than support for mathematical integers, however.

As an example, consider the expression `X #> 7`. According to the translation rule for variables in Figure 6, `X` $\rhd X : \emptyset$, and according to the translation rule for integers in Figure 15, $7 \rhd$ `(anInt 7)` $: \emptyset$. Finally, according to the first rule in Figure 15 for translating expressions in integer arithmetic, the whole expression gets translated to `(anInt (+ (theInt X) (theInt (anInt 7))))`, with the side constraint that `X` is an integer.

4 An SMT-LIB Encoding of the Motivating Example

Combining the different Prolog features that were discussed, Figure 16 gives the SMT-LIB encoding of the Prolog program for computing paths between cities from Section 1. All the commands used in this encoding can be obtained using the translation rules explained throughout this article. The encoding begins by declaring the universal data-type `U`. Its constructors include all the atoms and functions that appear in the clauses, in addition to `anInt` and `aList`. After that, the data-type `L` is declared for lists, as explained in Section 3.5. The SMT-LIB script continues by declaring the functions `distance` and `path` appearing in the Prolog program clauses. The rest of the encoding are assertions corresponding to the Prolog rules and facts, and finally the assertion as a clause with head `false`.

A CHC solver like Eldarica [5] can derive the status `unsat` for this SMT-LIB script, which implies that the clauses are contradictory. This can be interpreted as the negation of the last clause being derivable from the other clauses. A CHC solver could also discover the path discussed in Section 1.

When tightening the length bound to `(< (theInt D) 34)`, the problem becomes satisfiable, which can again be verified using a CHC solver.

5 Conclusion

We have presented work towards a new CHC solver front-end that allows input in Prolog format, bridging the gap between Prolog/CLP semantics and SMT-LIB. We are in the process of implementing the defined translation from Prolog to SMT-LIB in our Horn solver Eldarica [5], with the goal of achieving good coverage of the Prolog and CLP features.

The translation defined in this paper should be seen as a starting point, as there are several aspects that require further work, more research, or more discussion in the communities:

- We have only shown the translation rules for some of the most important CLP($\mathbb{Z}$) operators. We believe that many other theories and constraints can be handled in a similar fashion.
- We keep typing constraints dynamic, and this way stay close to the actual Prolog semantics. In terms of the efficiency of CHC solvers on the translated program, of course, it could be beneficial to perform some amount of type inference upfront. This way, one could translate integer variables in Prolog to native SMT-LIB `Int` variables, etc. However, CHC solvers with support for algebraic data-types tend to perform such type inference themselves, so that the payoff from being more clever during the translation is unclear.
- Our translation includes all typing constraints as assumptions (Figures 7 and 9), i.e., the well-typedness of a Prolog program is assumed but not verified. It is not entirely obvious how correct typing should be asserted in the SMT-LIB representation, however. In the list example in Figure 11, for instance, note that the first clause implies that the second and third argument of `list_concat` can be terms of any kind, whereas the second clause relies on the third argument being a list. The clauses therefore entail ill-typed statements like `list_concat([X|[]], 42, [X|42]])`. A CLP engine performing backward chaining will, however, not run into any typing errors.
- There are several aspects of Prolog semantics that are challenging in a translation to SMT-LIB. Those include, in particular, the *missing occurs-check* in equations, as well as *cuts*. It is unclear whether those features can or should be translated faithfully to SMT-LIB semantics.

Finally, it will be interesting to evaluate solver performance for the different design choices in the translation, for different CHC solvers, and to compare to the performance of state-of-the-art CLP engines. We have not done such a comparison yet due to the preliminary state of the implementation of the front-end. While we generally assume CLP to be significantly more efficient on classical constraint satisfaction problems than CHC, there might also be areas in which the abstraction-based algorithms used in CHC solvers have advantages.

```
(declare-datatypes () (
        (U
            (anInt (theInt Int))
            (aList (theList L))
            (waypoint (waypoint_1 U) (waypoint_2 U))
            tehran vienna paris munich rome lausanne
        )
        (L
            nil
            (cons (head U) (tail L))
        )
    )
)
(declare-fun distance (U U U) Bool)
(declare-fun path (U U U U) Bool)

(assert (distance tehran   vienna (anInt 31)))
(assert (distance vienna   paris  (anInt 10)))
(assert (distance vienna   munich (anInt 3)))
(assert (distance paris    munich (anInt 10)))
(assert (distance paris    rome   (anInt 11)))
(assert (distance lausanne rome   (anInt 6)))
(assert (distance lausanne munich (anInt 4)))
(assert (distance tehran   paris  (anInt 42)))
(assert
    (forall ( (A U) (B U) (D U) )
        (=> (distance B A D) (distance A B D))
    )
)
(assert
    (forall ( (A U) )
        (path A A (anInt 0) (aList (cons (waypoint A (anInt 0)) nil)))
    )
)
(assert
    (forall ( (A U) (B U) (C U) (D U) (N U) (P U) (Q U) )
        (=>
            (and
                (path A B P N) (distance B C Q)
                (= D (anInt (+ (theInt P) (theInt Q))))
                ((_ is aList) N) ((_ is anInt) P) ((_ is anInt) Q)
            )
            (path A C D (aList (cons (waypoint C D) (theList N))))
        )
    )
)
(assert
    (forall ( (D U) (X U) )
        (=>
            (and (path tehran munich D X) (< (theInt D) 40) ((_ is anInt) D))
            false
        )
    )
)

(check-sat)
```

Figure 16: The SMT-LIB encoding of the motivating example introduced in the introduction

References

[1] *The Golem Horn Solver*. Available at `https://github.com/usi-verification-and-security/golem`.

[2] *RInGen: Regular Invariant Generator*. Available at `https://github.com/Columpio/RInGen`.

[3] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2017): *The SMT-LIB Standard: Version 2.6*. Technical Report, Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[4] W. Chen & D. S. Warren (1996): *Tabled Evaluation with Delaying for General Logic Programs*. *Journal of the ACM* 43(1), pp. 20–74, doi:10.1016/0304-3975(89)90088-1.

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

[6] ISO (1995): *ISO/IEC 13211-1:1995: Information technology — Programming languages — Prolog — Part 1: General core*. Available at `http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO%2FIEC+13211%2D1%3A1995;http://www.iso.ch/cate/d21413.html`.

[7] J. Jaffar & J.-L. Lassez (1987): *Constraint Logic Programming*. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, Association for Computing Machinery, New York, NY, USA, p. 111–119, doi:10.1145/41625.41635.

[8] Anvesh Komuravelli, Arie Gurfinkel & Sagar Chaki (2014): *SMT-Based Model Checking for Recursive Programs*. In Armin Biere & Roderick Bloem, editors: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, *Lecture Notes in Computer Science* 8559, Springer, pp. 17–34, doi:10.1007/978-3-319-08867-9_2.

[9] Markus Triska (2012): *The Finite Domain Constraint Solver of SWI-Prolog*. In Tom Schrijvers & Peter Thiemann, editors: *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings*, *Lecture Notes in Computer Science* 7294, Springer, pp. 307–316, doi:10.1007/978-3-642-29822-6_24.

[10] Mark Wallace (2002): *Constraint Logic Programming*, pp. 512–532. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/3-540-45628-7_19.

[11] Jan Wielemaker, Tom Schrijvers, Markus Triska & Torbjörn Lager (2010): *SWI-Prolog*. arXiv:1011.5332.