EPTCS 448

Proceedings of the

21st Workshop on

Logical Frameworks and Meta Languages: Theory and Practice

Lisbon, Portugal, 24th July 2026

Edited by: Sophie Tourret and Olivier Hermant

Published: 14th July 2026
DOI: 10.4204/EPTCS.448
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

Logical frameworks and meta-languages form a common substrate for representing, implementing and reasoning about a wide variety of deductive systems of interest in logic and computer science. Their design, implementation and their use in reasoning tasks, ranging from the correctness of software to the properties of formal systems, have been the focus of considerable research over the last three decades.

The LFMTP workshop brought together designers, implementors and practitioners to discuss various aspects impinging on the structure and utility of logical frameworks, including the treatment of variable binding, inductive and co-inductive reasoning techniques and the expressiveness and lucidity of the reasoning process.

The 2026 instance of LFMTP was organized by Olivier Hermant and Sophie Tourret in Lisbon, Portugal, on 24 July, as part of the 9th Federated Logic Conference. We are very grateful to the conference organizers for providing the infrastructure and local coordination for the workshop as well as to EPTCS for providing the logistics of publishing these proceedings.

The members of the programme committee were:

- Olivier Hermant (co-chair, Mines Paris PSL)
- Miguel Pagano (Universidad Nacional de Córdoba)
- Elaine Pimentel (UCL)
- Loïc Pujet (Stockholm University)
- Florian Rabe (FAU Erlangen-Nürnberg)
- Alvaro Tasistro (Universidad ORT Uruguay)
- Alwen Tiu (The Australian National University)
- Sophie Tourret (co-chair, INRIA and MPI for Informatics)
- Niccolò Veltri (Tallinn University of Technology)
- Yiming Xu (LMU Munich)

The editors are very grateful for their thorough analysis of all submissions.

The workshop received 8 submissions, of which 7 were accepted at the workshop. Of these, 5 were selected to be part of these proceedings among which one was a work-in-progress. The workshop also had one invited talk by:

- Alberto Momigliano (Università degli Studi di Milano)
  "It's the End of the World as We Know It, and I Feel Funny"

Olivier Hermant and Sophie Tourret
PC chairs of LFMTP 2026
July 24, 2026

It's the End of the World as We Know It (And I Feel Funny)

Alberto Momigliano
Dipartimento di Informatica, Università degli Studi di Milano, Italia

This is a tale in two parts. The first offers a reappraisal of the history and accomplishments of mechanized metatheory, from the first LFM workshop to the age of AI. In particular, we will examine the influence, if any, of benchmarks [2, 4, 8, 9, 1, 3] on our research area. It seems indisputable that they, and by they I mean mostly the original POPLMark Challenge, helped bridge the gap between the formal verification and the PL and systems community, making verification more mainstream. Arguably, they have had a more limited impact on the theory and practice of proof assistants, which have evolved following other dynamics: even "new" systems such as Abella and Beluga are based on ideas developed around the turn of the century. Conversely, the same period has seen the emergence of several libraries for reasoning about binders, one of POPLMark's central concerns: *Nominal, Hybrid, Metalib, LNGen, LN, Autosubst, Knot, and DBLib*. Here again, many of the underlying ideas predate the benchmark age. Partial exceptions include libraries such as *generic syntax, SOAS*, and *Tealeaves*, whose origins are more strongly category-theoretic.

All this work is put into perspective, if not altogether into question, by the rise of the machines. General and specialized LLMs are showing their value in mathematics and formal verification. This is illustrated, e.g., by the recently announced *Signal Shot* endeavor, which aims to verify the Signal protocol and its Rust implementation using Lean[1]. LLMs are also beginning to enter mechanized metatheory, see [12] for a recent experience report, with the caveat that the rapid improvement of frontier models makes such papers liable to become obsolete shortly after publication.

While it is still early, we cannot help asking what the point is of developing new techniques or libraries if we can simply throw tokens at the problem. Perhaps de Moura is correct when he says:

> *No technology lasts forever, and that is a sign of progress, not failure. The goal is not to build a system that endures indefinitely [...] If something genuinely superior emerges, that means the mission succeeded.* (ibidem)

Even as we debate whether, as the old Gauls feared, the sky is about to fall on mechanized metatheory in the shape of the AI apocalypse, we proceed from the same old Gaulish response: tomorrow never comes. In the second part of this talk, we thus turn to some recent developments in the encoding of *substructural* systems, which play an increasingly important role in the study of resource-sensitive computation across a wide range of applications, including concurrency theory, quantum computing, and memory management. The landscape for encoding their metatheory is far less settled than for structural systems. We highlight two approaches: the first is the rediscovery of the unusual effectiveness of Crary's "linear" predicate [6]: the idea is to encode linear type systems in a structural framework such as LF by decorating typing rules with a condition ensuring that assumptions are used linearly (or, for that matter, in an affine, strict, or even modal way). This idea has recently been applied to the metatheory of session types, both in an HOAS setting [14] and in a more traditional one [5], using well-scoped de Bruijn indices in Rocq. It integrates easily with relational reasoning, as required in normalization or equivalence proofs. In fact, the linear predicate makes it surprisingly direct to extend a Howe-style proof such as [11]

[1] `https://leodemoura.github.io/blog/2026-4-20-signal-shot-the-platform-is-ready/`

Hermant, Tourret (Eds.): International Workshop on Logical Frameworks and Meta-Languages, Theory and Practice (LFMTP 26)
EPTCS 448, 2026, pp. iii–iv, doi:10.4204/EPTCS.448.0.1

to the linear setting, bypassing most of the technicalities detailed in [7]. For a more exotic example, see [10].

We conclude by looking at ways to support the encoding of substructural systems in general, from the approach described in [15] to work in progress on a Rocq library inspired by adjoint logic [13].

References

[1] Andreas Abel et al. (2019): *POPLMark reloaded: Mechanizing proofs by logical relations*. *J. Funct. Program.* 29, p. e19, doi:10.1017/S0956796819000170.

[2] Brian E. Aydemir et al. (2005): *Mechanized Metatheory for the Masses: The PoplMark Challenge*. In Joe Hurd & Thomas F. Melham, editors: *TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, *Lecture Notes in Computer Science* 3603, Springer, pp. 50–65, doi:10.1007/11541868_4.

[3] Marco Carbone et al. (2024): *The Concurrent Calculi Formalisation Benchmark*. In Ilaria Castellani & Francesco Tiezzi, editors: *Coordination Models and Languages 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings*, *Lecture Notes in Computer Science* 14676, Springer, pp. 149–158, doi:10.1007/978-3-031-62697-5_9.

[4] Andrew W. Appel, Robert Dockins & Xavier Leroy (2012): *A List-Machine Benchmark for Mechanized Metatheory*. *J. Autom. Reason.* 49(3), pp. 453–491, doi:10.1007/S10817-011-9226-1.

[5] Marco Carbone & Alberto Momigliano (2026): *Mechanising the Meta-Theory of Session Types in Rocq: a Tutorial*. To appear in Behavioural Application Programming Interfaces (Springer).

[6] Karl Crary (2010): *Higher-order representation of substructural logics*. In: *Proc. ICFP '10*, pp. 131–142, doi:10.1145/1863543.1863565.

[7] Roy L. Crole (2001): *Completeness of Bisimilarity for Contextual Equivalence in Linear Theories*. *Log. J. IGPL* 9(1), pp. 27–51, doi:10.1093/JIGPAL/9.1.27.

[8] 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.

[9] 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.

[10] Ryan Kavanagh, Chuta Sano & Brigitte Pientka (2025): *Deconstructed Proto-Quipper: A Rational Reconstruction*. *CoRR* abs/2510.20018, doi:10.48550/ARXIV.2510.20018.

[11] Alberto Momigliano (2012): *A supposedly fun thing I may have to do again: a HOAS encoding of Howe's method*. In: *Proceedings of the Seventh International Workshop on Logical Frameworks and Meta-Languages, Theory and Practice*, LFMTP '12, Association for Computing Machinery, New York, NY, USA, p. 33–42, doi:10.1145/2364406.2364411.

[12] Zoe Paraskevopoulou (2026): *Machine-Generated, Machine-Checked Proofs for a Verified Compiler (Experience Report)*. *CoRR* abs/2602.20082, doi:10.48550/ARXIV.2602.20082.

[13] Klaas Pruiksma (2024): *Adjoint Logic with Applications*. Ph.D. thesis, Carnegie Mellon University, USA, doi:10.1184/R1/25900861.V1.

[14] Chuta Sano, Ryan Kavanagh & Brigitte Pientka (2023): *Mechanizing session-types using a structural view: Enforcing linearity without linearity*. *Proc. ACM Program. Lang.* 7(OOPSLA), pp. 235:374–235:399, doi:10.1145/3622810.

[15] Daniel Zackon, Chuta Sano, Alberto Momigliano & Brigitte Pientka (2025): *Split Decisions: Explicit Contexts for Substructural Languages*. In Kathrin Stark, Amin Timany, Sandrine Blazy & Nicolas Tabareau, editors: *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2025, Denver, CO, USA, January 20-21, 2025*, ACM, pp. 257–271, doi:10.1145/3703595.3705888.

Barbed Similarity for the π-Calculus in Beluga: A Case Study in Coinductive Reasoning

Lea Trogni
Dipartimento di Matematica,
Università degli Studi di Milano, Italy

Gabriele Cecilia
School of Computer & Cyber Sciences,
Augusta University, Augusta, USA

Alberto Momigliano
Dipartimento di Informatica,
Università degli Studi di Milano, Italy

We formalize strong barbed similarity for the π-calculus in the Beluga proof assistant, completing a line of work addressing the Concurrent Calculi Formalization Benchmark. By extending previous developments to include replication, we give a coinductive encoding of behavioral equivalence based on barbs and internal actions. Using Beluga's copattern-based coinduction, we obtain concise and compositional proofs, including compatibility properties and a context lemma characterizing barbed precongruence. The case study demonstrates the effectiveness of combining HOAS and coinductive reasoning for mechanizing concurrent calculi.

1 Introduction

One of the reasons why people flock to cinemas to watch sequels and "rebooted" movies is that they know what they are getting into: they are familiar with the characters and the basic setup. Keeping with the cinematographic metaphor, we will cut to the chase and assume that the reader is already on board with the idea that the best assurance of the correctness of properties of a formal calculus is a machine-checked proof. In fact, this is becoming the norm in the semantics of programming languages; it is less established when *concurrency* is concerned, although the latter is of overwhelming importance in modern computing.

Enter the Concurrent Calculi Formalization Benchmark [4] (CCFB), which provides a suite of problems designed to evaluate the formalization of concurrent and distributed language models, in particular process calculi. Following the methodology of the POPLMark challenge, CCFB aims to assess current mechanization techniques, identify optimal formalization patterns, and ideally drive improvements in proof assistant tooling. The CCFB framework structures this evaluation around three orthogonal challenges: the management of linear resources, the handling of scope extrusion, and the implementation of coinductive proof techniques. For the latter, the challenge is to prove the "context lemma" for *strong barbed bisimilarity*, namely that this notion of bisimilarity can be turned into a congruence by making it sensitive to substitution and parallel composition.

Induction is one of the great success stories of proof assistants: although systems differ in the details of their foundations and automation, inductive definitions and proofs by induction are supported in essentially comparable ways across the board. Coinduction has had a less uniform history; for a survey, we refer to [17]. The earliest systematic approach, due to Paulson, treats coinductive predicates as greatest fixed points of monotone operators, from which the corresponding coinduction principles are derived.

A different tradition, familiar from Rocq/Coq, is based on guarded corecursion: coinductive objects are introduced by constructors, and recursive calls must occur under such constructors to ensure productivity. This discipline is often brittle in proof developments, especially when coinductive arguments are

Hermant, Tourret (Eds.): International Workshop on Logical Frameworks and Meta-Languages, Theory and Practice (LFMTP 26)
EPTCS 448, 2026, pp. 1–17, doi:10.4204/EPTCS.448.1

nested, combined with induction, or hidden behind auxiliary lemmas. Agda has explored several alternatives over time, from delays to sized types and observational presentations of codata to guarded type theories with the "later" modality [16]. Similarly, Beluga has embraced sized types and copatterns [19]. These developments are particularly relevant for mechanized concurrency, where bisimulations and up-to techniques routinely require coinductive proofs that are interleaved with substantial inductive reasoning about syntax, transitions, substitutions, and contexts.

The present paper is the conclusion of the Beluga "trilogy" of solutions to CCFB, initiated with [23] — featuring, among other contributions, a solution to the linearity challenge — and followed by [5] with the mechanization of the Harmony Lemma. As in any good sequel, we borrow from the latter paper the implementation of the syntax and the LTS-based operational semantics of the π-calculus, while crucially extending it with *replication*. After all, this is what makes behavioral equivalence interesting. And interesting it turns out to be, as there is a twist in this movie: the rule governing replication as proposed in [4] is insufficient to establish the main result of the challenge. In fact, under this rule, structural congruence is not included in strong barbed similarity. This is problematic, because structural congruence is the baseline syntactic equivalence between processes and the context lemma relies crucially on this specific inclusion.

While this lapse is mildly embarrassing, given the singleton intersection between the authors of the Benchmark and of the present paper, it is once more a reaffirmation of the usefulness of mechanizations in proof assistants.

We will assume familiarity with the basic notions of the π-calculus as in the first chapters of [18], as well as a working knowledge of Beluga, both of its syntax and of its approach to proof checking. In particular we will only briefly touch upon the way Beluga handles coinduction via observations and refer the reader to [19] for the theory and to [15] for an application.

In the following, the statements of informal lemmas, theorems and proofs are hyperlinked via the accompanying cute icon  to their formalization in the repository:

`https://github.com/LeaTrogni/formalizing-barbed-similarity-for-the-pi-calculus-in-beluga`

As a final note, we have cut one corner: we have concentrated on similarity, precongruence, etc., rather than bisimilarity and congruence. Extending the results to the symmetric case would only duplicate the code with no new insights and could be left to automation, perhaps a coding agent.

2 The π-Calculus and its Operational Semantics

To make the paper self-contained, we present the main definitions of the syntax and the labelled transition system (LTS) of the fragment of the π-calculus under study. For more details about the informal definitions, we refer the reader to [18].

2.1 Syntax

We follow the definitions of CCFB3 excluding sums and (mis)match, while we diverge from it by allowing the calculus to be name-passing rather than value-passing. That original separation of concerns is not a simplification in the HOAS approach, but just a quirk.

$$P,Q ::= \mathbf{0} \mid x(y).P \mid \bar{x}y.P \mid (P \mid Q) \mid (\nu x)P \mid !P$$

Recall that the input prefix $x(y).P$ and the restriction $(\nu y)P$ both bind the name y in P, while any other occurrence of names in a process is free. Accordingly, we write $\mathsf{fn}(P)$ and $\mathsf{bn}(P)$ for the sets of free and bound names occurring in a process.

Beluga is based on a two-level system, with the LF level for representing data and a computation level for reasoning about it via recursion and pattern matching. In the encoding, names are defined by an LF type `names` without any constructor, together with a context schema that allows names to occur in other (open) LF objects. Since we define no other context schema, all the variables in open terms will have type `names`. Processes are LF objects, encoded using (weak) HOAS for input and restriction, as well known and detailed in [5]; their encoding is displayed in Fig. 1. Throughout the development, lowercase identifiers denote LF types, while those beginning with an uppercase letter denote computation-level data.

```
LF names: type =;

LF proc: type =
  | p_zero: proc
  | p_in: names → (names → proc) → proc
  | p_out: names → names → proc → proc
  | p_par: proc → proc → proc
  | p_res: (names → proc) → proc
  | p_rep: proc → proc; --infix p_par 11 left.

schema ctx = names;
```

Figure 1: Encoding of names and processes.

2.2 Operational Semantics

Actions include inputs $x(y)$, free and bound outputs $\bar{x}y$ and $\bar{x}(y)$, and internal communications τ. In inputs and bound outputs, the occurrences of the name y are bound, while all other name occurrences in actions are free; from this, we obtain the standard notions of free names, bound names and names occurring in an action α, respectively denoted as $\mathsf{fn}(\alpha)$, $\mathsf{bn}(\alpha)$ and $\mathsf{n}(\alpha)$.

Our LTS, displayed in Fig. 2, presents some differences from the one in CCFB3. First, and less importantly, we have adopted *late* instead of *early* semantics: not only can they be proven equivalent in terms of transitions, see [5] for the mechanization of this folk result, but they induce the same strong barbed similarity, as established by Lemma 3.1 below and the discussion that follows. Secondly, as we teased in the introduction, we add two replication rules for communication to the LTS presented in CCFB3, since without them structural congruence is not included in strong barbed similarity: in the accompanying formalization we exhibit a pair of structurally congruent processes that are not barbed similar. This conflicts with the intended role of structural congruence as the strongest equivalence between processes, and the proof of the context lemma relies on this specific inclusion. The adoption of the two replication rules for communication is not arbitrary; after all, our LTS follows that in Sangiorgi and Walker's textbook, modulo late semantics. A popular alternative, found e.g. in [3, 8], uses a single rule to generate arbitrary copies of the replicated process. We refer to Sangiorgi and Walker for a discussion of the trade-offs between these formulations; the correspondence is understood modulo structural congruence.

Following the previous formalizations in [8, 13, 21], in Beluga we separate free steps, where the action does not bind any variable, from bound steps, where the action has a bound variable. Since some rules involve both free and bound actions at the same time, the LF types for free and bound steps are mutually defined. This approach does duplicate some rules, but gets rid of *all* provisos on free and bound occurrences, which are now taken care of by dependencies (or lack thereof) in second-order process variables.

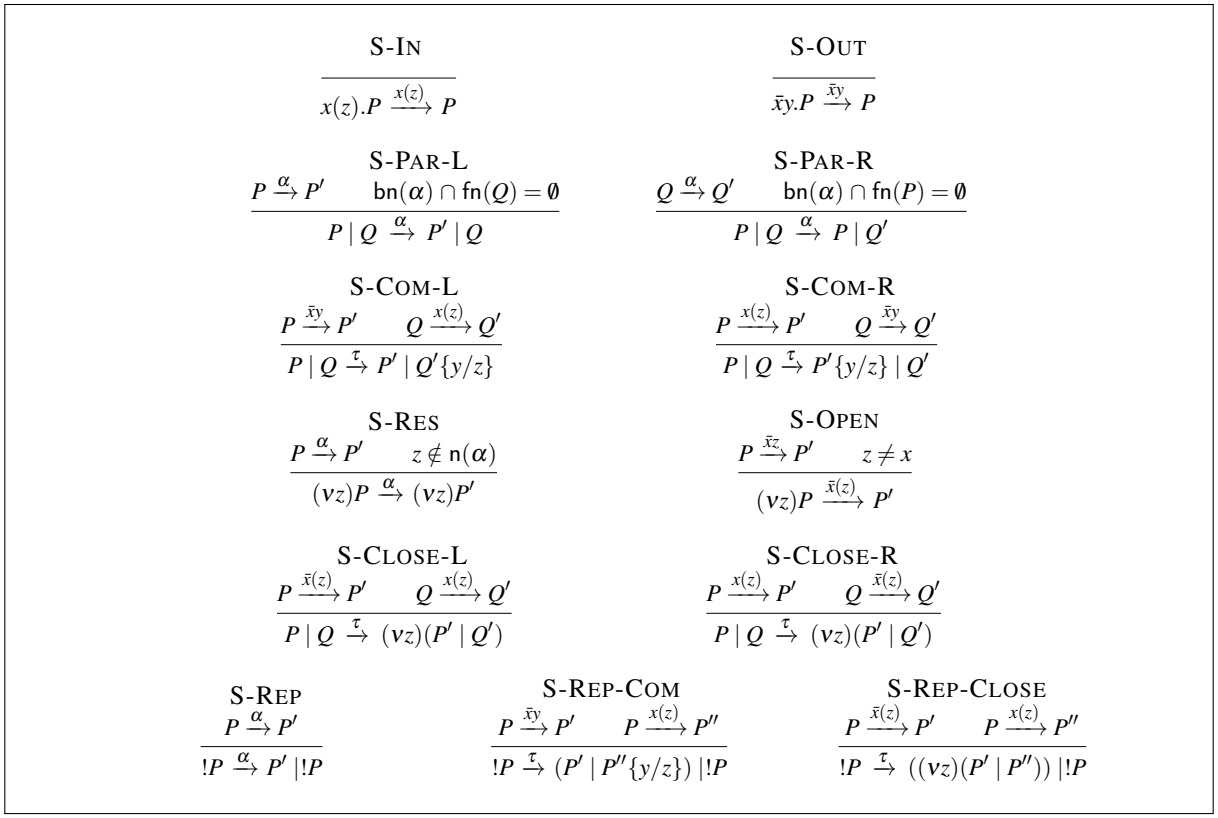

Figure 2: Transition rules.

2.3 Process Contexts and Structural Congruence

The notion of *(pre)congruence* is of paramount importance in the π-calculus: both in the reduction semantics, through *structural* congruence, and in the study of behavioral equivalence. Informally, a congruence is an equivalence relation that preserves the behavior of the constructors. Often, this is described in textbooks (e.g. [18]) via the notion of *process context*: a process where a (non-degenerate) occurrence of $\mathbf{0}$ is replaced by a hole. Given a context C and a process P, $C[P]$ denotes the process obtained by replacing the hole in C with P. Then, a process *pre*congruence is a binary relation $\mathscr{R}$ which is a preorder such that if $P \mathscr{R} Q$, then, for each context C, $C[P] \mathscr{R} C[Q]$.

There is however a peculiarity: since the goal of contexts is to observe how processes behave in every possible environment, they need to be able to capture the free variables in the processes they are filled with. In other words, contexts do not respect α-equivalence.

This brings in some additional issues with respect to a mechanization, in particular in a HOAS setting, where such entities are a non-starter [1]. A workaround is to close the relation under *compatibility* ([7,12]), as described by the rules in our case at the top of Fig. 3. Concretely, when we say that *structural congruence* is the smallest congruence satisfying the axioms in the bottom of Fig. 3, we are instantiating $\mathscr{R}$ with $\equiv$.

This is indeed the way structural congruence was formalized as an LF type `cong` in [5], which we have extended by adding the compatibility and unfolding laws for replication.

[1] HOAS is indeed compatible with weaker notions such as *evaluation* contexts, i.e., those that do not cross a binder.

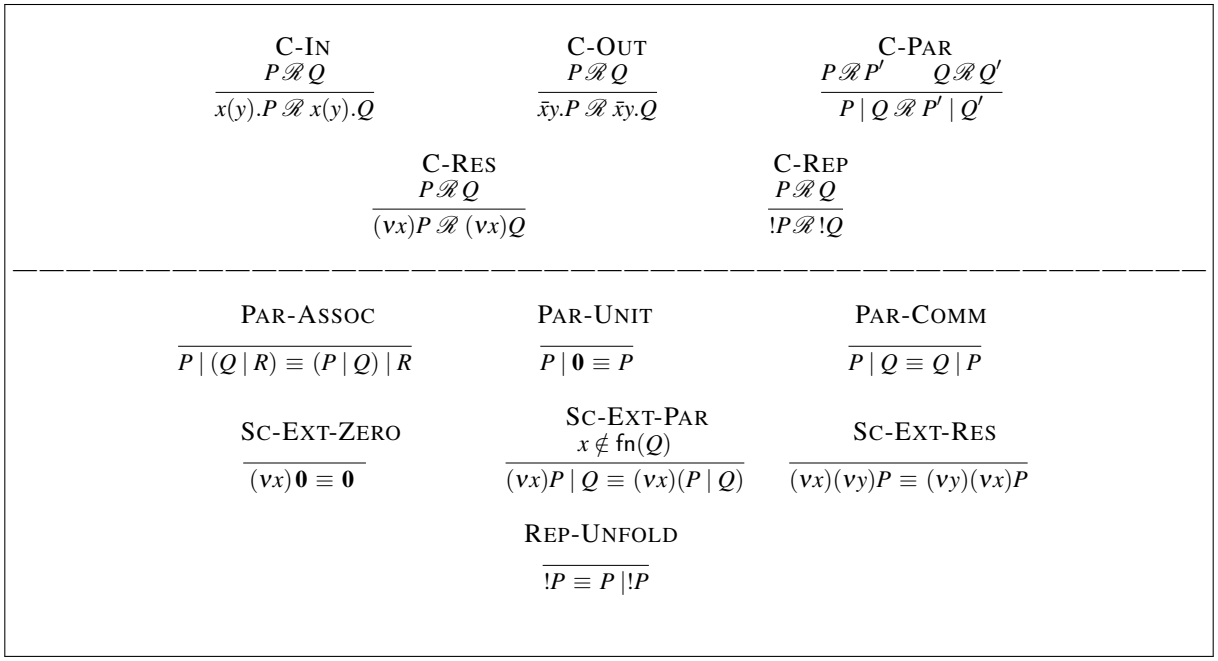

Figure 3: Compatibility rules and structural congruence axioms.

There are cases where going the compatibility route is inconvenient, if not plain inadequate, one being the definition of *strong barbed (pre)congruence*, as we shall see later on, and where contexts must be encoded. In a breakthrough, the authors of [11] introduced a clever trick to make contexts compatible (no pun intended) with a HOAS encoding, by means of a quaternary relation that implicitly captures the essence of a context:

$$\{(P,P',Q,Q') \mid P' = C[P],\ Q' = C[Q] \text{ for some context } C\}$$

Unlike in the Abella setting where this relation was introduced, a Beluga encoding is rather subtle as the LF contexts must be made explicit. In fact, in general $\mathsf{fn}(C[P]) \neq \mathsf{fn}(P)$, since the binders in C may capture names that occur free in P, and C may also introduce names that are fresh for P; while the latter can be avoided by working in an LF context that already contains all free variables of P and C, the former inevitably leads to a mismatch between the two LF contexts. Therefore, the formalization of this relation cannot be an LF predicate, where all the related terms are defined in an implicit context. Instead, we define the inductive type `PCtx` in Fig. 4, where the context of the second and fourth processes is a prefix of the context of the first and third processes.

```
inductive PCtx: (g:ctx)(g':ctx)[g' ⊢ proc] → [g ⊢ proc] → [g' ⊢ proc] → [g ⊢ proc] → ctype =
  | pidM: PCtx [g ⊢ P] [g ⊢ P] [g ⊢ Q] [g ⊢ Q]
  | pinM: PCtx [g' ⊢ P] [g,x:names ⊢ CP] [g' ⊢ Q] [g,x:names ⊢ CQ]
       → PCtx [g' ⊢ P] [g ⊢ p_in X (\x.CP)] [g' ⊢ Q] [g ⊢ p_in X (\x.CQ)]
  | poutM: PCtx [g' ⊢ P] [g ⊢ CP] [g' ⊢ Q] [g ⊢ CQ]
       → PCtx [g' ⊢ P] [g ⊢ p_out X Y CP] [g' ⊢ Q] [g ⊢ p_out X Y CQ]
  ...
```

Figure 4: Excerpt from encoding of contexts.

Structural congruence can alternatively be defined as a contextual equivalence via the inductive type `Cong` in Fig. 5, that uses the type `PCtx` in the *context closure* clause.

```
inductive Cong: (g:ctx) [g ⊢ proc] → [g ⊢ proc] → ctype =
  ...
% Replication unfolding
  | Rep_unfold: Cong [g ⊢ (p_rep P)] [g ⊢ (P p_par (p_rep P))]
% Context closure
  | C_ctx: Cong [g' ⊢ P] [g' ⊢ Q] → PCtx [g' ⊢ P] [g ⊢ CP] [g' ⊢ Q] [g ⊢ CQ]
        → Cong [g ⊢ CP] [g ⊢ CQ]
% Equivalence Relation Laws
  | C_ref: Cong [g ⊢ P] [g ⊢ P]
  | C_sym: Cong [g ⊢ P] [g ⊢ Q] → Cong [g ⊢ Q] [g ⊢ P]
  | C_trans: Cong [g ⊢ P] [g ⊢ Q] → Cong [g ⊢ Q] [g ⊢ R] → Cong [g ⊢ P] [g ⊢ R]
;
```

Figure 5: Excerpt from the inductive definition of structural congruence.

2.4 Preliminary Lemmas

We start by establishing some basic properties of the LTS and of process contexts. The former ones can be found in [18], modulo our choice of late semantics, while the latter ones, adapted from [11], stem from the implementation of process contexts.

Lemma 2.1 (Properties of the LTS)**.**

1. *If* $P \xrightarrow{\bar{x}y} P'$ *and* z *is a name, then either* $z = y$ *and* $\nu z P \xrightarrow{\bar{x}(z)} P'$ *or* $\nu z P \xrightarrow{\bar{x}y} \nu z P'$.
2. *If* $S \mid !P \xrightarrow{\tau} R$, *then there is* Q *such that* $S \mid (P \mid P) \xrightarrow{\tau} Q$ *and* $R \equiv Q \mid !P$.
3. *If* $x \notin \mathrm{fn}(P)$ *and* $P \xrightarrow{\alpha} P'$, *then* $x \notin (\mathrm{fn}(\alpha) \cup \mathtt{fn}(P'))$.
4. *If* $P \equiv Q$, *then*
   - *if* $P \xrightarrow{\alpha} P'$, *then there is* Q' *such that* $Q \xrightarrow{\alpha} Q'$ *and* $P' \equiv Q'$;
   - *if* $Q \xrightarrow{\alpha} Q'$, *then there is* P' *such that* $P \xrightarrow{\alpha} P'$ *and* $P' \equiv Q'$.

Proof. (Sketch) The first two statements are proven by a straightforward induction on the derivation of the transition in the hypothesis. The third one is split into two lemmas, separating free and bound actions, which are proven by a mutual induction on the derivation of the transition in the hypothesis. The last one is split into four lemmas (it requires to prove two different theses and in both we need to distinguish free and bound actions), which are proven by a mutual induction on the derivation of the congruence in the hypothesis. We note that the third and the fourth results are extensions of those in [5]. □

Now for properties of process contexts: some of them would be obvious with the on-paper definition of contexts, but are not trivial in the `PCtx` formalization. We recall that $((P,C_P),(Q,C_Q))$ means that there is a context C such that $C[P] = C_P$ and $C[Q] = C_Q$.

Lemma 2.2 (Properties of contexts)**.**

1. *(Symmetry) If* $((P,C_P),(Q,C_Q))$, *then* $((Q,C_Q),(P,C_P))$.

2. *(Transitivity) If* $((P,C_P),(R,C_R))$*, then for all* Q *there is* C_Q *such that* $((P,C_P),(Q,C_Q))$ *and* $((Q,C_Q),(R,C_R))$.
3. *(Functionality) If* $((P,C_P),(P,C'_P))$*, then* $C_P = C'_P$.
4. *(Context composition) If* $((P,C_P),(Q,C_Q))$ *and* $((C_P,C_{C_P}),(C_Q,C_{C_Q}))$*, then* $((P,C_{C_P}),(Q,C_{C_Q}))$.

Proof. All proofs are by induction on the derivation of the context relation in the hypothesis. □

As is well known [12], a binary relation on processes constitutes a contextual equivalence if and only if it is a compatible equivalence. Because Beluga's logic lacks support for higher-order predicates, this meta-theorem cannot be generalized; rather, the equivalence must be formalized independently for each specific relation.

Lemma 2.3. *Structural congruence can be characterized either via compatibility or by context closure.*

Proof. (Sketch) Both inclusions are proved by induction on the derivation of the congruence rule in the hypothesis. The inclusion of the compatible relation in the contextual one first requires to prove that the compatible relation is contextually closed too. □

The LF definition is more convenient for discussing the properties of the LTS, as in [5], while the inductive one is used later to show the inclusion of structural congruence in another precongruence defined via contexts.

3 Behavioral Equivalences

Intuitively, two processes can be considered equivalent when they exhibit the same behavior. However, the level of detail at which behaviors can be distinguished depends on the chosen observational granularity; this is where a variety of bisimilarity notions come into play. In CCFB3, the authors focus on *strong barbed bisimilarity*, which relates processes that can match internal (τ) transitions and preserve the same *barbs*, i.e. the ability to perform input or output on a given channel.

More precisely, the barb, or observability predicate, $P \downarrow_x$ (resp. $P \downarrow_{\overline{x}}$) holds if a process P can perform an input action with subject x (resp. an output action with subject $\overline{x}$). Following [18], this notion can be formalized in two equivalent ways, based either on the structure of the process P or on the (late) LTS:

i) $P \downarrow_x$ iff $P \equiv \nu z_1 \ldots \nu z_n (x(y).Q \mid R)$ for some $y, z_1, \ldots, z_n, Q, R$, with $x \notin \{z_1, \ldots, z_n\}$. Analogously, $P \downarrow_{\overline{x}}$ iff $P \equiv \nu z_1 \ldots \nu z_n (\overline{x}y.Q \mid R)$ for some $y, z_1, \ldots, z_n, Q, R$, with $x \notin \{z_1, \ldots, z_n\}$.
The telescopes, i.e. n-ary sequences of binders, appearing in this definition of barb are encoded in Beluga by two types `barb_in_rew` and `barb_out_rew`, that build the sequence of restrictions incrementally. Then, the two types encoding barbs identify processes congruent to such telescopes.

ii) $P \downarrow_x$ iff $P \xrightarrow{x(z)} P'$ for some z, P', and similarly $P \downarrow_{\overline{x}}$ iff $P \xrightarrow{\overline{x}y} P'$ or $P \xrightarrow{\overline{x}(z)} P'$ for some y, z, P'.
We formalize this at the meta level, but it could have been formulated as an LF type as easily. Since the LTS presents one input label and two output labels, `Barb_in` has one constructor, while `Barb_out` has two.

Definitions i) and ii) are equivalent, in the sense given by the following lemma. For brevity, we omit some auxiliary technical lemmas and refer to the repository for full details. 

Lemma 3.1. *Let P be a process, x be a name. The following are equivalent:*

1. $P \equiv \nu z_1 \ldots \nu z_n (x(y).Q \mid R)$ *for some* $y, z_1, \ldots, z_n, Q, R$, *with* $x \notin \{z_1, \ldots, z_n\}$;
2. $P \xrightarrow{x(z)} P'$ *for some* z, P'. 

Similarly, the following are equivalent:

1. $P \equiv \nu z_1 \ldots \nu z_n (\bar{x}y.Q \mid R)$ *for some* $y, z_1, \ldots, z_n, Q, R$, *with* $x \notin \{z_1, \ldots, z_n\}$;
2. $P \xrightarrow{\bar{x}y} P'$ *or* $P \xrightarrow{\bar{x}(z)} P'$ *for some* y, z, P'. 

Proof. (Sketch) The proof that i) implies ii) is a straightforward induction on the rewriting definitions. The converse proceeds by structural induction on P, followed by an inversion on the observable transition that stems from P; some subcases are handled by the omitted auxiliary lemmas. □

Being invariant w.r.t. the underlying LTS, definition i) can be adopted even in the setting of reduction semantics; however, definition ii) has the benefit of being more amenable to mechanized proofs without the need of algebraic rewrites. For this reason, we use the latter in the rest of the development.

A significant property of barbs is that they are preserved by contexts:

Lemma 3.2.

1. *If* $P \downarrow_x$ *implies* $Q \downarrow_x$ *and* $((P, C_P), (Q, C_Q))$, *then* $C_P \downarrow_x$ *implies* $C_Q \downarrow_x$. 
2. *If* $P \downarrow_{\bar{x}}$ *implies* $Q \downarrow_{\bar{x}}$ *and* $((P, C_P), (Q, C_Q))$, *then* $C_P \downarrow_{\bar{x}}$ *implies* $C_Q \downarrow_{\bar{x}}$. 

Proof. By induction on the derivation of the context relation. □

We can now give the definition of barbed simulation. A binary relation on processes $\mathscr{R}$ is a *strong barbed simulation* if the following conditions hold:

1. $\mathscr{R}$ is *barb preserving*: if $P \mathscr{R} Q$ and $P \downarrow_x$, then $Q \downarrow_x$; likewise, if $P \mathscr{R} Q$ and $P \downarrow_{\bar{x}}$, then $Q \downarrow_{\bar{x}}$.
2. $\mathscr{R}$ is a *reduction simulation*: if $P \mathscr{R} Q$ and $P \xrightarrow{\tau} P'$, then there is Q' such that $Q \xrightarrow{\tau} Q'$ and $P' \mathscr{R} Q'$.

The union of all barbed simulations is called *strong barbed similarity* and will be denoted by $\overset{\bullet}{\leq}_B$.

As explained above, barbs are not based on a specific LTS, and, as proven in [5], albeit without the replication operator, early and late semantics provide the same τ actions. Thus, our definition of strong barbed similarity turns out to be equivalent to the one based on the early semantics.

(Bi)simulations are perhaps the best known instance of coinductively defined relations. In calculi where the transition relation is well-founded — e.g., those where each transition produces a structurally smaller process, ensuring that every computation trace is finite — (bi)similarity can equivalently be characterized inductively. This is not the case in our fragment of the π-calculus, where infinite behaviors arise due to replication: thus, a genuinely coinductive treatment becomes necessary.

To our rescue, Beluga supports coinductive reasoning in a very flexible way [19]. Coinductive types are defined dually to inductive types: instead of constructors, they are characterized by destructors, or *observations*. More specifically, while an inductive type is defined by inference rules whose conclusion is an instance of the type, a coinductive type is defined by inference rules where an instance of the type is the premise. Beluga separates this premise from the rest of the rule by using `::`.

Reasoning on coinductive objects proceeds via *copattern matching*. By the Curry-Howard correspondence, proofs by (co)induction are encoded as total (co)recursive functions. In the coinductive case, totality requires coverage of all observations, while productivity ensures that each corecursive call is guarded by an observation; in the (current) absence of a productivity checker in Beluga, this condition must be verified manually. Observations of a coinductive object `c` are accessed using the syntax `c.observation_name`, mirroring field access in mainstream programming languages.

As an example of a coinductive definition, we can look at the formalization of strong barbed similarity in Fig. 6, which presents three observations, corresponding to the input barbs, the output barbs and the τ transitions. Recall that Beluga does not provide constructs for existential quantification, conjunction, or disjunction, so these must be encoded as inductive types.

```
coinductive BarbSim : (g:ctx) [g ⊢ proc] → [g ⊢ proc] → ctype =
  | (BarbSim_barb_in : BarbSim [g ⊢ P] [g ⊢ Q])
              :: Barb_in [g ⊢ P] [g ⊢ X] → Barb_in [g ⊢ Q] [g ⊢ X]
  | (BarbSim_barb_out : BarbSim [g ⊢ P] [g ⊢ Q])
              :: Barb_out [g ⊢ P] [g ⊢ X] → Barb_out [g ⊢ Q] [g ⊢ X]
  | (BarbSim_tau : BarbSim [g ⊢ P] [g ⊢ Q])
              :: [g ⊢ fstep P f_tau P'] → Ex_sim_barb [g ⊢ P] [g ⊢ Q] [g ⊢ P']

and inductive Ex_sim_barb : (g:ctx) [g ⊢ proc] → [g ⊢ proc] → [g ⊢ proc] → ctype =
  | Ex_sim_barb_def : [g ⊢ fstep Q f_tau Q'] → BarbSim [g ⊢ P'] [g ⊢ Q']
                  → Ex_sim_barb [g ⊢ P] [g ⊢ Q] [g ⊢ P']
;
```

Figure 6: Coinductive definition of strong barbed similarity.

Lemma 3.3 (Properties of strong barbed similarity)**.** *Strong barbed similarity*

1. *is a preorder;*
2. *is preserved by input prefix, output prefix and restriction;*
3. *includes structural congruence.*

Proof.

1. By a straightforward coinduction, as in [15].
2. To illustrate how Beluga implements a coinductive proof, we detail the restriction case. On paper, we would prove that $\{(\nu z P, \nu z Q) \mid P \dot{\leq}_B Q\}$ is a strong barbed simulation and therefore is included in $\dot{\leq}_B$. In Beluga, this is realized by the recursive function in Fig. 7, whose signature encodes the statement. The proof distinguishes cases on the possible observations — note the copattern syntax reminiscent of record selection:
   - In the barbs cases we employ Lemma 3.2, by which if the barbs of P are included in the barbs of Q, then for each context C (and in particular $\nu z[\_]$), the barbs of $C[P]$ are included in the barbs of $C[Q]$.
   - More interestingly, in the transition case we invert on the restriction rule for a τ action from P to some P' (the first `let`). Then, using the hypothesis $P \dot{\leq}_B Q$ (`BarbSim`), we can find a matching action from Q to Q', where $P' \dot{\leq}_B Q'$. Finally, we can corecursively call the theorem

on $P' \dot{\leq}_B Q'$ and, by applying the restriction rule forward, obtain the thesis. The corecursive call is valid, because it is guarded by the `BarbSim_tau` observation and the proof covers all cases, because we analyzed all the observable properties that characterize strong barbed similarity.

3. This follows immediately from Lemma 2.1.4, which was also crucial for one implication of the Harmony Lemma.

□

```
rec res_barb_sim: (g:ctx) BarbSim [g,x:names ⊢ P] [g,x:names ⊢ Q] →
                   BarbSim [g ⊢ p_res \x.P] [g ⊢ p_res \x.Q] =
fun d .BarbSim_barb_in b ⇒
       PCtx_preserves_barb_in (mlam Y ⇒ fn b' ⇒ d .BarbSim_barb_in b') (presM pidM) b
  | d .BarbSim_barb_out b ⇒
       PCtx_preserves_barb_out (mlam Y ⇒ fn b' ⇒ d .BarbSim_barb_out b') (presM pidM) b
  | d .BarbSim_tau s ⇒ let [_ ⊢ fs_res \x.S] = s in
       let Ex_sim_barb_def [_,x:names ⊢ S'] b' = d .BarbSim_tau [_,x:names ⊢ S] in
       Ex_sim_barb_def [_ ⊢ fs_res \x.S'] (res_barb_sim b');
```

Figure 7: Encoding of the proof of Lemma 3.3.2.

We now recall the notion of *strong barbed simulation up to* $\dot{\leq}_B$, namely a binary relation on processes $\mathscr{R}$ such that:

1. $\mathscr{R}$ is barb preserving.
2. if $P \mathscr{R} Q$ and $P \xrightarrow{\tau} P'$, then there is Q' such that $Q \xrightarrow{\tau} Q'$ and $P' \dot{\leq}_B \mathscr{R} \dot{\leq}_B Q'$.

As above, *strong barbed similarity up to* $\dot{\leq}_B$ is the union of all strong barbed simulations up to $\dot{\leq}_B$.

Again, we cannot develop a general theory of relations up to: we simply encode strong barbed similarity up to $\dot{\leq}_B$ as we did for strong barbed similarity — the only difference being in the inductive auxiliary type, which encodes the *up to* $\dot{\leq}_B$.

In this setup, strong barbed similarity up to $\dot{\leq}_B$ coincides with $\dot{\leq}_B$. The use here of *up-to technique* is just for convenience: in order to prove that a relation is included in strong barbed similarity, it is sufficient to prove that it is a strong barbed simulation up to $\dot{\leq}_B$ and therefore included in barbed similarity up to $\dot{\leq}_B$.

3.1 Barbed Precongruence

One way to distinguish two processes is to observe whether they exhibit the same behavior when placed in the same environment, or context; a desirable property of a (bi)similarity is to be preserved by every possible environment. Unfortunately, strong barbed similarity does not satisfy contextual closure. For instance, the two processes $\bar{x}y.\bar{a}b.0$ and $\bar{x}y.0$ are strong barbed similar, since their only barb is $\downarrow_{\bar{x}}$ and they cannot perform any internal transition. Conversely, plugging them into the context $[\_] \mid x(z).0$ modifies their observable behavior: the former can perform a transition $\bar{x}y.\bar{a}b.0 \mid x(z).0 \xrightarrow{\tau} \bar{a}b.0 \mid 0$, enabling the observation of a barb $\downarrow_{\bar{a}}$ that the process $\bar{x}y.0 \mid x(z).0$, after an internal transition, cannot produce.

Therefore, we aim to identify those pairs of processes whose behavior remains strongly barbed similar under all contexts. The processes P and Q are *strongly barbed precongruent*, denoted by $P \leq_B Q$, if, for each context C, $C[P] \dot{\leq}_B C[Q]$.

Among the properties of strong barbed precongruence, the most prominent is its characterization as the largest precongruence included in strong barbed similarity. It turns out that the latter can be formalized as a coinductive definition with only two observations, requiring the inclusion in $\dot{\leq}_B$ and the contextual closure.

Lemma 3.4 (Properties of strong barbed precongruence)**.** *Strong barbed precongruence*

1. *is a preorder;*
2. *is included in strong barbed similarity;*
3. *is a precongruence;*
4. *is the largest precongruence included in strong barbed similarity;*
5. *includes structural congruence.*

The first three proofs immediately follow from the definition of $\leq_B$, the fact that $\dot{\leq}_B$ is a preorder and context composition; the last two are straightforward coinductive arguments.

4 Context Lemma

The objective of CCFB3 is to prove that making barbed bisimilarity sensitive to substitutions and parallel composition is sufficient to establish barbed congruence. This result is an instance of a *context* lemma, namely a characterization of congruence that relaxes the universal quantification on arbitrary contexts, thereby simplifying proofs of congruence.

We recall that *substitutions* are endofunctions on names with finite support; they can be extended to processes and actions by simultaneously replacing all their free occurrences of names. To encode them, we rely on Beluga's built-in notion of simultaneous substitutions. In particular, given two Beluga contexts `g,h:ctx`, a substitution `$S:$[h |- g]` maps the names in `g` to names in `h`; given a process `P:[g |- proc]` that depends on names in `g`, the process `P[$S]:[h |- proc]` is obtained by replacing its names according to `$S`.

Next, we denote as $\leq_{|s} := \{(P,Q) \mid \text{for all } R,\sigma, (P\sigma \mid R) \dot{\leq}_B (Q\sigma \mid R)\}$ the relation obtained by closing barbed similarity under parallel composition and substitutions. Analogously to barbed precongruence, it is formalized by an inductive type `BarbPre'`.

The strategy to prove the context lemma consists in showing that $\leq_{|s}$ is a precongruence included in strong barbed similarity; since, by Lemma 3.4.4, strong barbed precongruence is the largest precongruence included in strong barbed similarity, it follows that $\leq_{|s} \subseteq \leq_B$. Below, we detail the results required to complete the proof.

First, an auxiliary lemma describing how transitions of a process $P\sigma$ arise from transitions of P:

Lemma 4.1. *If $P\sigma \xrightarrow{\alpha} P'$ and $\alpha \neq \tau$, then there are β, P'' such that $P \xrightarrow{\beta} P''$ and $\alpha = \beta\sigma$.*

Proof. (Sketch) By inversion on α and then structural induction on P. In the Beluga encoding, the three cases $\alpha = x(y), \bar{x}y$ and $\bar{x}(y)$ are addressed separately. □

Proving that $\leq_{|s} \subseteq \overset{\bullet}{\leq}_B$ is straightforward:

Lemma 4.2. *$\leq_{|s}$ is included in strong barbed similarity.*

Proof. It is sufficient to observe that, if $P \leq_{|s} Q$, then $P \equiv P(\mathsf{Id}) \mid 0 \overset{\bullet}{\leq}_B Q(\mathsf{Id}) \mid 0 \equiv Q$, where Id denotes the identity substitution. □

The proof that $\leq_{|s}$ is a precongruence is trickier and is based on the following result — this is, in fact, the only instance in which coinductive up-to techniques are used.

Lemma 4.3. *The following relations are included in strong barbed similarity:*

1. $\mathscr{R}_1 \coloneqq \{(x(y).(P\sigma) \mid R,\ x(y).(Q\sigma) \mid R) \colon P \leq_{|s} Q\}$.
2. $\mathscr{R}_2 \coloneqq \{(\bar{x}y.(P\sigma) \mid R,\ \bar{x}y.(Q\sigma) \mid R) \colon P \leq_{|s} Q\}$.
3. $\mathscr{R}_3 \coloneqq \{(!(P\sigma) \mid R,\ !(Q\sigma) \mid R) \colon P \leq_{|s} Q\}$.

Proof. We prove that $\mathscr{R}_1 \cup \overset{\bullet}{\leq}_B$ and $\mathscr{R}_2 \cup \overset{\bullet}{\leq}_B$ are strong barbed simulations, while $\mathscr{R}_3$ is a strong barbed simulation up to $\overset{\bullet}{\leq}_B$. The proofs, both as pen-and-paper and in Beluga, are carried out by coinduction, considering each possible observation and constructing the corresponding object required by the definition.

1. Checking that the elements of $\overset{\bullet}{\leq}_B$ satisfy all the observations is immediate, hence we focus on the elements of $\mathscr{R}_1$. Proving that $\mathscr{R}_1 \cup \overset{\bullet}{\leq}_B$ is barb preserving is also immediate, by looking at the structure of the pairs of processes in $\mathscr{R}_1$. To show that it is a reduction simulation, we proceed by inversion on the internal transition s that needs to be matched; in particular, we illustrate the case in which s has been derived from the S-COM-R rule, since it highlights one of the properties of substitutions that Beluga directly supports.

   The transition s has the form $x(z).P\sigma \mid R \xrightarrow{\tau} (P\sigma)\{y/z\} \mid R'$ for some y, R'; similarly, we can see that $x(z).Q\sigma \mid R \xrightarrow{\tau} (Q\sigma)\{y/z\} \mid R'$. As the composition of substitutions is a substitution, we conclude by observing that $((P\sigma)\{y/z\} \mid R',\ (Q\sigma)\{y/z\} \mid R') = (P(\sigma \circ \{y/z\}) \mid R',\ Q(\sigma \circ \{y/z\}) \mid R') \in \mathscr{R}_1$.
2. Analogous to the previous case.
3. Proving that $\mathscr{R}_3$ is barb preserving requires showing that a transition $s : !(P\sigma) \xrightarrow{\alpha} P'$, with $\alpha \neq \tau$, is matched by $!(Q\sigma)$. After inversion on s through the S-REP rule, the desired transition is built after applying Lemma 4.1.

   Conversely, proving the reduction closure up to $\overset{\bullet}{\leq}_B$ of $\mathscr{R}_3$ does not require any explicit inversion; rather, it is enough to apply Lemma 2.1.2 and the coinductive hypothesis to build a long chain of structural congruences and strong barbed similarities.

□

Lemma 4.4. *$\leq_{|s}$ is a precongruence.*

Proof. By induction on the structure of the given context. The prefixes and replication cases make use of Lemma 4.3, while the others rely on chains of $\equiv$ and $\overset{\bullet}{\leq}_B$. □

Lemma 4.5. *$\leq_{|s}$ is included in the largest precongruence included in strong barbed similarity.*

Proof. Follows immediately from Lemmas 4.2 and 4.4. □

Theorem 4.1 (Context Lemma)**.** $\leq_{|s}$ *is included in strong barbed precongruence.*

Proof. Follows from Lemma 3.4.4, characterizing barbed precongruence as the largest precongruence included in strong barbed similarity. □

5 Evaluation

The entire development comprises approximately 1500 lines of code, organized into nine files. It includes 23 definitions — three of which are coinductive — accounting for roughly 10% of the total codebase, as well as 53 theorems. Most of these theorems exactly correspond to the relevant results presented in [18]; however, several of the more intricate ones must be decomposed into multiple statements. This is particularly necessary when they involve combinations of induction and coinduction with differing proof goals. Notably, the only strengthening lemmas required in the Beluga formalization are those already needed at the paper level to satisfy the side conditions of the LTS rules concerning free and bound names. This once again underscores the advantages of using HOAS for handling binders.

A central feature of the π-calculus is its ability to pass names, enabling subprocesses to communicate channel names and subsequently use them. Concretely, input prefixes bind received names, and in all standard semantics there is a transition in which the binder is eliminated and the bound name is replaced by a fresh free name representing the received value. In Beluga, such single substitutions are treated as a special case of simultaneous substitutions, which allows them to be composed seamlessly, as demonstrated in Lemma 4.3. This is a net gain compared to other approaches (such as [3]), where the two notions need to be implemented from scratch.

One complication arising from substitutions in Beluga is their interaction with the coverage checker. Verifying coverage often requires solving non-trivial unification problems, which can become more challenging in the presence of substitutions: this is well understood in the theory, but simply not supported yet by the Beluga implementation. Consequently, Lemma 4.1 is formalized without a totality check. In fairness, we find more urgent the implementation of the *productivity* checker: for what it is worth, we have manually checked that all coinductive calls comply with productivity, as we understand it, but this falls somewhat short of a complete assurance of the correctness of our mechanization.

In the CCFB paper, the authors stated:

> *The crux of our challenge is the effective use of coinductive up-to techniques. The intention is that the result should be relatively easy to achieve once the main properties of bisimilarity are established.*

However, coinductive up-to techniques, which have a prominent role in the study of other bisimilarities, are only used here once, i.e. to prove the context lemma, and the formalization of strong barbed similarity up to $\overset{\bullet}{\leq}_B$ to does not diverge significantly from the formalization of strong barbed similarity. We conclude that the design of the challenge does not exercise significantly the up-to aspect.

6 Related Work

In this section we mostly concentrate on mechanizations of behavioral equivalence in process calculi.

In [2, 3], Bengtson and Parrow provide the largest formalization of the π-calculus in the literature using nominal logic in Isabelle/HOL to represent binders. The encoding of the operational semantics uses *commitments* [14] rather than labelled transitions, i.e. pairs of an action and a derivative process that binds the bound names of the action in the derivative. The treatment of replication follows the single-rule presentation of [8]. Their results include proving that strong and weak bisimulations, both early and late, are congruences, and that structural congruence is a bisimulation; the authors also provide an axiomatization of strong late bisimulation and prove that it is sound and complete. Coinductive definitions and proofs are encoded by using Isabelle/HOL's standard coinductive package. Since most of these details are handled by Isabelle, the main focus of Bengtson and Parrow is on the nominal logic aspects of the formalization, which, while elegantly supporting free and bound names, still entails a fair amount of work to establish basic infrastructural properties.

Ambal et al. [1] provide a Coq formalization of the higher-order π-calculus that systematically studies binder representations for mixed binding (process vs name binders) and establishes strong context bisimilarity as a congruence via Howe's method. Coinduction is used in the standard way to characterize bisimilarity; coinductive "guarded" proofs only appear in the final step showing that the Howe closure is included in bisimilarity. A direct Coq development in the style of ours would likely require libraries such as PACO [10]. Further, the HOAS approach encodes process and name binders in the same way, offering significant simplifications.

In [20], Tian and Sangiorgi study proof methods for bisimulation in CCS, focusing on contractions rather than equations. They formalize the congruence induced by weak bisimilarity (rooted bisimilarity) in HOL4 using the coinductive package `Hol_coreln`. Their development does not treat processes up to α-conversion, and restriction is not a binder; contexts are therefore represented as single-binder λ-expressions applied to processes. As in our work, the coarsest congruence contained in bisimilarity is characterized via closure under composition, although here it is ancillary to the main result on unique solutions of rooted contractions.

Tiu and Miller [21] give a proof-search specification of the finite π-calculus in a logic equipped with definitions, fixed points, and the ∇ quantifier for generic judgments. Their encoding is equivalent to ours w.r.t. syntax and LTS. It does seem more general insofar that the interaction between the $\forall$, $\exists$, and ∇ quantifiers explains the usual distinctions between early, late, and, remarkably, open bisimulation. For the latter the ∇ quantifier plays a role analogous to Sangiorgi's *distinctions*.

Veltri and Vezzosi [22] give a much more extensive formalization of process-calculus semantics in Guarded Cubical Agda: they develop fully abstract denotational models for CCS and the early π-calculus, showing that denotational equality coincides with bisimilarity. Their encoding uses well-scoped de Bruijn syntax, in contrast with our HOAS representation in Beluga. Their approach to coinduction is also quite different: guarded recursion, clocks, and cubical path equality provide the semantic infrastructure through which bisimilarity is recovered as equality, whereas we reason directly with coinductive operational relations.

7 Conclusions and Future Work

What happens if a challenge turns out to be not too challenging? Is it the designers' fault or the encoders' merit? In our previous paper we mused on how uneventful the formalization had been. We have to repeat ourselves: the HOAS encoding of the syntax and semantics of the calculus, which we have inherited from a long tradition (dating back to [13]), abstracts all the issues related to free and bound names that have preoccupied all first-order encodings. What the present paper brings to the table is a very intensive

use of coinductive reasoning: here, Beluga's copattern approach shines, much more than in a previous coinductive case study (Howe's method, see [15], which turns out to exercise coinduction only in a limited fashion).

Beluga's underlying sized types discipline is intrinsically compositional, making nesting coinductive proofs completely unproblematic. This is in sharp contrast with the guarded recursion approach native to Rocq, which would have been simply unfeasible and compelled the user to switch to dedicated libraries such as PACO [10]. Again, it is the support for both HOAS and coinduction via observation that makes, in our judgment, the development so smooth — Agda supports a similar approach to coinduction, but not HOAS; Abella offers the converse trade-off.

Future Work. There are several directions in which the present formalization could be extended. A natural first step is to establish the converse direction of the context lemma. The proof presented in [18] proceeds by constructing a suitable context such that a pair of strongly barbed congruent processes can be composed in parallel under substitution so as to yield strongly barbed bisimilar processes. The argument relies on multi-step internal transitions whose length depends on the size of the support of the substitution. It is not clear to us whether Beluga's theory of substitution directly captures this style of argument, particularly when it involves a fine-grained analysis of substitutions and their supports. Moreover, the proof makes essential use of the symmetry of strong barbed bisimilarity, so we would have to carry out the symmetric proofs after all.

Open bisimulation is particularly appealing in our setting because Beluga's contextual objects and simultaneous substitutions seem to provide much of the infrastructure needed to express substitution-closed behavioral relations. The main additional burden would be the treatment of distinctions/freshness constraints and their interaction with the LTS. Since Beluga does not have a ∇ quantifier, one must prove that the resulting substitution-based relation coincides with the standard distinction-indexed definition, or else identify precisely which variant of open similarity has been mechanized. One possibility to avoid an explicit representation of distinctions could be to look at the theory of *quasi-open* bisimilarity [6, 9].

Another technically involved extension concerns the coincidence between strong barbed congruence and early congruence. The standard proof requires the introduction of a stratification of strong early bisimilarity, with the key inclusion shown via a contrapositive argument. This reliance on classical reasoning renders a direct mechanization in Beluga particularly challenging.

Finally, Lassen in [12] studies contextual closure as the greatest *adequate* congruence in the context of program equivalence for a functional language. Since adequacy has a barbed "flavor", the question is whether analogous characterizations can be established for bisimilarity-based congruences in the π-calculus.

Acknowledgments

This work is partially supported by the National Science Foundation under Grant No. 2242786 (SHF:Small:Concurrency In Reversible Computations). We thank Brigitte Pientka for several helpful conversations.

AI usage statement. The formalization, definitions, and proofs presented in this paper are entirely the authors' work. Generative AI tools were used solely for editorial assistance (improving clarity and English wording), and had no role in the technical development or validation of the results.

References

[1] Guillaume Ambal, Sergueï Lenglet & Alan Schmitt (2021): *HOπ in Coq*. *J. Autom. Reason.* 65(1), pp. 75–124, doi:10.1007/S10817-020-09553-0.

[2] Jesper Bengtson & Joachim Parrow (2007): *A Completeness Proof for Bisimulation in the π-calculus Using Isabelle*. *Electronic Notes in Theoretical Computer Science* 192, pp. 61–75, doi:10.1016/J.ENTCS.2007.08.017.

[3] Jesper Bengtson & Joachim Parrow (2009): *Formalising the π-Calculus using Nominal Logic*. *Log. Methods Comput. Sci.* 5, doi:10.2168/LMCS-5(2:16)2009.

[4] Marco Carbone, David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, Frederik Krogsdal Jacobsen, Alberto Momigliano, Luca Padovani, Alceste Scalas, Dawit Tirore, Martin Vassor, Nobuko Yoshida & Daniel Zackon (2024): *The Concurrent Calculi Formalisation Benchmark*. In Ilaria Castellani & Francesco Tiezzi, editors: *Coordination Models and Languages*, Springer Nature Switzerland, Cham, pp. 149–158, doi:10.1007/978-3-031-62697-5_9.

[5] Gabriele Cecilia & Alberto Momigliano (2024): *A Beluga Formalization of the Harmony Lemma in the π-Calculus*. In Florian Rabe & Claudio Sacerdoti Coen, editors: *Proceedings Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP@FSCD 2024, Tallinn, Estonia, 8th July 2024*, EPTCS, pp. 1–17, doi:10.4204/EPTCS.404.1.

[6] Yuxi Fu (2005): *On Quasi-Open Bisimulation*. *Theor. Comput. Sci.* 338(1-3), pp. 96–126, doi:10.1016/J.TCS.2004.10.041.

[7] Simon J. Gay & Vasco T. Vasconcelos (2025): *Session Types*. Cambridge University Press, doi:10.1017/9781009000062.

[8] Furio Honsell, Marino Miculan & Ivan Scagnetto (2001): *π-Calculus in (Co)Inductive Type Theory*. *Theor. Comput. Sci.* 253(2), pp. 239–285, doi:10.1016/S0304-3975(00)00095-5.

[9] Ross Horne, Ki Yung Ahn, Shang-Wei Lin & Alwen Tiu (2018): *Quasi-Open Bisimilarity with Mismatch is Intuitionistic*. In Anuj Dawar & Erich Grädel, editors: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, ACM, pp. 26–35, doi:10.1145/3209108.3209125.

[10] Chung-Kil Hur, Georg Neis, Derek Dreyer & Viktor Vafeiadis (2013): *The Power of Parameterization in Coinductive Proof*. In: *POPL '13: Proc. 40th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, ACM, New York, p. 193–206, doi:10.1145/2429069.2429093.

[11] Adrienne Lancelot, Beniamino Accattoli & Maxime Vemclefs (2025): *Barendregt's Theory of the λ-Calculus, Refreshed and Formalized*. In Yannick Forster & Chantal Keller, editors: *16th International Conference on Interactive Theorem Proving, ITP 2025, September 28 to October 1, 2025, Reykjavik, Iceland*, *LIPIcs* 352, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, doi:10.4230/LIPIcs.ITP.2025.13.

[12] Søren Bøgh Lassen (1998): *Relational Reasoning about Functions and Nondeterminism*. Doctoral dissertation, Faculty of Science of the University of Aarhus. Available at `https://www.brics.dk/DS/98/2/`.

[13] Dale Miller (1994): *Specification of the π-Calculus*. Available at `http://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/examples/pi-calculus/toc.html`.

[14] Robin Milner (1993): *The Polyadic π-Calculus: a Tutorial*. In Friedrich L. Bauer, Wilfried Brauer & Helmut Schwichtenberg, editors: *Logic and Algebra of Specification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 203–246, doi:10.1007/978-3-642-58041-3_6.

[15] Alberto Momigliano, Brigitte Pientka & David Thibodeau (2019): *A Case-Study in Programming Coinductive Proofs: Howe's Method*. *Math. Struct. Comput. Sci.* 29(8), pp. 1309–1343, doi:10.1017/S0960129518000415.

[16] Hiroshi Nakano (2000): *A Modality for Recursion*. In: *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*, IEEE Computer Society, pp. 255–266, doi:10.1109/LICS.2000.855774.

[17] Damien Pous & Dmitriy Traytel (2026): *Handbook of Proof Assistants*, chapter Coinductive methods. Springer. Forthcoming.

[18] Davide Sangiorgi & David Walker (2001): *The π-Calculus - a Theory of Mobile Processes*. Cambridge University Press, doi:10.2178/bsl/1182353926.

[19] David Thibodeau, Andrew Cave & Brigitte Pientka (2016): *Indexed Codata Types*. In Jacques Garrigue, Gabriele Keller & Eijiro Sumii, editors: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, ACM, pp. 351–363, doi:10.1145/2951913.2951929.

[20] Chun Tian & Davide Sangiorgi (2020): *Unique Solutions of Contractions, CCS, and their HOL Formalisation*. *Inf. Comput.* 275, p. 104606, doi:10.1016/J.IC.2020.104606.

[21] Alwen Tiu & Dale Miller (2010): *Proof Search Specifications of Bisimulation and Modal Logics for the π-Calculus*. *ACM Trans. Comput. Log.* 11(2), pp. 13:1–13:35, doi:10.1145/1656242.1656248.

[22] Niccolò Veltri & Andrea Vezzosi (2023): *Formalizing CCS and π-calculus in Guarded Cubical Agda*. *J. Log. Algebraic Methods Program.* 131, p. 100846, doi:10.1016/J.JLAMP.2022.100846.

[23] Daniel Zackon, Chuta Sano, Alberto Momigliano & Brigitte Pientka (2025): *Split Decisions: Explicit Contexts for Substructural Languages*. In Kathrin Stark, Amin Timany, Sandrine Blazy & Nicolas Tabareau, editors: *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2025, Denver, CO, USA, January 20-21, 2025*, ACM, pp. 257–271, doi:10.1145/3703595.3705888.

Proof Theory and Dependent Type Theory: Distinct Foundations for Designing Proof Assistants

Dale Miller
Inria Saclay and LIX, Institut Polytechnique de Paris

This paper examines the foundational distinctions between proof theory and dependent type theory (DTT) in the design of interactive theorem provers. While several implemented systems are designed using the dependently typed λ-calculus to represent proofs, no major proof assistant is designed using modern structural proof theory, even though, as I will argue here, the sequent calculus offers a compelling alternative framework. Six specific topics are proposed where the proof-theoretic perspective is arguably superior to the DTT perspective. These topics include the separation of logic from proof structure, the strategic use of non-determinism in proof reconstruction, and the avoidance of complex typing-discipline issues such as universe levels and proof irrelevance. The final topic—the treatment of bindings—is further developed to demonstrate how a natural, intensional approach is achieved through the mobility of binders. This methodology is illustrated via the Abella theorem prover, which leverages λ-tree syntax and the ∇-quantifier to provide an elegant environment for reasoning about the meta-theory of languages and logics involving complex binding.

1 Introduction

In 2006, Wiedijk published a small volume [57] that showed the formalization of the proof of the irrationality of $\sqrt{2}$ in the following seventeen interactive theorem provers:

> ACL2, Alfa/Agda, B Method, Coq, HOL, IMPS, Isabelle/Isar, Lego, Metamath, Mizar, Minlog, Nuprl, Ωmega, Otter/Ivy, PVS, PhoX, and Theorema.

These systems vary significantly in their underlying logical foundations. Some are based on first-order logic, while others accept higher-order quantification. Employing a range of proof structures, some of these systems use Frege-style proofs (involving axioms and a small set of inference rules) because these can support simple proof-checking kernels. Others take a more sophisticated approach to proof structures and employ either a natural deduction style à la Gentzen and Prawitz, or the dependently typed λ-calculus directly. The role of *types* in these systems varies greatly. Most of these systems allow at least multi-sorted, first-order quantification. Others implement Church's Simple Theory of Types and allow quantification over function and predicate types. In addition, certain predicates that have well-understood reasoning principles can often be recast as types, with the usual consideration of whether type checking and/or type inference are decidable for those predicates cum types. Some of these systems (and the more recent Lean system [45]), are based on *dependent type theory* (DTT), whose origin is often traced back to De Bruijn [8] and Martin-Löf [37].

In this paper, I argue that modern aspects of proof theory can offer an alternative perspective to the design narrative behind the use of dependently typed λ-calculus in proof assistants. I will assume here that the reader is familiar with the latter formalism but is not familiar with the many advances that have occurred within the general topic of structural proof theory. I will survey that topic in the next section.

Hermant, Tourret (Eds.): International Workshop on Logical Frameworks and Meta-Languages, Theory and Practice (LFMTP 26)
EPTCS 448, 2026, pp. 18–28, doi:10.4204/EPTCS.448.2

2 Structural proof theory

Structural proof theory was started by Gentzen [25] with his invention of natural deduction and the sequent calculus. In particular, Gentzen described his intentions about natural deduction:

> I intended first to set up a formal system which comes as close as possible to actual reasoning. The result is a *'calculus of natural deduction'*.

Thus, it is no surprise that this calculus of natural deduction has had important influences on various proof assistants.

Early in the introduction of [25], Gentzen justifies the introduction of the sequent calculus as a technical vehicle for stating and proving the *Hauptsatz* for *both* intuitionistic and classical logic. In the general area of computational logic, sequent calculus is often seen as a technical device for tracing the search for an actual proof. In fact, there is the old chestnut: "Natural deductions are proofs while sequent calculus proofs are the computation of a proof" (see, for example, [28, Section 5.4]).

As we shall see in this paper, the sequent calculus can be viewed, not only as a useful technical device, but as a central framework for developing and justifying possible proof assistant designs. It is also possible to list several successful deployments of the sequent calculus in domains other than proof system design.

- Gentzen used the sequent calculus to prove the consistency of Peano arithmetic [26].
- Various researchers, for example [32, 46], have used the sequent calculus to provide independence proofs in certain axioms with respect to other axioms within mathematical theories.
- Often, the most successful proof system for a modern logic, such as modal and linear logics, is given using the sequent calculus.
- Various logic programming languages have been developed using the sequent calculus in classical, intuitionistic, and linear logics [40].
- The design of at least one model checker has been built on a sequent calculus proof theory that accounts for fixed points [7, 30].

Of the many recent developments in the theory of the sequent calculus that might not be familiar to those working on designing and implementing proof systems, we can list the notions of *polarity* and *focusing* (both inspired by the sequent calculus of linear logic) and the notion of the *mobility of binders*. These topics will be expanded on in the following section.

3 A proof theorist's view of dependent type theory

In this section, I identify six topics where a proof-theoretic perspective offers distinct advantages over that of DTT. Naturally, one could conversely argue that DTT provides a superior framework in other areas; its strengths include mature implementations, widespread adoption, robust support for automated inference, and the ability to provide concise specifications in a variety of domains. My objective here is to articulate the proof-theoretic perspective, as it is seldom represented in the literature. Furthermore, by focusing on foundational issues, I relegate the current state of software implementation to a secondary concern.

3.1 Regarding the structure of proof

A given DTT usually settles two questions simultaneously:

1. Which logic is being formalized? This is typically intuitionistic logic.
2. What is a proof? These are typically natural deduction proofs encoded as dependently typed λ-terms.

A proof theorist separates these questions, thereby allowing for a possibly large variety of proof systems for a fixed logic. For example, when fixing the logic to be intuitionistic, proofs can be formalized using natural deduction, sequent calculus, and tableaux [21], and inference rules can be applied in a forward chaining as well as a backward chaining fashion. When fixing the logic to be classical logic, many more proof structures are possible, additionally including Herbrand disjunctions, resolution refutations, natural deduction with restart [22], etc. If the logic is fixed to be linear, then all the previous proof structures are possible, including various forms of proof nets.

Of course, once one has a solid implementation of, say, natural deduction proofs as dependently typed λ-terms, one has a highly expressive computational setting capable of encoding a rich collection of other proof systems, as witnessed by the many proof systems that have been encoded into Dedukti [4]. Still, there is something to be said for treating these other proof systems as structures in their own right rather than merely as encodings into some other proof structure.

3.2 Problems with the proof-as-λ-term approach

Another frequently claim is that the λ-calculus is a canonical computational model for sequential computation. While I am not arguing against that particular sentiment, it is worth noting that many things about the (untyped) λ-calculus are far from canonical. For example, while λ-reduction is a confluent operation on λ-terms, the paths to (weak) normal forms are highly non-deterministic, with different computational platforms adopting different reduction strategies (call-by-value, call-by-name, call-by-need, etc), and these can affect correctness (e.g., call-by-value may not terminate with a normal form while call-by-name does) as well as efficiency.

Dependently typed λ-terms also introduce many complications. For example, there are issues around proof irrelevance (too many subproofs kept), implicit arguments (too inconvenient to supply all arguments), and universe levels (needed to organize rich typing structures). Approaches to encoding logic that omit such a rich and complex typing discipline do not need to address these issues, at least not in a foundational setting. Also, proofs-as-λ-terms can be highly redundant (leading to, for example, explicitly and implicitly typed versions of LF [50, 54]) and they do not include any explicit structure-sharing mechanisms. The sequent calculus, on the other hand, provides a simple and natural explicit mechanism [43].

3.3 The mutual development of dependent type theory and proof theory

Many central topics in the study of proof theory and DTT were investigated around the same time and influenced each other.

1. The notion of normalizing proofs was first developed in the context of sequent calculus (via cut-elimination [25]) and natural deduction (via normalization [52, 53]). These normalization processes (including the elimination of non-atomic initial rules) yield the familiar notions of β and η-rules. Of course, these normalization processes were also studied in the untyped and simply

typed λ-calculus by Church also in the 1930s and 1940s [14, 15]. The introduction of these concepts into dependently typed λ-calculus waited until de Bruijn [8] and Martin-Löf [36] introduced their typed calculi.

2. Linear logic [27] appeared first as a development in proof theory and later moved to dependently typed λ-calculus shortly after (see, for example, [9, 31]).

3. The notion of dependently typed λ-terms in *canonical form* (as defined for LF in [29]) is closely related to the proof theoretic concept of *uniform proofs* (as shown in [19, 20]). Eventually, the linear logic notions of *polarization* and *focusing* [3] were generalized within the proof theory setting to classical and intuitionistic logics (see, for example, [16, 17, 35]). Focused proofs are now generally understood to yield useful notions of canonical normal forms of proofs [13, 39].

3.4 The dependent type theory approach to classical logic

Gentzen [25] abandoned natural deduction since he could not see how that style of proof could be used uniformly to yield the Hauptsatz for both classical and intuitionistic logic. His solution for a more uniform approach to the proof theory of both logics was a multiple-conclusion sequent calculus. Unfortunately, Gentzen's better approach is generally ruled out in DTT, at least in the type theories in use in proof assistants today. The gap between treating classical logic proofs as dependently typed λ-terms plus axioms, such as the excluded middle, and viewing classical logic proofs, such as say Herbrand disjunctions (expansion trees) or resolution refutation, seems huge. Of course, elaborate techniques have been developed to bridge this gap, at least, in principle: see, for example, [18, 33].

3.5 Non-determinism provides a valuable resource

The trusted proof-checking kernels of proof assistants based on dependently typed λ-calculus are generally functional programs that expect the proof structures they check to contain enough information to make checking completely deterministic. It is well known that non-determinism can be a valuable resource in designing and verifying certificates: just consider the P and NP complexity classes. Allowing non-determinism in the design of both proof certificates and proof checkers allows for important trade-offs between proof certificate size and checking time.

Having a kernel that allows non-determinism (via backtracking search) is certainly possible (as in logic-programming-based kernels). Example: Consider a proof certificate for the claim that the sequent $\Gamma \vdash A$ is an instant of the *initial* rule, which can only be the case when A is a member of Γ. On one hand, the certificate could be constant in size while the proof checking kernel would then need to search for A in Γ: something that could be accomplished with a non-deterministic search (made deterministic in actual deterministic kernel using search). On the other hand, the certificate could provide the exact name/label of the assumption in Γ that is also labeled with A. In this case, the certificate is no longer constant-sized, but its checking would be a more direct, deterministic process. Having this trade-off between proof certificate size and non-deterministic proof reconstruction seems desirable. A kernel for such checking is a rather direct combination of some form of (higher-order) unification and backtracking search. While incorporating such features into the trusted code base of a kernel will increase the complexity of such kernels, these features have been well-developed and exploited in proof systems written in the λProlog programming language [41] and the Isabelle prover [47] since the late 1980s.

3.6 Treatment of bindings

In most modern proof assistant based on DTT or first-order logic, no canonical treatment for bindings has appeared. Instead, a wide range of technical approaches to encoding bindings have been deployed and studied. In general, the specifications of bindings in syntax are still considered problematic in most conventional systems, given the existence of challenge problems (POPLMark [5], POPLMark reloaded [1]), case studies, benchmarks, and surveys. The Twelf [49] and Beluga [51] systems are exceptions since they offer a computing environment where computed values can be dependently typed λ-terms. However, of the proof assistants lists in Section 1, only Isabelle and Minlog offer direct support for bindings in data structures and the associated notion of (higher-order) unification on such structures.

The sequent calculus provides an elegant and powerful setting to specify and compute with syntactic objects containing bindings, something that, to date, has not been matched in the setting of dependently typed λ-calculi. The approach available in the sequent calculus is outlined in more detail in the following section.

4 Sequents and binders

By making a slight embellishment of Gentzen's formulation of sequents, we write sequents here as $\Sigma : \Gamma \vdash C$, where the collection of variables in Σ is interpreted as being bound over $\Gamma \vdash C$. The variables in Σ correspond to Gentzen's notion of *eigenvariables*.

To illustrate how binder can be used with sequent calculus proof rules, consider specifying the predicate *typeof*, which relates an encoding of untyped λ-terms (given by the function $\lceil\cdot\rceil$) with an encoding of simple types (using $\rightarrow$ for the function type constructor). Here, we assume as is customary in many encodings of λ-terms (see, [41], for example) that binders in the untyped λ-terms are mapped to bindings in the result of the encoding.

When reading inference rules from conclusion to premises (as one does to understand the impact of an inference rule during the search for a proof), binders can be *instantiated*, as in the following instance of the left-introduction for $\forall$.

$$\frac{\Sigma : \Gamma, \mathit{typeof}\ c\ (\mathit{int} \rightarrow \mathit{int}) \vdash C}{\Sigma : \Gamma, \forall\alpha(\mathit{typeof}\ c\ (\alpha \rightarrow \alpha)) \vdash C}\ \forall L$$

Here, the bound variable α is instantiated with the term *int*, a term denoting the integer type. Binders can also *move*, as illustrated by the following inference rules.

$$\frac{\dfrac{\Sigma, \mathbf{x} : \Gamma, \mathit{typeof}\ x\ \alpha \vdash \mathit{typeof}\ \lceil B\rceil\ \beta}{\Sigma : \Gamma \vdash \forall\mathbf{x}(\mathit{typeof}\ x\ \alpha \supset \mathit{typeof}\ \lceil B\rceil\ \beta)}\ \forall R, \supset R}{\Sigma : \Gamma \vdash \mathit{typeof}\ \lceil\lambda\mathbf{x}.B\rceil\ (\alpha \rightarrow \beta)}$$

Here, the lower inference rule is justified by some external definition of the predicate *typeof* (a familiar specification technique for defining typing dating back to the early days of λProlog and LF [20, 29]). Here, binders move from *term-level* (λx) to *formula-level* ($\forall x$) to *proof-level* (eigenvariable): that is, this bound variable never becomes free. Thus, *binder mobility* is one way to formally realize A. Perlis's Epigram 47: "There is no such thing as a free variable." [48]. A consequence of this approach to binders is that the technique for implementing binders (e.g., named variables, nameless dummies, etc) does not need to be revealed to the person writing the logical specification of, in this case, *typeof*: instead, binders never stop being bound.

Another aspect of binders in the proof-theory setting is that they do not need to represent function spaces: they can be viewed, in the appropriate logical setting, as abstractions over syntax. Consider, for example, the formula

$$\forall w.\ \lambda x.x \neq \lambda x.w,$$

where w and x are given the same primitive type. If one views the λ-binding as specifying a function (i.e., extensionally), then this formula is not a theorem since the identity and a constant-valued function can coincide on singleton domains. If one views the λ-binder as simply part of the syntax (up to α-conversion, of course), then this formula should be a theorem since no (capture avoiding substitution) instance of $\lambda x.w$ can be equal to $\lambda x.x$.

This approach to binders in which syntactic binders are movable is called the *λ-tree syntax* approach [38], and it is available in the λProlog programming language [41], the LF logical framework [29], and the Rocq plugin for Elpi [55].

The sequent calculus can support at least one additional binding site other than the prefixed Σ binder. In particular, we add a binding context to each occurrence of a formula in a sequent. Thus, instead of the sequent $\Sigma\colon B_1,\ldots,B_n \vdash B_0$, we have the more general

$$\Sigma\colon \sigma_1 \triangleright B_1,\ldots,\sigma_n \triangleright B_n \vdash \sigma_0 \triangleright B_0.$$

Here, σ_i is a list of distinct variables scoped over B_i. The expression $\sigma_i \triangleright B_i$ is called a *generic judgment*. In order to exploit these proof-level binding sites, a new formula-level binder is needed: this is the role of the ∇-quantifier, for which the left and right introduction rules are given as

$$\frac{\Sigma : (\sigma, x : \tau) \triangleright B, \Gamma \longrightarrow \mathscr{C}}{\Sigma : \sigma \triangleright \nabla_\tau x.B, \Gamma \longrightarrow \mathscr{C}} \qquad \text{and} \qquad \frac{\Sigma : \Gamma \longrightarrow (\sigma, x : \tau) \triangleright B}{\Sigma : \Gamma \longrightarrow \sigma \triangleright \nabla_\tau x.B}\ .$$

Because of the similarity of the left and right introduction rules, it follows easily that ∇ is self-dual: that is, both the sequents $\neg\nabla x.Bx \vdash \nabla x.\neg Bx$ and $\nabla x.\neg Bx \vdash \neg\nabla x.Bx$ are provable. Furthermore, ∇ around an equality judgment can simply reduce to an equality judgment around λ-terms: that is, $\nabla x.(t = s)$ is provable exactly when $\lambda x.t = \lambda x.s$ is provable. Thus, the formula $\forall w.\ \lambda x.x \neq \lambda x.w$, turns out to be logically equivalent to $\forall w.\neg(\lambda x.x = \lambda x.w)$, which can be proved in a suitable system (such as Abella) since the (higher-order) unification problem $(\lambda x.x = \lambda x.w)$ has no unifiers. For details on how ∇-quantification interacts with the propositional constants, the other quantifiers, and induction and coinduction, see [24, 42].

The Abella proof assistant [6, 23] was developed on proof theoretic principles and includes support for λ-tree syntax and the ∇-quantifier.

5 The Abella proof assistant

During the search for a proof in Abella, one works with collections of *goals*, which are displayed as

```
Variables: x1 ... xm
H1 : A1
...
Hn : An
============================
 C
```

where $x_1,\ldots,x_m$ are universally quantified variables, `H1`, ..., `Hn` are *hypothesis labels* that are each associated with a unique *hypothesis formula* drawn from $A_1,\ldots,A_n$ and C is a formula called the *conclusion*

of the goal. The variables and hypotheses comprise the *context* of the goal. Not only does Abella view such a goal as a sequent, in this case being,

$$x_1 : \tau_1, \ldots, x_m : \tau_m :: A_1, \ldots, A_n \vdash C,$$

but its tactics for advancing the search for a proof are understood using sequent calculus inference rules [11, 12]. Abella implements the ∇-quantifier and the structural principles behind the logical equivalences $(\nabla x \nabla y.B) \equiv (\nabla y \nabla x.B)$ and $(\nabla x.C) \equiv C$, provided x is not free in C. As a result of these equivalences, it is possible to treat the local binding signatures using an equivalent mechanism that relies on nominal constants [24] (as a result, the local signatures are not displayed explicitly in the Abella goal structure above).

Abella is well-suited for reasoning about the meta-theory of languages and logics, especially those involving binding. For example, various aspects of the meta-theory of the λ-calculus have had successful Abella specifications: see, for example, a formulation of the λ-cube [2], a mechanical formulation of higher-ranked polymorphic type inference [58], and a formalization of parts of Barendregt's theory of the lambda-calculus [34]. Specifications of the operational semantics of the π-calculus and its meta-theory can make significant use of Abella's support for binder mobility. For example, the difference between open and closed bisimulation for the π-calculus is a simple matter of switching between using intuitionistic logic and classical logic [42]. Similarly, the presence of the three quantifiers $\forall$, $\exists$, and ∇ provides an easy and elegant way to capture the large collection of modal operators that have been proposed to capture an array of properties of π-calculus [44, 56]. Additionally, rather direct treatment of bisimulation-up-to techniques for the π-calculus can also be captured [10]. The website for Abella, `https://abella-prover.org/`, contains several other example formulations using Abella.

6 Conclusion

In this paper, I have argued that modern structural proof theory, rooted in the sequent calculus, offers a robust and compelling foundational alternative to basing interactive theorem provers on the "proofs-as-terms" paradigm of DTT. I have offered six explicit topics for which the proof theory approach can be argued to provide a possibly deeper and more flexible design than that typically offered by dependently typed λ-calculus. The last of these topics—the treatment of binding in syntactic expressions—is perhaps most significant. In this setting, the proof-theoretic approach provides a natural treatment of bindings via binder mobility. By treating binders as movable abstractions over syntax rather than function spaces, we achieve an elegant framework for reasoning about the meta-theory of complex languages. Many of these proof-theoretic notions have been built into the Abella theorem prover: in particular, this prover makes available the λ-tree syntax approach to bindings and the ∇-quantifier.

Acknowledgments I thank the reviewers for their helpful comments and perspectives on an earlier draft of this paper.

References

[1] Andreas Abel, Guillame Allais, Aliya Hameer, Alberto Momigliano, Brigitte Pientka, Steven Schaefer & Kathrin Stark (2019): *POPLMark Reloaded: Mechanizing Proofs by Logical Relations*. *Journal of Functional Programming* 29, doi:10.1017/S0956796819000170.

[2] Beniamino Accattoli (2012): *Proof pearl: Abella Formalization of λ-Calculus Cube Property*. In Chris Hawblitzel & Dale Miller, editors: *Second International Conference on Certified Programs and Proofs, LNCS* 7679, Springer, pp. 173–187, doi:10.1007/978-3-642-35308-6_15.

[3] Jean-Marc Andreoli (1992): *Logic Programming with Focusing Proofs in Linear Logic*. *J. of Logic and Computation* 2(3), pp. 297–347, doi:10.1093/logcom/2.3.297.

[4] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant & Ronan Saillard (2023): *Dedukti: a Logical Framework based on the λΠ-Calculus Modulo Theory*. Technical Report, arXiv, doi:10.48550/ARXIV.2311.07185.

[5] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich & Steve Zdancewic (2005): *Mechanized Metatheory for the Masses: The POPLmark Challenge*. In: *Theorem Proving in Higher Order Logics: 18th International Conference, LNCS* 3603, Springer, pp. 50–65, doi:10.1007/11541868_4.

[6] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu & Yuting Wang (2014): *Abella: A System for Reasoning about Relational Specifications*. *J. of Formalized Reasoning* 7(2), pp. 1–89, doi:10.6092/issn.1972-5787/4650.

[7] David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur & Alwen Tiu (2007): *The Bedwyr System for Model Checking Over Syntactic Expressions*. In F. Pfenning, editor: *21th Conf. on Automated Deduction (CADE), LNAI* 4603, Springer, New York, pp. 391–397, doi:10.1007/978-3-540-73595-3_28.

[8] N. G. de Bruijn (1980): *A Survey of the Project AUTOMATH*. In J. P. Seldin & R. Hindley, editors: *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, Academic Press, New York, pp. 589–606, doi:10.1016/S0049-237X(08)70203-9.

[9] Iliano Cervesato & Frank Pfenning (1996): *A Linear Logic Framework*. In: *11th Symp. on Logic in Computer Science*, IEEE Computer Society Press, New Brunswick, New Jersey, pp. 264–275, doi:10.1109/LICS.1996.561339.

[10] Kaustuv Chaudhuri, Matteo Cimini & Dale Miller (2015): *A Lightweight Formalization of the Metatheory of Bisimulation-Up-To*. In Xavier Leroy & Alwen Tiu, editors: *Proceedings of the 4th ACM-SIGPLAN Conference on Certified Programs and Proofs*, ACM, Mumbai, India, pp. 157–166, doi:10.1145/2676724.2693170. Available at `https://hal.inria.fr/hal-01091524/document`.

[11] Kaustuv Chaudhuri, Arunava Gantait & Dale Miller (2025): *Designing a Safe Forward Chaining Tactic Using Productive Proofs*. In G. L. Pozzato & T. Uustalu, editors: *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX), LNAI* 15980, Springer, pp. 299–317, doi:10.1007/978-3-032-06085-3_16.

[12] Kaustuv Chaudhuri, Ulysse Gérard & Dale Miller (2018): *Computation-as-Deduction in Abella: Work in Progress*. In: *13th international Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, Oxford, United Kingdom. Available at `https://hal.inria.fr/hal-01806154`.

[13] Kaustuv Chaudhuri, Dale Miller & Alexis Saurin (2008): *Canonical Sequent Proofs via Multi-Focusing*. In G. Ausiello, J. Karhumäki, G. Mauri & L. Ong, editors: *Fifth International Conference on Theoretical Computer Science, IFIP* 273, Springer, pp. 383–396, doi:10.1007/978-0-387-09680-3_26.

[14] Alonzo Church (1932): *A Set of Postulates for the Foundation of Logic*. *Annals of Mathematics* 33(2), pp. 346–366, doi:10.2307/1968337.

[15] Alonzo Church (1941): *The Calculi of Lambda-Conversion*. Princeton University Press.

[16] V. Danos, J.-B. Joinet & H. Schellinx (1995): *LKQ and LKT: Sequent Calculi for Second Order Logic Based Upon Dual Linear Decompositions of Classical Implication*. In J.-Y. Girard, Y. Lafont & L. Regnier, editors: *Advances in Linear Logic, London Mathematical Society Lecture Note Series* 222, Cambridge University Press, pp. 211–224, doi:10.1017/CBO9780511629150.

[17] R. Dyckhoff & S. Lengrand (2006): *LJQ: a Strongly Focused Calculus for Intuitionistic Logic*. In A. Beckmann & *et al.*, editors: *Computability in Europe 2006, LNCS* 3988, Springer, pp. 173–185, doi:10.1007/11780342_19.

[18] Gabriel Ebner & Matthias Schlaipfer (2018): *Efficient Translation of Sequent Calculus Proofs Into Natural Deduction Proofs*. In Boris Konev, Josef Urban & Philipp Rümmer, editors: *Proceedings of the 6th Workshop on Practical Aspects of Automated Reasoning co-located with Federated Logic Conference 2018 (FLoC 2018), Oxford, UK, July 19th, 2018*, *CEUR Workshop Proceedings* 2162, CEUR-WS.org, pp. 17–33. Available at `https://ceur-ws.org/Vol-2162/paper-03.pdf`.

[19] Amy Felty (1991): *Encoding Dependent Types in an Intuitionistic Logic*. In Gérard Huet & Gordon D. Plotkin, editors: *Logical Frameworks*, Cambridge University Press, pp. 215–251. Available at `https://inria.hal.science/inria-00075041v1`.

[20] Amy Felty & Dale Miller (1990): *Encoding a Dependent-Type λ-Calculus in a Logic Programming Language*. In Mark Stickel, editor: *Proceedings of the 1990 Conference on Automated Deduction*, *LNAI* 449, Springer, pp. 221–235, doi:10.1007/3-540-52885-7_90.

[21] Melvin Fitting (1983): *Proof Methods for Modal and Intuitionistic Logics*. D. Reidel, Dordrecht, The Netherlands, doi:10.1007/978-94-017-2794-5.

[22] D. M. Gabbay & U. Reyle (1984): *N-Prolog: An Extension of Prolog with Hypothetical Implications. I*. *Journal of Logic Programming* 1, pp. 319–355, doi:10.1016/0743-1066(84)90029-3.

[23] Andrew Gacek (2008): *The Abella Interactive Theorem Prover (System Description)*. In A. Armando, P. Baumgartner & G. Dowek, editors: *Fourth International Joint Conference on Automated Reasoning*, *LNCS* 5195, Springer, pp. 154–161, doi:10.1007/978-3-540-71070-7_13.

[24] Andrew Gacek, Dale Miller & Gopalan Nadathur (2011): *Nominal abstraction*. *Information and Computation* 209(1), pp. 48–73, doi:10.1016/j.ic.2010.09.004.

[25] Gerhard Gentzen (1935): *Investigations into Logical Deduction*. In M. E. Szabo, editor: *The Collected Papers of Gerhard Gentzen*, North-Holland, Amsterdam, pp. 68–131, doi:10.1007/BF01201353. Translation of articles that appeared in 1934-35. Collected papers appeared in 1969.

[26] Gerhard Gentzen (1969): *The Consistency of Elementary Number Theory*. In M. E. Szabo, editor: *The Collected Papers of Gerhard Gentzen*, North-Holland, Amsterdam, pp. 132–213, doi:10.1016/S0049-237X(08)70823-1. Translated from the 1936 German original.

[27] Jean-Yves Girard (1987): *Linear Logic*. *Theoretical Computer Science* 50(1), pp. 1–102, doi:10.1016/0304-3975(87)90045-4.

[28] Jean-Yves Girard, Paul Taylor & Yves Lafont (1989): *Proofs and Types*. Cambridge University Press.

[29] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A Framework for Defining Logics*. *Journal of the ACM* 40(1), pp. 143–184, doi:10.1145/138027.138060.

[30] Quentin Heath & Dale Miller (2019): *A proof theory for model checking*. *J. of Automated Reasoning* 63(4), pp. 857–885, doi:10.1007/s10817-018-9475-3.

[31] Martin Hofmann (2003): *Linear Types and Non-Size Increasing Polynomial Time Computation*. *Information and Computation* 183(1), pp. 57–85, doi:10.1016/S0890-5401(03)00009-9.

[32] Oiva Ketonen (2022): *Investigations into the Predicate Calculus*. College Publications, doi:10.1007/S11225-024-10123-3. Ed. by S. Negri and J. von Plato.

[33] Anja Petković Komel, Michael Rawson & Martin Suda (2025): *Case Study: Verified Vampire Proofs in the LambdaPi-calculus Modulo*. *arXiv*, doi:10.48550/arxiv.2503.15541.

[34] Adrienne Lancelot, Beniamino Accattoli & Maxime Vemclefs (2025): *Barendregt's Theory of the lambda-Calculus, Refreshed and Formalized*. In: *16th International Conference on Interactive Theorem Proving (ITP 2025)*, *LIPIcs* 352, pp. 13:1–13:22, doi:10.4230/LIPIcs.ITP.2025.13.

[35] Chuck Liang & Dale Miller (2009): *Focusing and Polarization in Linear, Intuitionistic, and Classical Logics*. *Theoretical Computer Science* 410(46), pp. 4747–4768, doi:10.1016/j.tcs.2009.07.041. Abstract Interpretation and Logic Programming: A Special Issue in Honor of Professor Giorgio Levi.

[36] Per Martin-Löf (1982): *Constructive Mathematics and Computer Programming*. In: *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, North-Holland, Amsterdam, pp. 153–175, doi:10.1016/S0049-237X(09)70189-2.

[37] Per Martin-Löf (1984): *Intuitionistic Type Theory*. Studies in Proof Theory Lecture Notes, Bibliopolis, Napoli.

[38] Dale Miller (2019): *Mechanized Metatheory Revisited*. *Journal of Automated Reasoning* 63(3), pp. 625–665, doi:10.1007/s10817-018-9483-3.

[39] Dale Miller (2025): *Linear logic using negative connectives*. In Maribel Fernández, editor: *10th International Conference on Formal Structures for Computation and Deduction (FSCD 2025)*, LIPIcs, pp. 29:1–29:22, doi:10.4230/LIPIcs.FSCD.2025.29.

[40] Dale Miller (2025): *Proof Theory and Logic Programming: Computation as Proof Search*. Cambridge University Press, doi:10.1017/9781009561280. Preprint available at `https://www.lix.polytechnique.fr/Labo/Dale.Miller/ptlp/`.

[41] Dale Miller & Gopalan Nadathur (2012): *Programming with Higher-Order Logic*. Cambridge University Press, doi:10.1017/CBO9781139021326.

[42] Dale Miller & Alwen Tiu (2005): *A proof theory for generic judgments*. *ACM Trans. on Computational Logic* 6(4), pp. 749–783, doi:10.1145/1094622.1094628.

[43] Dale Miller & Jui-Hsuan Wu (2023): *A positive perspective on term representations*. In Bartek Klin & Elaine Pimentel, editors: *31st EACSL Annual Conference on Computer Science Logic (CSL 2023)*, *LIPIcs* 252, Dagstuhl, Germany, pp. 3:1–3:21, doi:10.4230/LIPIcs.CSL.2023.3.

[44] Robin Milner, Joachim Parrow & David Walker (1993): *Modal Logics for Mobile Processes*. *Theoretical Computer Science* 114(1), pp. 149–171, doi:10.1016/0304-3975(93)90156-N.

[45] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn & Jakob von Raumer (2015): *The Lean Theorem Prover (System Description)*. In Amy P. Felty & Aart Middeldorp, editors: *25th International Conference on Automated Deduction (CADE-25)*, *Lecture Notes in Computer Science* 9195, Springer, pp. 378–388, doi:10.1007/978-3-319-21401-6_26.

[46] Sara Negri & Jan von Plato (2011): *Proof Analysis: A Contribution to Hilbert's Last Problem*. Cambridge University Press, Cambridge, doi:10.1017/CBO9780511921629.

[47] Lawrence C. Paulson (1986): *Natural Deduction as Higher-Order Resolution*. *Journal of Logic Programming* 3, pp. 237–258, doi:10.17863/CAM.23642.

[48] Alan J. Perlis (1982): *Epigrams on Programming*. *ACM SIGPLAN Notices* 17(9), pp. 7–13, doi:10.1145/947955.1083808.

[49] Frank Pfenning & Carsten Schürmann (1999): *System Description: Twelf — A Meta-Logical Framework for Deductive Systems*. In H. Ganzinger, editor: *16th Conf. on Automated Deduction (CADE)*, *LNAI* 1632, Springer, Trento, pp. 202–206, doi:10.1007/3-540-48660-7_14.

[50] Brigitte Pientka (2013): *An insider's look at LF type reconstruction: everything you (n)ever wanted to know*. *Journal of Functional Programming* 23(1), pp. 1–37, doi:10.1017/S0956796812000408.

[51] Brigitte Pientka & Joshua Dunfield (2010): *Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)*. In J. Giesl & R. Hähnle, editors: *Fifth International Joint Conference on Automated Reasoning*, *LNCS* 6173, pp. 15–21, doi:10.1007/978-3-642-14203-1_2.

[52] Jan von Plato (2008): *Gentzen's proof of normalization for natural deduction*. *Bulletin of Symbolic Logic* 14(2), pp. 240–257, doi:10.2178/bsl/1208442829.

[53] Dag Prawitz (1965): *Natural Deduction*. Almqvist & Wiksell, Uppsala.

[54] Jason Reed (2008): *Redundancy Elimination for LF*. *Electronic Notes in Theoretical Computer Science* 199, pp. 89–106, doi:10.1016/j.entcs.2007.11.014.

[55] Enrico Tassi (2026): *Elpi: rule-based extension language*. Habilitation à diriger des recherches, Université Côte d'Azur, Valbonne, France. Available at `http://www-sop.inria.fr/members/Enrico.Tassi/hdr/hdr.pdf`.

[56] Alwen Tiu & Dale Miller (2010): *Proof Search Specifications of Bisimulation and Modal Logics for the π-calculus*. *ACM Trans. on Computational Logic* 11(2), pp. 13:1–13:35, doi:10.1145/1656242.1656248.

[57] Freek Wiedijk, editor (2006): *The Seventeen Provers of the World*. *LNCS* 3600, Springer.

[58] Jinxu Zhao, Bruno C. d. S. Oliveira & Tom Schrijvers (2018): *Formalization of a Polymorphic Subtyping Algorithm*. In: *International Conference on Interactive Theorem Proving (ITP 2018)*, *LNCS* 10895, pp. 604–622, doi:10.1007/978-3-319-94821-8_36.

Anti-Unification Completeness Analysis in PVS

Mauricio Ayala-Rincón
Universidade de Brasília and Universidade Federal de Goiás
Exact Sciences Institute and Institute of Mathematics and Statistics
Brasília D.F. and Goiânia, Brazil

Thaynara Arielly de Lima
Universidade Federal de Goiás
Institute of Mathematics and Statistics
Goiânia, Brazil

Maria Júlia Dias Lima
Universidade de Brasília
Graduate Program in Informatics
Brasília D.F., Brazil

Temur Kutsia
Johannes Kepler Universität
Research Institute for
Symbolic Computation
Linz, Austria

Marcos Mercandeli-Rodrigues
Universidade de Brasília
Graduate Program in Mathematics
Brasília D.F., Brazil

In syntactic anti-unification, one is concerned with finding the commonalities between terms, while (uniformly) abstracting their differences. The original goal of anti-unification development in the seventies was to automate inductive reasoning. Recent applications of anti-unification techniques include efficiently transforming sequential code into parallel code, detecting code clones, and preventing software failures. Previous work addressed the elements required to verify, in the Prototype Verification System (PVS), termination and soundness of a functional algorithm based on inference rules for syntactic anti-unification. This paper dissects all aspects required to formally establish the completeness of the rule-based algorithm, highlighting the significant differences in the formalizations of anti-unification and unification.

Keywords: Equational Reasoning, Anti-unification, Generalization, Computational Verification, Interactive Theorem Proving, PVS.

1 Introduction

Motivation and Contextualization

Generalization is the scientific principle of drawing broad conclusions from specific observations, aiming to identify patterns or laws that hold beyond the original cases studied. In logic, this idea of generalization is formalized by a concept called *anti-unification*. It consists of finding the common structure and uniformly abstracting over differences when comparing two objects of a certain type, and it was first investigated and developed independently in the 1970s by Plotkin [33] and Reynolds [35].

Given terms s and t, one says that s is *more general* than t (or that t is *more specific* than s) and writes $s \preceq t$ iff there exists a substitution σ such that $s\sigma = t$. The relation $\preceq$ is known as *instantiation preorder* [16]. A *generalizer* for two terms s and t is a term r such that $r \preceq s$ and $r \preceq t$. The $\preceq$-minimal generalizers for s and t are called *least general generalizers* for s and t. Thus, the syntactic anti-unification problem can be formally stated as: For terms s and t, construct their least general generalizers. It is known that syntactic anti-unification is of type unitary [22, 29], i.e., any instance of the problem has a unique solution (up to renaming). For instance, the terms X, $f(X,Y)$, $f(g(X,Y),Z)$, and $f(g(X,Y),X)$ are generalizers for $f(g(c,d),c)$ and $f(g(g(u,v),v),g(u,v))$, but only $f(g(X,Y),X)$ is their least general generalizer.

The first procedural algorithms for solving the syntactic anti-unification problem were presented independently by Plotkin [33] and Reynolds [35] in the 1970s. Their algorithm essentially reads the pair of input expressions, identifies the smallest positions where conflicts occur, and uniformly assigns fresh

Hermant, Tourret (Eds.): International Workshop on Logical Frameworks
and Meta-Languages, Theory and Practice (LFMTP 26)
EPTCS 448, 2026, pp. 29–46, doi:10.4204/EPTCS.448.3

variables to those positions [35]. That allows the construction of the least general generalizer as being the join in a certain complete non-modular lattice whose order is $\succeq$ [35]. Huet presented an algorithm in 1976 in [28] by means of a recursive function λ that acts in the form $\lambda(s,t) = f(\lambda(s_1,t_1),\ldots,\lambda(s_m,t_m))$ if $s = f(s_1,\ldots,s_m)$ and $t = f(t_1,\ldots,t_m)$ and $\lambda(s,t) = \phi(s,t)$ otherwise, where ϕ is a bijection between pairs of terms and variables (ensuring that λ solves conflicts uniformly). A good exposition of that algorithm is presented by Lassez et al. in [29].

Rule-based procedures for anti-unification were presented by Pfenning [32] for the Calculus of Constructions, by Alpuente et al. [3] for order-sorted generalization, by the same authors for anti-unification modulo associativity and commutativity [4], and by Baumgartner et al. [20] for simply-typed lambda-terms in η-long β-normal forms, among others. Alpuente et al.'s proofs of completeness were non-constructive and relied on the anti-unification type; in the syntactic case, for instance, on the fact that syntactic anti-unification is unitary. A more preferred proof of completeness for rule-based algorithms would be constructive, as the one presented in this paper.

Real-world applications of anti-unification are, for instance, the identification of regularities in sequential code to transform it into efficient parallel code [18]; preventing failures and detecting errors in software [30]; repairing software bugs [17,24,39]; detecting code cloning and plagiarism [21,38]; maintaining mathematical or software libraries [27,34]; detecting similarities among chemical compounds to infer their carcinogenicity [5]. An open-source library of anti-unification algorithms (implemented in Java) for first- and second-order unranked terms, higher-order patterns, and nominal terms was presented in [19]. The paper [22] surveys the state-of-the-art in theory and applications of anti-unification.

Surprisingly, the current interest in the practical applications of anti-unification techniques has not yet been followed by the development of formal, mechanically verified certificates in proof assistants. That situation contrasts with the case for unification, where one aims to identify two expressions. For instance, successful work related to the formalization of unification and matching in interactive theorem provers are the ones developed in [23] for AC-matching (in Coq), in [8] for nominal C-matching through unification with protected variables (also in Coq), in [10] for nominal unification, in [13] for nominal C-unification, in [11] for nominal AC-matching, and in [14] for AC-unification (all in PVS). Advancing this line of research to provide formal certificates for proof assistants for anti-unification techniques and to strike a balance between theoretical development and practical application is of utmost importance. The only formalization of an anti-unification algorithm known to date is due to [9] for syntactic anti-unification. In that work, an algorithm for syntactic anti-unification is specified in PVS, and its termination and soundness are mechanically verified. Only the formal certificate for completeness remained to be constructed, and providing such a construction is the goal of the current work. Completing that work is of utmost importance, for it will allow the extraction of certified executable code of a sound and complete algorithm for syntactic anti-unification.

This paper formally studies the additional requirements that are essential for proving the completeness of algorithms based on the standard anti-unification inference rules (see Figure 1) that appear in [9]. The main motivation of this study is to understand in detail the greater complexity of proofs required for anti-unification compared to those for formalizations of unification algorithms (e.g., [6, 36, 37]). In [9], for formalizing soundness, important differences were highlighted, particularly in the cases of detection of *solved* and *syntactically equal* problems. In anti-unification, two crucial aspects concern the treatment of such problems. The former corresponds to equational questions between terms headed by different (function) symbols. In contrast, the latter corresponds to trivial (i.e., syntactically equal atomic) problems in which the terms are either the same constant or the same variable. Surprisingly, most of the formalization work (more than 90%, as quantified in [9]) for verifying the anti-unification algorithm was devoted to these two kinds of apparently straightforward cases. For obtaining a formal proof, the rigor-

ous analysis of these cases requires more elaboration than the one usually devoted to the completeness of anti-unification in pen-and-paper proofs (e.g., [1, 28, 33, 35]).

In unification, if a problem is what we call a *solved* problem above in the context of anti-unification, one has a *failure* because terms with different head function symbols can not be unified; and syntactically equal problems are trivially resolved using the identity substitution. In anti-unification, the situation is completely different. A solved equation does not mean a failure, but the detection of a difference in the structure of the terms being compared. That difference is recorded in the computed substitution of the current configuration. In fact, unlike unification, anti-unification never fails. To solve the general case of anti-unification, one should proceed by further checking for other occurrences of the same difference, thereby detecting all possible regularities.

In anti-unification, the decomposition of the problem is guided by mutually distinct labels associated with subproblems of the input problem and by a substitution that records the problem's structure. As soon as a solved subproblem is detected, it is stored separately. The essential elements for addressing solved and syntactic equations for formalizing soundness in [9] include preservation properties for the sets of labels of the unsolved and solved subproblems, and for the variables in the domain and the image of the computed substitution.

Main Contribution

- For the formalization of completeness, we discuss why the preservation properties developed for soundness together with the adjustments of generalizer notions (Subsection 3.2) are enough to address the inference rules for decomposition.
- Although the required notions for dealing with the cases of solved and syntactic rules in the soundness proof were much more elaborate than those required for unification, these notions were not enough for formalizing anti-unification completeness. We introduce the required additional elements in notions such as arbitrary generalizers so that a series of invariance and preservation properties are guaranteed; among them, properties that assure that the steps of the algorithm do not change anti-unification problems, and that the partially computed solutions can always be refined into generalizers that are less general than any arbitrary generalizer (Subsection 3.1). Based on these properties, a restricted notion of generalizer is introduced (Subsection 3.2).
- Finally, using the restricted notion of generalizer, we formalize (Subsection 3.3) the completeness theorem (stated as Theorem 20) and obtain as a corollary the main result that states the completeness of the algorithm regarding arbitrary generalizers (Corollary 21).

Organization

Section 2 presents the required background following standard nomenclature on anti-unification, which is the one used in the formalization [9]. Section 3 is the kernel of the paper. Subsection 3.1 presents the additional notions required to record preservation and invariance properties that compile the history of the algorithm's computation. Subsection 3.2 presents the adjustments required to address completeness; then, Subsection 3.3 discusses how solved and syntactic subproblems are treated. Subsection 3.4 discusses the analysis of decomposition rules. Finally, Section 4 concludes and briefly discusses future work. Several points in the paper include links to the specification (PDF). An extended version of this paper presents quantitative information on a direct approach for dealing with the decomposition inference rules.

2 Background

The background related to terms, substitutions, and configurations is present in [9] and will be restated here to provide the proper vocabulary. The set of *terms* is generated by the standard nominal grammar $s,t ::= c \mid X \mid () \mid (s,t) \mid f\,s$ adapted from [12], where c stands for *constants*, X stands for *variables*, s and t stand for terms, (s,t) stands for *pairs* of terms, $()$ stands for the *unit*,[1] f stands for *function symbols*, and $f\,s$ stands for *functional applications*. It is specified in PVS by means of the abstract data type (ADT) `first_order_term`, requiring types for constants, function symbols, and variables as parameters. A *basic substitution* is a binding $(X \mapsto t)$, where X is a variable and t is a term. A *substitution* is a finite list of basic substitutions (the *identity substitution* is the empty list ι). Those are specified in PVS in the theory `first_order_substitution` using pairs and lists of these pairs. The action of a substitution on a term is standard. The domain of a substitution σ is denoted by $\texttt{dom}(\sigma)$. The set of variables in its range is denoted by $\texttt{rvars}(\sigma)$. The substitutions recognized by the predicate `nice?` defined in the previous theory are the most important and widely employed in the specification.

Example 1 (Niceness)**.** *A substitution* $(X_1 \mapsto t_1)\cdots(X_m \mapsto t_m)$ *is* nice *iff* $\texttt{vars}(t_i) \cap \{X_i,\ldots,X_m\} = \emptyset$ *and* $X_i \neq X_j$ *for all* $i,j = 1,\ldots,m$ *with* $i \neq j$*. For instance,* $(X \mapsto (Y,Z))\,(W \mapsto fX)$ *is nice, while* $(X \mapsto fX)$ *is not.*

An anti-unification problem is encoded by an *anti-unification triple* (*AUT*), which is an expression of the form $s \triangleq_X t$, where s and t are terms, called its *left-* and *right-hand sides*, respectively, and X is a fresh variable for s and t called its *label*. That structure is specified in PVS by means of a record type `AUT`, where record accessors for label and left- and right-hand sides are present. The expression eq_X will be used to denote an AUT whose label is the variable X. Its left- and right-hand sides will be denoted by $\texttt{lhs}(eq_X)$ and $\texttt{rhs}(eq_X)$, respectively. To solve an anti-unification problem, one must compare two terms and decompose their common structure. That motivates an *AUT classification* that allows no superposition: An AUT eq_X is *decomposable* iff $\texttt{lhs}(eq_X)$ and $\texttt{rhs}(eq_X)$ are either both pairs or both functional applications headed by the same function symbol; it is *trivial* iff $\texttt{lhs}(eq_X) = \texttt{rhs}(eq_X)$ is the unit, a constant or a variable; and it is *solved* in any other case. That AUT classification guides the design of the inference rules presented in Figure 1 and, in PVS, it is specified by means of unary predicates `match_DecF?`, `match_DecP?`, `match_Synt?`, and `match_Sol?` that checks whether or not an AUT matches the application of a rule.

During the decomposition, the labels are the most general solutions to anti-unification (sub)problems. Since those subproblems occur in different positions in the common structure of terms, it makes sense to label new subproblems with fresh labels to distinguish them. The decomposition produces a finite number of AUTs that are described as a finite collection. In PVS, that collection is specified as a list of AUTs. To a finite collection of AUTs A, one associates two sets whose members are variables that play completely distinct roles in the anti-unification algorithm. The first one is its *set of labels* $\texttt{lbls}(A)$, which collects all the labels of the members of A. The second one is its *set of variables* $\texttt{vars}(A)$, which collects all the variables that occur in the left- and right-hand sides of its members. To refer to a collection of subproblems obtained from a problem consistently, it is necessary to define a *valid set of AUTs* as being a finite set A of AUTs such that $\texttt{lbls}(A) = |A|$ and $\texttt{vars}(A) \cap \texttt{lbls}(A) = \emptyset$. In PVS, valid sets of AUTs were specified by means of valid lists of AUTs, which are recognized by the unary predicate `valid_AUTs?`.

[1] The unit is useful for term flattening and for efficiently encoding finitary and variadic functions in the presence of pairs. For instance, considering addition $+\colon \mathbb{N} \to \mathbb{N}$, one can define $\sum\colon \texttt{list}(\mathbb{N}) \to \mathbb{N}$ recursively by $\sum() := 0$ and $\sum(m,L) := m + \sum L$.

As decomposition proceeds, it may be necessary to address an anti-unification problem that was previously handled but at a distinct position in the common structure. Such problems are essentially the same but encoded by AUTs that differ only in their labels. Those are called *repeated AUTs*. That can be extended to sets of AUTs: An AUT eq_X is *repeated in* a set A of AUTs iff there exists an AUT eq_Y in A such that eq_X and eq_Y are repeated. The binary predicates `repeated_AUT?` and `AUT_repeated_in?` are specified to recognize such repetitions.

$$\text{Decompose-Function (DecF)}\ \frac{\langle f\,s \triangleq_X f\,t, U \mid S \mid \sigma\rangle}{\langle s \triangleq_Y t, U \mid S \mid \sigma\,(X \mapsto f\,Y)\rangle}$$

$$\text{Decompose-Pair (DecP)}\ \frac{\langle (s,u) \triangleq_X (t,v), U \mid S \mid \sigma\rangle}{\langle s \triangleq_Y t, u \triangleq_Z v, U \mid S \mid \sigma\,(X \mapsto (Y,Z))\rangle}$$

$$\text{Solve-Repeated (SolR)}\ \frac{\langle s \triangleq_X t, U \mid S \mid \sigma\rangle}{\langle U \mid S \mid \sigma\,(X \mapsto X')\rangle}\ \text{if } s \triangleq_X t \text{ is solved and } s \triangleq_{X'} t \in S$$

$$\text{Solve-Non-Repeated (SolNR)}\ \frac{\langle s \triangleq_X t, U \mid S \mid \sigma\rangle}{\langle U \mid s \triangleq_X t, S \mid \sigma\rangle}\ \text{if } s \triangleq_X t \text{ is solved and not repeated in } S$$

$$\text{Syntactic (Synt)}\ \frac{\langle s \triangleq_X s, U \mid S \mid \sigma\rangle}{\langle U \mid S \mid \sigma\,(X \mapsto s)\rangle}\ \text{if } s \triangleq_X s \text{ is trivial}$$

Figure 1: Standard anti-unification inference rules.

During the decomposition of the common term structure, one must understand the current state and the states visited to reach it. In that way, the problems already computed and those remaining to be computed will be identified. That is encoded by means of a *valid configuration*, which is an expression of the form $\langle \mathcal{C}_{\mathtt{Uns}} \mid \mathcal{C}_{\mathtt{Sol}} \mid \mathcal{C}_{\mathtt{Sub}}\rangle$, where $\mathcal{C}_{\mathtt{Sub}}$ is a substitution, $\mathcal{C}_{\mathtt{Uns}}$ and $\mathcal{C}_{\mathtt{Sol}}$ are valid sets of AUTs called its *computed substitution* and *unsolved* and *solved* parts, respectively, and each of those three components satisfy the following constraints:

1. The sets $\mathcal{C}_{\mathtt{Uns}}$ and $\mathcal{C}_{\mathtt{Sol}}$ are disjoint and $\mathcal{C}_{\mathtt{Uns}} \cup \mathcal{C}_{\mathtt{Sol}}$ is a valid set of AUTs;

2. The set $\mathcal{C}_{\mathtt{Sol}}$ contains only solved AUTs that are not repeated in $\mathcal{C}_{\mathtt{Sol}}$;

3. The sets $\mathtt{lbls}(\mathcal{C}_{\mathtt{Uns}})$, $\mathtt{lbls}(\mathcal{C}_{\mathtt{Sol}})$, and $\mathtt{dom}(\mathcal{C}_{\mathtt{Sub}})$ are pairwise disjoint. The sets $\mathtt{dom}(\mathcal{C}_{\mathtt{Sub}})$ and $\mathtt{rvars}(\mathcal{C}_{\mathtt{Sub}})$ are also disjoint.

The inference rules in Figure 1 induce a reduction relation over valid configurations, which is denoted as $\Rightarrow$ using standard rewriting notation [15]; thus, $\Rightarrow^*$, $\Rightarrow^+$, and $\Rightarrow^n$ denote the reflexive transitive closure, transitive closure, and the n-reduction steps derivation relations obtained from $\Rightarrow$, respectively. In specific derivations, superscripts with the names of the applied rules are added to the relation symbol $\Rightarrow$. For the example in the introduction, we have the following derivation:

$$\langle f\,(g\,(c,d),c) \triangleq_X f\,(g\,(g\,(u,v),v),g\,(u,v)) \mid \emptyset \mid \iota\rangle \xrightarrow{\texttt{DecF},\texttt{DecP}}$$
$$\langle g\,(c,d) \triangleq_Y g\,(g\,(u,v),v),c \triangleq_Z g\,(u,v) \mid \emptyset \mid X \mapsto f\,(Y,Z)\rangle \xrightarrow{\texttt{DecF},\texttt{DecP}}$$
$$\langle c \triangleq_{Y_1} g\,(u,v),d \triangleq_{Y_2} v,c \triangleq_Z g\,(u,v) \mid \emptyset \mid X \mapsto f\,(g\,(Y_1,Y_2),Z)\rangle \xrightarrow{\texttt{SolNR},\texttt{SolNR}}$$
$$\langle c \triangleq_Z g\,(u,v) \mid c \triangleq_{Y_1} g\,(u,v),d \triangleq_{Y_2} v \mid X \mapsto f\,(g\,(Y_1,Y_2),Z)\rangle \xrightarrow{\texttt{SolR}}$$
$$\langle \emptyset \mid c \triangleq_{Y_1} g\,(u,v),d \triangleq_{Y_2} v \mid X \mapsto f\,(g\,(Y_1,Y_2),Y_1)\rangle$$

A configuration is specified in PVS by means of `Configuration`, which is a record type with record accessors for unsolved, solved, and computed substitution (always a nice one). Among configurations, the valid ones are recognized by the unary predicate `validConfiguration?`. The intuition for each component of a valid configuration is as follows. Its unsolved part encodes the problems to be computed, its solved part encodes the problems where the common term structure differs and which were detected for the first time, and its computed substitution encodes the common term structure already decomposed. Thus, a valid configuration encodes the entire history of the decomposition up to that point.

The decomposition of the term structure is carried out by the inference rules presented in Figure 1. The algorithm `Antiunify` is specified in [9] as a recursive unary function that maps valid configurations into valid configurations, and it consists of the exhaustive application of the rules to a valid input configuration. Since the unsolved part of a valid configuration is specified by a list of AUTs, the rules are always applied to the first member of that list. The size of the unsolved part of a configuration, which is the sum of the `sizes` of its members, was the chosen metric for ensuring *termination* in [9].

The rules presented in Figure 1 are specified in [9] by means of unary functions `DecF`, `DecP`, `Synt`, and `Solve` that require an input valid configuration that matches the application of the rule and returns an output configuration according to their (dependent) type. That type encodes, among other properties, that the output configuration is a valid configuration with a *smaller size* than the input configuration. PVS generated a *type correctness condition* (TCC) for each function, and each one of those TCCs required a manual proof. The advantage of this approach is that PVS generates a termination TCC for the specification of `Antiunify`, and the prover can automatically prove it using the information encoded in the (dependent) types of the input-output configurations of the inference rules. Thus, an exhaustive application of the rules terminates in a configuration with an empty unsolved part. Those are called *normal configurations* (or *final configurations*) and are recognized in PVS by means of the predicate `normal_configuration?`.

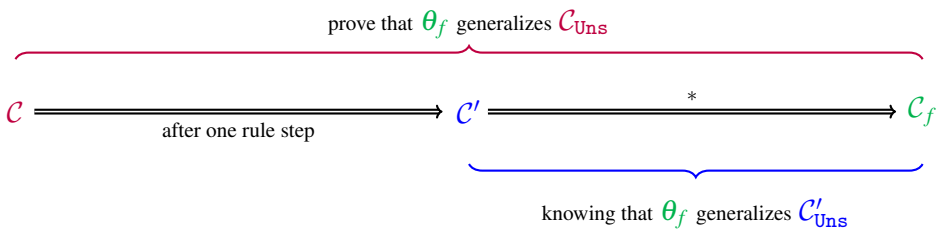

Figure 2: Inductive step in the soundness theorem presented in [9].

The background was enough to prove `antiunif_is_sound` in [9]. The argument was constructed by induction on the size of the unsolved part of a valid configuration together with a case analysis concerning the employed rule in the inductive step, which is presented in Figure 2. The effort required

was measured in [9] by the number of proof commands (for the proof and also its dependencies) and the amounts are 2.09% for DecF, 4.56% for DecP, 61.70% for Synt, 29.60% for SolR, and 2.05% for SolNR. Surprisingly, the rules Synt and SolR together make up 91.10% of the entire effort and algorithmically, going from $\mathcal{C}$ to $\mathcal{C}'$, they only add to $\mathcal{C}_{\mathtt{Sub}}$ a new basic substitution $(X \mapsto a)$, where X is a label in $\mathcal{C}_{\mathtt{Uns}}$ and a is an atomic term. While that transformation seems trivial in a pen-and-paper proof, a considerable amount of work is required to analyze its effects in a mechanized and formally verified proof for the case where a is a variable. In that case, the branches for those two rules relied on dependencies (lemmas) that stated the invariance and preservation properties concerning the action of θ_f on the variable a. These properties explained why all that effort was required.

3 Dissection Towards a Mechanized Completeness Proof

Using the formalization elements developed for the proof of soundness in [9], one can try to address the completeness proofs for the decomposition rules (DecF and DecP). Indeed, for an inductive proof like the one presented in Figure 2 using a notion `r_generalizer` of arbitrary generalizer that restrict its domain and range avoiding occurrences of the variables used by the algorithm, the completeness case analysis of these two rules will amount more than 2500 and 3500 PVS proof commands for the DecF and DecP rules, respectively (see the extended version of this paper). Such an approach is sufficient to guarantee the completeness of the algorithm for the subclass of anti-unification problems that do not require applications of the rules SolR and Synt, but not for arbitrary problems. An important class of such problems comprises those that do not repeat variables or constants. For those problems, the decomposition rules will only lead to solved AUTs that cannot appear more than once. Since no variable nor constant appears repeatedly, no application of the rules SolR or Synt is possible.

Surprisingly, following the same inductive schema with the same notion of generalizer did not work for the SolR, SolNR, and Synt rules, as this section explains. The problem arises in the inductive step: given an arbitrary generalizer γ for the problem encoded by a configuration $\mathcal{C}$, to apply the inductive hypothesis, one needs to build an associated generalizer γ' for the configuration $\mathcal{C}'$; and using the hypothesis that it is more general than the computed solution given by θ_f, in the final configuration $\mathcal{C}_f$, prove that γ is also more general than θ_f (see Figure 3).

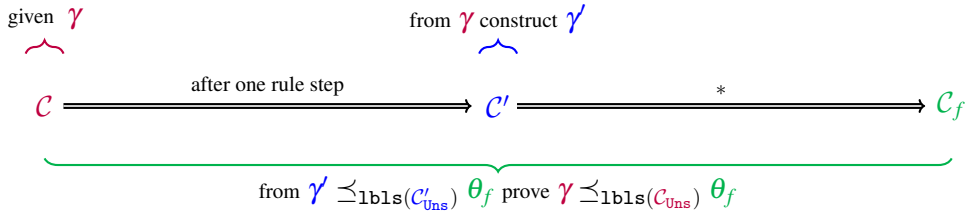

Figure 3: Inductive step in the experiments towards a completeness proof.

Initially, Subsection 3.1 explains new notions that are required to compile configuration preservation properties for the analysis of the Synt, SolR, and SolNR rules. These notions are used to record invariance properties throughout the history of computations. Afterward, Subsection 3.2 presents crucial adjustments required to address completeness for those rules; finally, Subsection 3.3 discusses how these adjusted notions are used in the formalization of the inductive proof. Although the branches of the decomposition rules can be derived from the elements developed in [9], for insertion into the case analysis of this new inductive schema, they require straightforward adaptations, as explained in Subsection 3.4.

3.1 New Notions Required for the Completeness Analysis

An *initial configuration* is a valid configuration with an empty solved part and identity computed substitution, and it is recognized in PVS by means of the unary predicate `initial_configuration?`. In PVS, a single application of any of the rules is recognized by the binary predicate `Antiunify?` specified from the binary predicates `DecF?`, `DecP?`, `Synt?`, and `Solve?`. These predicates check whether a configuration is derived from another configuration by applying the respective inference rule. The binary predicate `RTC(Antiunify?)`, constructed using the reflexive-transitive closure of abstract reduction relations, `RTC(R)`, from the NASAlib theory of rewriting (see, e.g., [25, 31]). For brevity, in this section, we will also use standard rewriting notation for the specified predicate `Antiunify?`; thus, $\Rightarrow$ and $\Rightarrow^*$ will denote `Antiunify?` and `RTC(Antiunify?)`, respectively.

All information pertaining to the derivation of a valid configuration from an initial configuration is stored in the former. All labels that were used during its derivation encode positions in the common term structure where subproblems were detected. Labels are transformed at each step taken: New labels are generated, labels are moved from the unsolved part to the solved part, to the domain of the computed substitution, or to its range. That motivates the following definition:

Definition 2 (Labels of a Valid Configuration)**.** *The set of labels of a valid configuration $\mathcal{C}$ is defined as* $\mathtt{lbls}(\mathcal{C}) := \mathtt{lbls}(\mathcal{C}_{\mathtt{Uns}}) \cup \mathtt{lbls}(\mathcal{C}_{\mathtt{Sol}}) \cup \mathtt{dom}(\mathcal{C}_{\mathtt{Sub}})$.

The information encoded by labels is essential if one aims for completeness, as labels encode positions. One important preservation property is stated in the next lemma and proved by induction on the number of steps in a derivation. Intuitively, all information encoded by the labels of a valid configuration is inherited by every valid configuration derived from it. In particular, if $\mathcal{C} \Rightarrow \mathcal{C}'$ by means of `SolR`, `SolNR`, or `Synt`, then $\mathtt{lbls}(\mathcal{C}') = \mathtt{lbls}(\mathcal{C})$.

Lemma 3 (Preservation of Labels)**.** *If $\mathcal{C} \Rightarrow^* \mathcal{C}'$, then* $\mathtt{lbls}(\mathcal{C}) \subseteq \mathtt{lbls}(\mathcal{C}')$.

Another important class of variables in a valid configuration derived from an initial configuration are those which appear on the left- and right-hand sides of the AUTs of the latter. In the decomposition of the common term structure, one can detect a trivial AUT whose both sides are the same variable, and the rule `Synt` transforms that AUT accordingly, storing the following information at the computed substitution: The label that encodes the position where the problem was encountered must be mapped to that variable that occurs at that position. That is the only possible transformation regarding those types of variables, and it motivates the following definition:

Definition 4 (Protected Variables of a Valid Configuration)**.** *The set of protected variables of a valid configuration $\mathcal{C}$ is defined as* $\mathtt{prtd}(\mathcal{C}) := \mathtt{vars}(\mathcal{C}_{\mathtt{Uns}} \cup \mathcal{C}_{\mathtt{Sol}}) \cup (\mathtt{rvars}(\mathcal{C}_{\mathtt{Sub}}) \setminus \mathtt{lbls}(\mathcal{C}_{\mathtt{Uns}} \cup \mathcal{C}_{\mathtt{Sol}}))$.

A variable that occurs in the left- or right-hand side of an AUT of $\mathcal{C}_0$ can occur in a subsequent derived configuration $\mathcal{C}$ in $\mathtt{vars}(\mathcal{C}_{\mathtt{Uns}} \cup \mathcal{C}_{\mathtt{Sol}}) \cup \mathtt{rvars}(\mathcal{C}_{\mathtt{Sub}})$. How they appear in those sets is understood by the action of the inference rules employed in the derivation. One important invariance property is presented in the following lemma, which is proved by induction on the number of steps of a derivation. A crucial step in that proof is understanding how the set of variables of the computed substitution of a valid configuration changes after applications of the standard inference rules.

Lemma 5 (Invariance of Protected Variables)**.** *If $\mathcal{C} \Rightarrow^* \mathcal{C}'$, then* $\mathtt{prtd}(\mathcal{C}) = \mathtt{prtd}(\mathcal{C}')$.

An important consequence of the previous lemma is the following: the set of protected variables of any configuration derived from an initial configuration is exactly the set of variables of the unsolved part of the latter. Also, it is important to notice that $\mathtt{lbls}(\mathcal{C}) \cap \mathtt{prtd}(\mathcal{C}) = \emptyset$ for any valid configuration $\mathcal{C}$. Another type of invariant present throughout the computation is the problem being solved. To discuss that, one requires the proper vocabulary:

Definition 6 (Lateral Substitutions)**.**

1. *The* left *(resp.* right*)* substitution associated with *a valid set A of AUTs is the substitution ρ_L^A (resp. ρ_R^A) with* $\mathtt{dom}(\rho_L^A) = \mathtt{lbls}(A)$ *(resp.* $\mathtt{dom}(\rho_R^A) = \mathtt{lbls}(A)$*) such that* $X\rho_L^A = \mathtt{lhs}(eq_X)$ *(resp.* $X\rho_R^A = \mathtt{rhs}(eq_X)$*) for every X in* $\mathtt{lbls}(A)$*;*
2. *The* left *and* right substitutions associated with *a valid configuration $\mathcal{C}$ are:*

$$\rho_L^{\mathcal{C}} := \rho_L^{\mathcal{C}_{\mathtt{Uns}} \cup \mathcal{C}_{\mathtt{Sol}}} \qquad \text{and} \qquad \rho_R^{\mathcal{C}} := \rho_R^{\mathcal{C}_{\mathtt{Uns}} \cup \mathcal{C}_{\mathtt{Sol}}}.$$

By definition, one can see that $\mathtt{rvars}(\rho_L^{\mathcal{C}}) \cup \mathtt{rvars}(\rho_R^{\mathcal{C}}) \subseteq \mathtt{prtd}(\mathcal{C})$, thus the lateral substitutions associated with $\mathcal{C}$ are idempotent. Also, their restriction to any part of $\mathcal{C}$ is the corresponding lateral substitution associated with that part. Lateral substitutions play an important role in reconstructing AUTs that encode anti-unification problems. The invariance result hinted at before can be stated as the next lemma. It is proved by induction on the length of the derivation, paying attention to how each inference rule modifies the lateral and computed substitutions associated with a configuration, as well as to the role of the label of the AUT transformed by the inference rule.

Lemma 7 (Problem Invariance)**.** *If $\mathcal{C} \Rightarrow^* \mathcal{C}'$, then $X\mathcal{C}'_{\mathtt{Sub}}\rho_L^{\mathcal{C}'} = X\mathcal{C}_{\mathtt{Sub}}\rho_L^{\mathcal{C}}$ and $X\mathcal{C}'_{\mathtt{Sub}}\rho_R^{\mathcal{C}'} = X\mathcal{C}_{\mathtt{Sub}}\rho_R^{\mathcal{C}}$ for every X in* $\mathtt{dom}(\mathcal{C}_{\mathtt{Sub}})$.

Since all information is preserved in a derivation, one can think intuitively about Lemma 7 by employing a jigsaw puzzle: One breaks the entire picture (an AUT of the unsolved part of a valid configuration) into small pieces (AUTs obtained from that AUT by applications of inference rules to configurations) in a way that, if all of them are put together (the compositions of the computed and lateral substitutions), then one obtains the original picture again. That is the *problem invariance*: No matter how one breaks a problem into subproblems, the problem remains the same throughout the computation.

All the preservation properties presented so far can be used to completely describe the history of computation in the derivation of a valid configuration from an initial one. For instance, an argument by induction is sufficient for proving the following lemma:

Lemma 8. *[History of the Solved Part] Let $\mathcal{C}_0$ be an initial configuration and $\mathcal{C}$ be a valid configuration such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C}$. For every AUT eq_X in $\mathcal{C}_{\mathtt{Sol}}$, there exist two valid configurations $\mathcal{C}'$ and $\mathcal{C}''$ such that eq_X belongs to $\mathcal{C}'_{\mathtt{Uns}} \cap \mathcal{C}''_{\mathtt{Sol}} \setminus \mathcal{C}'_{\mathtt{Sol}}$ and $\mathcal{C}_0 \Rightarrow^* \mathcal{C}' \Rightarrow \mathcal{C}'' \Rightarrow^* \mathcal{C}$, where the step $\mathcal{C}' \Rightarrow \mathcal{C}''$ uses* SolNR.

Another important description of the history of computation during a derivation concerns the domain of the computed substitution. That history is presented in the next lemma, which can be proved by induction using Lemma 7:

Lemma 9. *[History of the Computed Substitution] Let $\mathcal{C}_0$ be an initial configuration and $\mathcal{C}$ be a configuration such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C}$. For every X in* $\mathtt{dom}(\mathcal{C}_{\mathtt{Sub}})$*, exactly one of the following holds:*

1. *$X\mathcal{C}_{\mathtt{Sub}}$ is a variable and exactly one of the following holds:*
   - **1.1.** *$X\mathcal{C}_{\mathtt{Sub}}$ is the label of the AUT*
   $$X\mathcal{C}_{\mathtt{Sub}}\rho_L^{\mathcal{C}} \triangleq_{X\mathcal{C}_{\mathtt{Sub}}} X\mathcal{C}_{\mathtt{Sub}}\rho_R^{\mathcal{C}}$$
   that belongs to $\mathcal{C}_{\mathtt{Sol}}$;
   - **1.2.** *$X\mathcal{C}_{\mathtt{Sub}}$ is a member of* $\mathtt{prtd}(\mathcal{C})$ *and the AUT*
   $$X\mathcal{C}_{\mathtt{Sub}}\rho_L^{\mathcal{C}} \triangleq_X X\mathcal{C}_{\mathtt{Sub}}\rho_R^{\mathcal{C}}$$
   is the trivial AUT
   $$X\mathcal{C}_{\mathtt{Sub}} \triangleq_X X\mathcal{C}_{\mathtt{Sub}}$$
   that belongs to $\mathcal{C}'_{\mathtt{Uns}}$ for some valid configuration $\mathcal{C}'$ such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C}' \Rightarrow^* \mathcal{C}$;

2. *$XC_{\texttt{Sub}}$ is either the unit or a constant and the AUT*

$$XC_{\texttt{Sub}}\rho_L^{\mathcal{C}} \triangleq_X XC_{\texttt{Sub}}\rho_R^{\mathcal{C}}$$

is the trivial AUT

$$XC_{\texttt{Sub}} \triangleq_X XC_{\texttt{Sub}}$$

that belongs to $\mathcal{C}'_{\texttt{Uns}}$ for some valid configuration $\mathcal{C}'$ such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C}' \Rightarrow^* \mathcal{C}$;

3. *$XC_{\texttt{Sub}}$ is either a functional application or a pair, and the AUT*

$$XC_{\texttt{Sub}}\rho_L^{\mathcal{C}} \triangleq_X XC_{\texttt{Sub}}\rho_R^{\mathcal{C}}$$

is decomposable and belongs to $\mathcal{C}'_{\texttt{Uns}}$ for some valid configuration $\mathcal{C}'$ such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C}' \Rightarrow^* \mathcal{C}$.

3.2 Adjustment of Generalizer Notions

At this point, one has sufficient grounds to provide the building blocks of a definition that encodes what a generalizer for a valid configuration is. Lemma 7 guarantees the consistency of the following definition:

Definition 10 (Extended Set of AUTs)**.** *The* extended set of AUTs *of a valid configuration $\mathcal{C}$ is defined as:*

$$\mathcal{C}_{\texttt{Ext}} := \left\{ XC_{\texttt{Sub}}\rho_L^{\mathcal{C}} \triangleq_X XC_{\texttt{Sub}}\rho_R^{\mathcal{C}} \,\middle|\, X \in \texttt{dom}(\mathcal{C}_{\texttt{Sub}}) \right\}.$$

A close inspection of the previous definition and the definition of lateral substitutions ensures that the set just introduced is a valid set of AUTs. Suppose $\mathcal{C} \Rightarrow \mathcal{C}'$. If `SolNR` is the employed rule, then $\mathcal{C}'_{\texttt{Ext}} = \mathcal{C}_{\texttt{Ext}}$. If any other rule is the employed one, then $\mathcal{C}'_{\texttt{Ext}} = \mathcal{C}_{\texttt{Ext}} \cup \{ eq_X \}$, where eq_X is the AUT in $\mathcal{C}_{\texttt{Uns}}$ that is transformed by the rule. Thus, an inductive argument together with Lemma 7 provides the following natural result:

Lemma 11 (Preservation of Extended Set)**.** *If $\mathcal{C} \Rightarrow^* \mathcal{C}'$, then $\mathcal{C}_{\texttt{Ext}} \subseteq \mathcal{C}'_{\texttt{Ext}}$.*

Notice that in the set of labels of a valid configuration, the labels of the unsolved and solved parts are obtained directly from the AUTs that are already present in the configuration. The labels in the domain of the substitution refer to AUTs that were already transformed, and the extended set of AUTs is a very natural way to recover them. Thus, one can define:

Definition 12 (Total Set of AUTs)**.** *The* total set of AUTs *of a valid configuration $\mathcal{C}$ is defined by*

$$\mathcal{C}_{\texttt{Tot}} := \mathcal{C}_{\texttt{Uns}} \cup \mathcal{C}_{\texttt{Sol}} \cup \mathcal{C}_{\texttt{Ext}}.$$

A close inspection of the previous definition ensures that the set just introduced is a valid set of AUTs. Moreover, the identity $\texttt{lbls}(\mathcal{C}_{\texttt{Tot}}) = \texttt{lbls}(\mathcal{C})$ holds. Suppose $\mathcal{C} \Rightarrow \mathcal{C}'$. If `SolNR`, `SolR`, or `Synt` is the employed rule, then $\mathcal{C}'_{\texttt{Tot}} = \mathcal{C}_{\texttt{Tot}}$. If any decomposition rule is the employed one, then $\mathcal{C}'_{\texttt{Ext}} = \mathcal{C}_{\texttt{Ext}} \cup A$, where A is the set of new AUTs introduced by the employed rule. Thus, an inductive argument provides the following natural result:

Lemma 13 (Preservation of Total Set)**.** *If $\mathcal{C} \Rightarrow^* \mathcal{C}'$, then $\mathcal{C}_{\texttt{Tot}} \subseteq \mathcal{C}'_{\texttt{Tot}}$.*

Given two terms s and t, a term g is a generalizer for s and t iff there exist substitutions σ and τ such that $g\sigma = s$ and $g\tau = t$. Thus, a generalizer for an AUT is a generalizer for both its sides. In the case of a valid configuration, one requires a notion of a generalizer that *uniformly generalizes* all the AUTs associated with that configuration. Thus, a generalizer for a valid configuration must be a substitution, and any comparison between generalizers must rely on the instantiation preorder restricted to the set of labels of the configuration. A natural definition is the following one:

Definition 14 (Total Generalizer for a Valid Configuration)**.**

1. *Let A be a valid set of AUTs. A substitution γ is a* total generalizer for *A iff there exist* left *and* right cohesion substitutions τ_L^A *and* τ_R^A*, respectively, such that*

$$X\gamma\tau_L^A = \mathtt{lhs}(eq_X) \qquad \text{and} \qquad X\gamma\tau_R^A = \mathtt{rhs}(eq_X)$$

   for every eq_X in A;

2. *A substitution γ is a* total generalizer for *a valid configuration $\mathcal{C}$ iff γ is a total generalizer for $\mathcal{C}_{\mathtt{Tot}}$. Its left and right cohesion substitutions are denoted by $\tau_L^{\mathcal{C}}$ and $\tau_R^{\mathcal{C}}$, respectively.*

That is the notion of a generalizer for a valid configuration required for proving completeness. The identity ι is a total generalizer for every valid configuration $\mathcal{C}$ and the lateral substitutions of $\mathcal{C}_{\mathtt{Tot}}$ are its cohesion substitutions. That mirrors the fact that a variable generalizes any term. As a bonus, the new notions presented so far are already enough to prove soundness in a way that is different from the ones presented in [9] or in [4]. The new proof of soundness relies on the preservation properties stated in Lemmas 7 and 13, and it is done by induction on the number of steps of the derivation.

The pen-and-paper proof of completeness that is going to be presented here is constructive and completely different from the one presented in [4], because no results on confluence or the type of syntactic anti-unification will be assumed. On the contrary, that type will be a consequence of termination, soundness, and completeness. Before presenting the proof, certain properties of generalizers for valid configurations must be proved. The first one is a natural definitional corollary:

Lemma 15 (Total Generalizer Coherence)**.** *If A is a valid set of AUTs, γ is a total generalizer for it, and eq_X is a member of A, then $X\gamma$ is a total generalizer for eq_X. More precisely:*

1. *If eq_X is a decomposable AUT, then:*
   - **1.1.** *If $\mathtt{lhs}(eq_X)$ is a functional application, then $X\gamma$ is either a variable or a functional application starting with the same root symbol of $\mathtt{lhs}(eq_X)$;*
   - **1.2.** *If $\mathtt{lhs}(eq_X)$ is a pair, then $X\gamma$ is either a variable or a pair;*
2. *If eq_X is a trivial AUT, then $X\gamma$ is the unit, a constant or a variable;*
3. *If eq_X is a solved AUT, then $X\gamma$ is a variable;*

A particularly important preservation result concerning total generalizers for valid configurations is the following one. Despite being a natural definitional corollary, it essentially informs that a total generalizer cannot associate the same solution to problems encoded by AUTs of distinct classes:

Lemma 16 (Preservation of Classification)**.** *Let A be a valid set of AUTs and γ be an arbitrary total generalizer for A. For all eq_X and eq_Y in A, if $X\gamma = Y\gamma$, then eq_X and eq_Y are repeated.*

The key step in the proof of the previous lemma is the existence of cohesion substitutions associated with the total generalizer γ. Under the hypothesis $X\gamma = Y\gamma$, they force the left and right-hand sides of the AUTs eq_X and eq_Y to be respectively the same terms. A particularly important consequence of the previous lemma is the following lemma, which deals with syntactic and solved AUTs:

Lemma 17 (Separation)**.** *Let A be a valid set of AUTs, γ be an arbitrary total generalizer for A, and eq_X and eq_Y be arbitrary members of A.*

1. *If eq_X and eq_Y are trivial AUTs and $\mathtt{lhs}(eq_X) \neq \mathtt{lhs}(eq_Y)$, then $X\gamma \neq Y\gamma$;*
2. *If all members of A are not repeated in A and $X \neq Y$, then $X\gamma \neq Y\gamma$.*

An important application of Lemmas 15 and 17 is the following. If $\mathcal{C}$ is a valid configuration, then the members of $\mathcal{C}_{\mathtt{Sol}}$ are not repeated in it. Thus, a total generalizer for $\mathcal{C}$ maps distinct labels in $\mathtt{lbls}(\mathcal{C}_{\mathtt{Sol}})$ to distinct variables. In other words, a total generalizer for a valid configuration is able to separate non-repeated solved (resp. non-repeated trivial) AUTs that appear in a derivation ending in that same configuration. This property will play an important role in a step of the proof of completeness. Another important property of total generalizers for valid configurations is presented in the following lemma:

Lemma 18 (Choice of Total Generalizer)**.** *Let $\mathcal{C}$ be a valid configuration and $\mathbb{A}$ be a set of variables. For every total generalizer γ for $\mathcal{C}$ there exists a total generalizer γ' for $\mathcal{C}$ such that $\mathtt{rvars}(\gamma') \cap \mathbb{A} = \emptyset$ and $\gamma' \simeq_{\mathtt{lbls}(\mathcal{C})} \gamma$.*

To prove the previous lemma, one needs to enumerate all the variables in $\mathtt{rvars}(\gamma) \cap \mathbb{A}$ (if there are any) and rename all of them into distinct fresh variables using a renaming α. New cohesion substitutions are constructed using α and the old ones. Lemma 18 serves as inspiration for providing the following definition:

Definition 19 (Restricted Total Generalizer)**.** *Let $\mathcal{C}$ be a valid configuration and $\mathcal{C}_f$ be a final configuration such that $\mathcal{C} \Rightarrow^* \mathcal{C}_f$. A total generalizer γ for $\mathcal{C}$ is a* restricted total generalizer for $\mathcal{C}$ relative to $\mathcal{C}_f$ *iff $\mathtt{rvars}(\gamma) \cap (\mathtt{lbls}(\mathcal{C}_f) \cup \mathtt{prtd}(\mathcal{C}_f)) = \emptyset$ and $\gamma\gamma = \gamma$.*

The specification of that type of total generalizer is crucial in the proof of completeness. Lemma 18 shows that if a valid configuration admits a total generalizer, then it admits a restricted total generalizer that is equivalent to the original one on the labels of that configuration. That means that one can restrict their attention to that class of more well-behaved total generalizers for valid configurations. Also, since equivalence holds, completeness is not lost by such a restriction. That is the path followed in the PVS specification. We are ready to state completeness. All the preservation properties, invariants, and (restricted) total generalizer properties presented so far will naturally guide the construction of the proof. It will be dismembered into parts. `Synt`, `SolNR`, and `SolR` rules are presented in Subsection 3.3, while `DecF` and `DecP` are presented in Subsection 3.4.

Theorem 20 (Strong Completeness for Restricted Total Generalizers)**.** *Let $\mathcal{C}_0$ be an initial configuration, $\mathcal{C}_f$ be a final configuration such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C}_f$, and θ_f be the computed substitution of $\mathcal{C}_f$. For every valid configuration $\mathcal{C}$ such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C} \Rightarrow^* \mathcal{C}_f$ and every restricted total generalizer γ for $\mathcal{C}$ relative to $\mathcal{C}_f$ it is the case that $\gamma \preceq_{\mathtt{lbls}(\mathcal{C})} \theta_f$.*

The proof of Theorem 20 is done by induction on the number of steps taken in $\mathcal{C} \Rightarrow^* \mathcal{C}_f$. In the base case, one has $\mathcal{C} = \mathcal{C}_f$. Let S_f be the solved part of $\mathcal{C}_f$. By the definition of total generalizer for $\mathcal{C}_f$, one can always assume that $\mathtt{dom}(\gamma) \cap \mathtt{prtd}(\mathcal{C}_f) = \emptyset$. Since $\mathtt{rvars}(\theta_f) \subseteq \mathtt{lbls}(S_f) \cup \mathtt{prtd}(\mathcal{C}_f)$, one employs the definition of restricted total generalizer and Lemmas 8, 9, 15, and 17 to conclude that $\gamma \preceq_{\mathtt{lbls}(\mathcal{C}_f)} \theta_f$.

For the inductive step, one assumes the result for derivations of length m and supposes that $\mathcal{C}_0 \Rightarrow^* \mathcal{C} \Rightarrow^{m+1} \mathcal{C}_f$. Thus, there exists a valid configuration $\mathcal{C}'$ such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C} \Rightarrow \mathcal{C}' \Rightarrow^m \mathcal{C}_f$. The analysis now proceeds by considering the rule employed in the step $\mathcal{C} \Rightarrow \mathcal{C}'$. One must start with an arbitrary restricted total generalizer γ for $\mathcal{C}$ relative to $\mathcal{C}_f$ and use it to construct a restricted total generalizer γ' for $\mathcal{C}'$ relative to $\mathcal{C}_f$ to use the induction hypothesis (i.h.). That construction must be so that one can push the comparison using the instantiation preorder back to γ. The situation is presented in Figure 4. Once the proof is finished, one can employ Lemma 18 to derive the following corollary from which one concludes that the type of syntactic anti-unification is unitary:

Corollary 21 (Strong Completeness)**.** *Let $\mathcal{C}_0$ be an initial configuration, $\mathcal{C}_f$ be a final configuration such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C}_f$, and θ_f be the computed substitution of $\mathcal{C}_f$. For every valid configuration $\mathcal{C}$ such that $\mathcal{C}_0 \Rightarrow^* \mathcal{C} \Rightarrow^* \mathcal{C}_f$ and every total generalizer γ for $\mathcal{C}$ it is the case that $\gamma \preceq_{\mathtt{lbls}(\mathcal{C})} \theta_f$.*

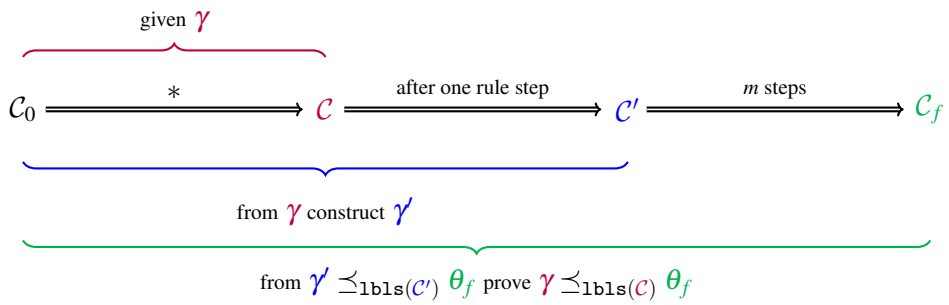

Figure 4: Inductive step in the proof of completeness.

3.3 Completeness Analysis for Syntactic and Solve Rules

This subsection is concerned with the branches of the proof of completeness where the step $\mathcal{C} \Rightarrow \mathcal{C}'$ is achieved by means of the rules `Synt`, `SolNR`, or `SolR`. Let γ be an arbitrary restricted total generalizer for $\mathcal{C}$ relative to $\mathcal{C}_f$. The information concerning both γ and the history of the computation is required for constructing a restricted total generalizer γ' for $\mathcal{C}'$ relative to $\mathcal{C}_f$ that allows the use of the reasoning illustrated in Figure 4.

1. If `Synt` is the employed rule, then there exists a trivial AUT eq_X in $\mathcal{C}_{\mathtt{Uns}}$ such that $\mathcal{C}'_{\mathtt{Uns}} = \mathcal{C}_{\mathtt{Uns}} \setminus \{eq_X\}$, $\mathcal{C}'_{\mathtt{Sol}} = \mathcal{C}_{\mathtt{Sol}}$, and $\mathcal{C}'_{\mathtt{Sub}} = \mathcal{C}_{\mathtt{Sub}}(X \mapsto \mathtt{lhs}(eq_X))$. By Lemma 16, one knows that $X\gamma$ is different from every variable that is assigned by γ to the labels of non-trivial AUTs. The problem encoded by the trivial AUT eq_X may appear in different positions in the common term structure. The information about it may be already stored in $\mathcal{C}_{\mathtt{Uns}} \setminus \{eq_X\}$ or in $\mathcal{C}_{\mathtt{Sub}}$.
2. If `SolNR` is the employed rule, then there exists a solved AUT eq_X in $\mathcal{C}_{\mathtt{Uns}}$ such that $\mathcal{C}'_{\mathtt{Uns}} = \mathcal{C}_{\mathtt{Uns}} \setminus \{eq_X\}$, $\mathcal{C}'_{\mathtt{Sol}} = \mathcal{C}_{\mathtt{Sol}} \cup \{eq_X\}$, and $\mathcal{C}'_{\mathtt{Sub}} = \mathcal{C}_{\mathtt{Sub}}$. In other words, this is the first time that the problem encoded by the solved AUT eq_X is encountered. From Lemma 17, it follows that $X\gamma$ is different from $Y\gamma$ for every Y in $\mathtt{lbls}(\mathcal{C}_{\mathtt{Sol}})$. By Lemma 16, one knows that $X\gamma$ is different from every variable that is assigned by γ to the labels of non-solved AUTs. Notice that no information about the problem is already stored in $\mathcal{C}_{\mathtt{Sub}}$. Nonetheless, information about it may already be stored in $\mathcal{C}_{\mathtt{Uns}} \setminus \{eq_X\}$.
3. If `SolR` is the employed rule, then there exist solved AUTs eq_X in $\mathcal{C}_{\mathtt{Uns}}$ and $eq_{X'}$ in $\mathcal{C}_{\mathtt{Sol}}$ such that $\mathcal{C}'_{\mathtt{Uns}} = \mathcal{C}_{\mathtt{Uns}} \setminus \{eq_X\}$, $\mathcal{C}'_{\mathtt{Sol}} = \mathcal{C}_{\mathtt{Sol}}$, and $\mathcal{C}'_{\mathtt{Sub}} = \mathcal{C}_{\mathtt{Sub}}(X \mapsto X')$. In other words, the problem encoded by the solved AUT $eq_{X'}$ is encountered again. From Lemma 17, it follows that $X\gamma$ and $X'\gamma$ are different from $Y\gamma$ for every Y in $\mathtt{lbls}(\mathcal{C}_{\mathtt{Sol}}) \setminus \{X'\}$. Since eq_X and $eq_{X'}$ are repeated AUTs, it is not necessarily the case that $X\gamma = X'\gamma$. By Lemma 16, one knows that $X\gamma$ is different from every variable that is assigned by γ to the labels of non-solved AUTs. The problem encoded by the solved AUT eq_X may appear in different positions in the common term structure, and the information about it is already stored in $\mathcal{C}_{\mathtt{Sol}}$.

That explains why the elements provided by [9] are not sufficient for addressing those rules and why it is crucial to require in the specification of a total generalizer for a valid configuration that it must also generalize both its solved part and the domain of its computed substitution together with its unsolved part. Only requiring that for the latter does not necessarily provide all the needed information regarding the history of the computation to compare the restricted total generalizer and the final computed substitution in those branches of the proof. Now, both the definition of total generalizer and the preservation properties

previously discussed come to the rescue. Since the employed rule is Synt, SolNR, or SolR, one knows by Lemma 13 that $\mathcal{C}'_{\mathtt{Tot}} = \mathcal{C}_{\mathtt{Tot}}$ and $\mathtt{lbls}(\mathcal{C}') = \mathtt{lbls}(\mathcal{C})$. In other words, all the information required for the comparison is already present in $\mathcal{C}$ and is preserved from $\mathcal{C}$ to $\mathcal{C}'$. Thus, one concludes that γ is already a restricted total generalizer for $\mathcal{C}'$ relative to $\mathcal{C}_f$ and $\gamma \preceq_{\mathtt{lbls}(\mathcal{C})} \theta_f$ follows immediately from the induction hypothesis. Although the pen-and-paper proof seems straightforward, much of the effort in the verification will be related to its dependencies.

3.4 Completeness Analysis for Decomposition Rules

This subsection is concerned with the branches of the proof of completeness where the step $\mathcal{C} \Rightarrow \mathcal{C}'$ is achieved by means of either DecF or DecP. Let γ be an arbitrary restricted total generalizer for $\mathcal{C}$ relative to $\mathcal{C}_f$.

1. If DecF is the employed rule, then there exist a decomposable AUT $eq_X = ft_L \stackrel{\triangle}{=}_X ft_R$ in $\mathcal{C}_{\mathtt{Uns}}$ and a variable Y fresh for $\mathcal{C}$ such that $\mathcal{C}'_{\mathtt{Uns}} = \mathcal{C}_{\mathtt{Uns}} \cup \{ eq_Y \} \setminus \{ eq_X \}$, $\mathcal{C}'_{\mathtt{Sol}} = \mathcal{C}_{\mathtt{Sol}}$, $\mathcal{C}'_{\mathtt{Sub}} = \mathcal{C}_{\mathtt{Sub}} (X \mapsto fY)$, and $eq_Y = t_L \stackrel{\triangle}{=}_Y t_R$. By Lemma 15, the term $X\gamma$ is either a variable or ft for some term t.

1.1. Suppose $X\gamma = ft$ for some term t. Define:

$$\gamma' \coloneqq \gamma|_{\mathtt{dom}(\gamma) \setminus \{Y\}} (Y \mapsto t).$$

That is a restricted total generalizer for $\mathcal{C}'$ relative to $\mathcal{C}_f$. By the i.h., $\gamma' \preceq_{\mathtt{lbls}(\mathcal{C}')} \theta_f$. There exists a substitution δ' such that $\gamma'\delta' =_{\mathtt{lbls}(\mathcal{C}')} \theta_f$. Letting

$$\delta \coloneqq (Y \mapsto t)\, \delta',$$

one concludes that $\gamma\delta =_{\mathtt{lbls}(\mathcal{C})} \theta_f$. Indeed,

$$X\gamma\delta = ft\delta' = fY\gamma'\delta' = fY\theta_f = X\theta_f,$$

and $Z\gamma\delta = Z\gamma'\delta'$ for Z in $\mathtt{lbls}(\mathcal{C}) \setminus \{X\}$. Thus, $\gamma \preceq_{\mathtt{lbls}(\mathcal{C})} \theta_f$;

1.2. Suppose $X\gamma = U$ is a variable. There exist cohesion substitutions $\tau_L^{\mathcal{C}}$ and $\tau_R^{\mathcal{C}}$ of γ. There exists a substitution δ'' such that $\gamma\delta'' =_{\mathtt{lbls}(\mathcal{C})} \theta_f\gamma$. Let W be fresh for $\mathcal{C}_f$, γ, $\tau_L^{\mathcal{C}}$, $\tau_R^{\mathcal{C}}$, and δ'. Define:

$$\gamma' \coloneqq \gamma|_{\mathtt{dom}(\gamma) \setminus \{Y\}} (U \mapsto fW)\, (Y \mapsto W).$$

That is a restricted total generalizer for $\mathcal{C}'$ relative to $\mathcal{C}_f$. By the i.h., $\gamma' \preceq_{\mathtt{lbls}(\mathcal{C}')} \theta_f$. There exists a substitution δ' such that $\gamma'\delta' =_{\mathtt{lbls}(\mathcal{C}')} \theta_f$. Letting

$$\delta \coloneqq (U \mapsto fW)\, (Y \mapsto W)\, \delta',$$

one concludes that $\gamma\delta =_{\mathtt{lbls}(\mathcal{C})} \theta_f$. Indeed,

$$X\gamma\delta = fW\delta' = fY\gamma'\delta' = fY\theta_f = X\theta_f,$$

and $Z\gamma\delta = Z\gamma'\delta'$ for Z in $\mathtt{lbls}(\mathcal{C}) \setminus \{X\}$. Thus, $\gamma \preceq_{\mathtt{lbls}(\mathcal{C})} \theta_f$;

2. The analysis for DecP is similar and will be omitted.

Crucial steps in verifying the previous arguments rely on the constraints imposed by the definition of restricted total generalizers. For instance, the constraint $\mathtt{rvars}(\gamma) \cap (\mathtt{lbls}(\mathcal{C}_f) \cup \mathtt{prtd}(\mathcal{C}_f)) = \emptyset$ is vital for the construction of the substitution γ'. Moreover, the fact that a total generalizer for a valid configuration generalizes the unsolved part of that configuration is crucial for applying the induction step in verifying these branches of the proof.

4 Conclusion

This paper presented a rigorous pen-and-paper proof of the completeness theorem for syntactic anti-unification and the discussion clarified the formal elements required to mechanically verify the theorem inductively in PVS (and other proof assistants as well). In particular, the paper highlighted important differences that make that mechanization exercise much more complex than verifying the algorithmic solutions to unification, for which the cases of solved and syntactically equal subproblems are resolved immediately, leading to failures and trivial solutions. For formalizing the proof of the completeness theorem in PVS, we addressed these problems with the necessary theoretical rigor. The greater complexity of mechanizing verification for anti-unification algorithms lies in how to record the history of decomposing problems into solved and syntactic subproblems, whose regularities (repetitions) are further encoded by storing repeated solved and syntactic subproblems via the substitution solution under construction. The precise notions that guarantee the construction of a solution more specific than any arbitrary generalizer involve elaborate invariant properties related to the preservation of the problem (Lemma 7), history of the computed solution (Lemma 9), preservation of the extended problem and coherence of the partial solution generated during the computation (Lemmas 13, and 15). In addition, by definition of a restricted notion of arbitrary generalizers (Definition 19), the main completeness result is obtained as a corollary (Corollary 21) of the completeness for such a class of generalizers (Theorem 20). All results presented in Subsection 3.1 are already completely specified and formally verified in PVS.

Future work

Work in progress includes the formalization in PVS of the definitions and lemmas presented in Section 3.2 to obtain a mechanically and formally verified proof of completeness for the functional rule-based algorithm `Antiunify`, from which certified executable code can be extracted. Additional work includes the extension of the formalization for verifying algorithms to anti-unification modulo various algebraic properties widely used in mathematics, such as commutativity, associativity [1, 2], absorption [7, 26], and their combinations.

Acknowledgments

This work was partially supported by the Austrian Science Fund (FWF) project P 35530, the Rustaveli National Science Foundation of Georgia under project FR-25-5957, the Brazilian Research Council CNPq Universal grant 407461/2025-6, and Research productivity grants 313290/2021-0 and 6/2026-7, and a PDSE scholarship of the Brazilian Higher Education Council (CAPES) Finance Code 001 to the last author.

References

[1] María Alpuente, Santiago Escobar, Javier Espert, and José Meseguer. ACUOS: A System for Modular ACU Generalization with Subtyping and Inheritance. In *European Conference on Logics in Artificial Intelligence*, volume 8761 of *Lecture Notes in Computer Science*, pages 573–581, 2014, doi: 10.1007/978-3-319-11558-0_40.

[2] María Alpuente, Santiago Escobar, José Meseguer, and Julia Sapiña. Order-Sorted Equational Generalization Algorithm Revisited. *Annals of Mathematics and Artificial Intelligence*, 90(5):499–522, 2022, doi: 10.1007/s10472-021-09771-1

[3] María Alpuente, Santiago Escobar, José Meseguer, and Pedro Ojeda. Order-Sorted Generalization. In *17th International Workshop on Functional and (Constraint) Logic Programming*, volume 246 of *Electronic Notes in Theoretical Computer Science*, pages 27–38. Elsevier, 2008, doi: 10.1016/j.entcs.2009.07.013.

[4] María Alpuente, Santiago Escobar, Javier Espert, and José Meseguer. A Modular Order-Sorted Equational Generalization Algorithm. *Information and Computation*, 235:98–136, 2014, doi: 10.1016/j.ic.2014.01.006.

[5] Eva Armengol. Usages of Generalization in Case-Based Reasoning. In *Case-Based Reasoning Research and Development*, volume 4626 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2007, doi: 10.1007/978-3-540-74141-1_3.

[6] Andréia Borges Avelar, André Luiz Galdino, Flávio Leonardo Cavalcanti de Moura, and Mauricio Ayala-Rincón. First-order Unification in the PVS Proof Assistant. *Logic Journal of the IGPL*, 22(5):758–789, 2014, doi: 10.1093/jigpal/jzu012.

[7] Mauricio Ayala-Rincón, David M. Cerna, Andrés Felipe González Barragán, and Temur Kutsia. Equational Anti-Unification over Absorption Theories. In *Automated Reasoning - 12th International Joint Conference, Part II*, volume 14740 of *Lecture Notes in Computer Science*, pages 317–337. Springer, 2024, doi: 10.1007/978-3-031-63501-4_17.

[8] Mauricio Ayala-Rincón, Washington de Carvalho Segundo, Maribel Fernández, Gabriel Ferreira Silva, and Daniele Nantes-Sobrinho. Formalising Nominal C-Unification Generalised with Protected Variables. *Mathematical Structures in Computer Science*, 31(3):286–311, 2021, doi: 10.1017/S0960129521000050.

[9] Mauricio Ayala-Rincón, Thaynara Arielly de Lima, Maria Júlia Dias Lima, Mariano Miguel Moscato, and Temur Kutsia. Verification of an Anti-Unification Algorithm in PVS. In *NASA Formal Methods - 17th International Symposium*, volume 15682 of *Lecture Notes in Computer Science*, pages 54–71. Springer, 2025, doi: 10.1007/978-3-031-93706-4_4.

[10] Mauricio Ayala-Rincón, Maribel Fernández, and Ana Cristina Rocha Oliveira. Completeness in PVS of a Nominal Unification Algorithm. In *10th Workshop on Logical and Semantic Frameworks with Applications*, volume 323 of *Electronic Notes in Theoretical Computer Science*, pages 57–74. Elsevier, 2015, doi: 10.1016/j.entcs.2016.06.005.

[11] Mauricio Ayala-Rincón, Maribel Fernández, Gabriel Ferreira Silva, Temur Kutsia, and Daniele Nantes-Sobrinho. Nominal AC-Matching. In *16th International Conference on Intelligent Computer Mathematics*, volume 14101 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2023, doi: 10.1007/978-3-031-42753-4_4.

[12] Mauricio Ayala-Rincón, Maribel Fernández, Gabriel Ferreira Silva, Temur Kutsia, and Daniele Nantes-Sobrinho. Certified First-Order AC-Unification and Applications. *Journal of Automated Reasoning*, 68(25):1–48, 2024, doi: 10.1007/s10817-024-09714-5.

[13] Mauricio Ayala-Rincón, Maribel Fernández, Gabriel Ferreira Silva, and Daniele Nantes-Sobrinho. A Certified Functional Nominal C-Unification Algorithm. In *29th International Symposium on Logic-Based Program Synthesis and Transformation*, volume 12042 of *Lecture Notes in Computer Science*, pages 123–138. Springer, 2020, doi: 10.1007/978-3-030-45260-5_8.

[14] Mauricio Ayala-Rincón, Maribel Fernández, Gabriel Ferreira Silva, and Daniele Nantes Sobrinho. A Certified Algorithm for AC-Unification. In *7th International Conference on Formal Structures for Computation and Deduction*, volume 228 of *Leibniz International Proceedings in Informatics*, pages 8:1–8:21, 2022, doi: 10.4230/LIPIcs.FSCD.2022.8.

[15] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998, doi: 10.1017/CBO9781139172752.

[16] Franz Baader and Wayne Snyder. Unification Theory. In *Handbook of Automated Reasoning*. Elsevier, 2001, doi: 10.1016/B978-044450813-3/50010-2.

[17] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to Fix Bugs Automatically. *Proceedings of the ACM on Programming Languages*, 3:159:1–159:27, 2019, doi: 10.1145/3360585.

[18] Adam David Barwell, Cristopher Brown, and Kevin Hammond. Finding Parallel Functional Pearls: Automatic Parallel Recursion Scheme Detection in Haskell Functions via Anti-Unification. *Future Generation Computer Systems*, 79:669–686, 2018, doi: 10.1016/j.future.2017.07.024.

[19] Alexander Baumgartner and Temur Kutsia. A Library of Anti-Unification Algorithms. In *European Conference on Logics in Artificial Intelligence*, volume 8761 of *Lecture Notes in Computer Science*, pages 543–557, 2014, doi: 10.1007/978-3-319-11558-0_38.

[20] Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. A Variant of Higher-Order Anti-Unification. In *24th International Conference on Rewriting Techniques and Applications*, volume 21 of *Leibniz International Proceedings in Informatics*, pages 113–127, 2013, doi: 10.4230/LIPIcs.RTA.2013.113.

[21] Peter. E. Bulychev, Egor V. Kostylev, and Vladimir A. Zakharov. Anti-Unification Algorithms and Their Applications in Program Analysis. In *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, Revised Papers*, volume 5947 of *Lecture Notes in Computer Science*, pages 413–423. Springer, 2009, doi: 10.1007/978-3-642-11486-1_35

[22] David M. Cerna and Temur Kutsia. Anti-Unification and Generalization: A Survey. In *32nd International Joint Conference on Artificial Intelligence*, pages 6563–6573. International Joint Conferences on Artificial Intelligence Organization, 2023, doi: 10.24963/ijcai.2023/736.

[23] Evelyne Contejean. A Certified AC Matching Algorithm. In *Rewriting Techniques and Applications, 15th International Conference*, volume 3091 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2004, doi: 10.1007/978-3-540-25979-4_5.

[24] Reudismam Rolim de Sousa, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. Learning Quick Fixes from Code Repositories. In *35th Brazilian Symposium on Software Engineering*, pages 74–83. ACM, 2021, doi: 10.1145/3474624.3474650.

[25] André Luiz Galdino and Mauricio Ayala-Rincón. A Formalization of the Knuth-Bendix(-Huet) Critical Pair Theorem. *Journal of Automated Reasoning*, 45(3):301–325, 2010, doi: 10.1007/s10817-010-9165-2.

[26] Andrés Felipe González Barragán. *Anti-Unification in Absorptive Theories*. PhD thesis, University of Brasília, 2025. Available at `http://repositorio.unb.br/handle/10482/54148`.

[27] Fabian Huch. Supporting Maintenance of Formal Mathematics with Similarity Search. In *18th International Conference on Intelligent Computer Mathematics*, volume 16136 of *Lecture Notes in Computer Science*, pages 91–109. Springer, 2025, doi: 10.1007/978-3-032-07021-0_6.

[28] Gérard Huet. Résolution d'Équations dans des Langages d'ordre $1, 2, \ldots, \omega$. Thèse d'État, Université de Paris VII, 1976.

[29] Jean-Louis Lassez, Michael J. Maher, and Kim Marriott. Unification Revisited. In *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, 1988, doi: 10.1016/b978-0-934613-40-8.50019-1.

[30] Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, Chetan Bansal, Chandra Maddila, Balasubramanyan Ashok, Sumit Asthana, Christian Bird, and Aditya Kumar. Rex: Preventing Bugs and Misconfiguration in Large Services Using Correlated Change Analysis. In *17th USENIX Symposium on Networked Systems Design and Implementation*, pages 435–448. USENIX Association, 2020.

[31] Ana Cristina Rocha Oliveira, André Luiz Galdino, and Mauricio Ayala-Rincón. Confluence of Orthogonal Term Rewriting Systems in the Prototype Verification System. *Journal of Automated Reasoning*, 58(2):231–251, 2017, doi: 10.1007/s10817-016-9376-2.

[32] Frank Pfenning. Unification and Anti-Unification in the Calculus of Constructions. In *6th Annual Symposium on Logic in Computer Science*, pages 74–85. IEEE Computer Society, 1991, doi: 10.1109/LICS.1991.151632.

[33] Gordon David Plotkin. A Note on Inductive Generalization. *Machine Intelligence*, 5(1):153–163, 1970.

[34] Daniel Ramos, Catarina Gamboa, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. SPELL: Synthesis of Programmatic Edits using LLMs. *CoRR*, abs/2602.01107, 2026, doi: 10.48550/arXiv.2602.01107.

[35] John Charles Reynolds. Transformational Systems and the Algebraic Structure of Atomic Formulas. *Machine Intelligence*, 5(1):135–151, 1970.

[36] José-Luis Ruiz-Reina, José-Antonio Alonso, María-José Hidalgo, and Francisco-Jesús Martín-Mateos. Mechanical Verification of a Rule-Based Unification Algorithm in the Boyer-Moore Theorem Prover. In *1999 Joint Conference on Declarative Programming*, pages 289–304, 1999.

[37] José-Luis Ruiz-Reina, Franciso-Jesús Martín-Mateos, José-Antonio Alonso, and María-José. Hidalgo. Formal Correctness of a Quadratic Unification Algorithm. *Journal of Automated Reasoning*, 37(1-2):67–92, 2006, doi: 10.1007/s10817-006-9030-5.

[38] Wim Vanhoof and Gonzague Yernaux. Generalization-Driven Semantic Clone Detection in CLP. In *Logic-Based Program Synthesis and Transformation - 29th Int. Symposium, Revised Selected Papers*, volume 12042 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 2019, doi: 10.1007/978-3-030-45260-5_14.

[39] Emily Rowan Winter, Vesna Nowack, David Bowes, Steve Counsell, Tracy Hall, Sæmundur Óskar Haraldsson, John R. Woodward, Serkan Kirbas, Etienne Windels, Olayori McBello, Abdurahman Atakishiyev, Kevin Kells, and Matthew W. Pagano. Towards Developer-Centered Automatic Program Repair: Findings from Bloomberg. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1578–1588. Association for Computing Machinery, 2022, doi: 10.1145/3540250.3558953.

Work-in-Progress: A Tactic for Pattern Matching in Autosubst

Mathews George
Heriot-Watt University
Edinburgh, UK
mg2079@hw.ac.uk

Kathrin Stark
Heriot-Watt University
Edinburgh, UK
k.stark@hw.ac.uk

Autosubst enables automatic equality-checking up to the σ-calculus for assumption-free equalities, allowing users to avoid cumbersome reasoning about de Bruijn indices. While effective in many cases, this approach is inapplicable when matching against typing rules, reduction relations, or lemmas, requiring users to either phrase typing rules in a way that they work with Autosubst or even stating explicitly an alternative de Bruijn term. But even without β-reduction, solutions of matching may not be unique. This paper presents a work-in-progress method for automatically pattern matching against assumptions, evaluated on standard case studies including the POPLMark and POPLMark Reloaded challenges.

1 Introduction

De Bruijn terms are a canonical technique for representing terms with binders [11], and have a long history of being used in mechanised formalisations of languages with binders in proof assistants. Despite this, they are often as regarded as cumbersome in practice [6], largely due to the overhead involved in proving and applying the many technical lemmas required for substitution and renaming.

The Autosubst code generator [33, 36] addresses this issue by automating reasoning about de Bruijn terms in the Rocq (formerly Coq) proof assistant in many cases. Given a signature describing a custom syntax, Autosubst automatically proves the substitution lemmas corresponding to the equations in the σ-calculus [1]. Central to the approach, Autosubst comes with a simplification tactic called `asimpl` which normalizes terms according to these equations. As the de Bruijn algebra forms a sound and complete model of the σ-calculus [32], and as the rewriting system is convergent [10], equality in the de Bruijn algebra is decidable by this tactic. In summary, this allows users to work with de Bruijn syntax without manually applying many technical lemmas.

However, this approach becomes less effective in situations where we want to unify a goal with an assumption, in Rocq's case via the `apply` or `eapply` tactic. In particular, Autosubst's automation is designed for assumption-free reasoning and does not support matching modulo σ-equivalence. As a consequence, the user must compensate for these limitations in their proof scripts. Common workarounds include stating rules in an indirect form to facilitate matching, introducing auxiliary lemmas with the right instantiations, and manually transforming goals during proofs to make them applicable, together with directly defining the values of appearing existential variables.

A representative example, is the formulation of the rule for type application in System $F_{<:}$ in the POPLMark solution based on Autosubst:

$$T\_Tapp\ :\ \forall\, \Delta\, \Gamma\, s\, A\, A'\, B\, B',\ \ \Delta;\Gamma \vdash s : \forall A.B \to\ \Delta \vdash A' <: A \to\ B' = B[A' \cdot id] \\ \to \Delta;\Gamma \vdash s\, A' : B'$$

Hermant, Tourret (Eds.): International Workshop on Logical Frameworks and Meta-Languages, Theory and Practice (LFMTP 26)
EPTCS 448, 2026, pp. 47–57, doi:10.4204/EPTCS.448.4

Here, the result type is expressed using a fresh meta-variable B' rather than the more direct expression $B[A' \cdot id]$. While this formulation enables the use of the `eapply` tactic, it shifts the burden to the user, who (when applying this rule to a goal) must subsequently discharge the additional equality premise. This premise typically contains existential variables, and the `asimpl` tactic is only helpful in the case that the user can give the value of these existential variables upfront.

For example, proving the context morphism lemma in the POPLMark challenge, in the corresponding case we get stuck with the following goal

$$\Delta; \Gamma \vdash s[\sigma; \tau]\ A'[\sigma] : B[A'[\sigma] \cdot \sigma]$$

where, when applying `T_Tapp`, we need to solve the equation $B[A[\sigma] \cdot \sigma] = ?B[A[\sigma] \cdot id]$ with a constraint on $?B$ as $\Delta; \Gamma \vdash s[\sigma] : \forall A[\sigma].?B$.

Such indirect formulations are pervasive. They arise not only in typing rules, but also in reduction relations [15, 2], weakening and renaming lemmas, and even in mechanisations of dependent type theory [3]. Even when rules are stated directly, auxiliary lemmas are often introduced solely to make them compatible with `apply`.

This raises a natural question: can we recover the level of automation provided by Autosubst while supporting matching modulo σ-equivalence in the presence of assumptions?

The problem is inherently difficult. Moura et al. showed that Dowek et al.'s method [13] does not decide second-order matching for the full $\lambda\sigma$-calculus [25], but even without β, solutions are not necessarily unique. In practice, this means that a matching problem may admit multiple candidates, not all of which satisfy the premises in the context.

Example 1 (Matching up to σ-equivalence is not unique.)**.** The σ-matching problem we need to solve is constrained by the premises in which the unknown variables appear. In the above example, we need to solve the equation $B[A[\sigma] \cdot \sigma] = ?B[A[\sigma] \cdot id]$ with a constraint on $?B$ as $\Delta; \Gamma \vdash s[\sigma] : \forall A[\sigma].?B$. We have at least two solutions for $?B$, $B[\Uparrow\ \sigma]$ and $B[A[\sigma \circ \uparrow] \cdot \sigma \circ \uparrow]$. The latter solution does not hold with the premise.

As a consequence, any practical solution must rely on heuristics. In the worst case, an adversarial example can always be constructed where a heuristic selects an incorrect solution, even when a correct one exists.

Contributions. We extend the Autosubst framework with a tactic for pattern-matching modulo σ-equivalence. Our main contributions are as follows:

1. We introduce a heuristic tactic `as_apply`, intended as a replacement for the `apply/eapply` tactic in Autosubst-based developments.
2. We generalize this tactic to support the full input language of Autosubst.
3. We evaluate our approach on several case studies, including the entire POPLMark and POPLMark Reloaded Challenge and show that despite its theoretical limitations, first results suggest it performs well in practice.

The development including all results is available online [18].

2 Background

We consider de Bruijn terms for the λ-calculus [11]:

$$s,t \in \mathbb{T} ::= n \mid s\ t \mid \lambda.s \qquad (n \in \mathbb{N})$$

where $s\ t$ denotes application, $\lambda.s$ abstraction, and n are de Bruijn indices which serve as variables bound by abstraction.

Parallel substitutions σ, τ are total functions mapping indices to terms. Intuitively, a substitution can be seen as an infinite sequence of terms. Sometimes, we need to deal with a special kind of substitutions, written ξ, ζ, that map indices to indices known as *renamings*.

A *de Bruijn algebra* is a first-order two-sorted algebra formed of de Bruijn terms and substitutions, and certain constants and operations from the σ-calculus of Abadi et al. [1]. The central operation is instantiation: $s[\sigma]$ replaces the free indices of the term s with the terms provided by the substitution σ.

The σ-calculus comes with a defined set of primitives. The cons operation $s \cdot \sigma$ prepends a term s to a substitution σ. The composition of substitutions is defined as $(\sigma \circ \tau)\ n := (\sigma\ n)[\tau]$. We further have constant substitutions such as the identity $id := \lambda n.n$ and the shift $\uparrow := \lambda n.(n+1)$. The lifting of a substitution is defined as and used as notation for $\Uparrow\ \sigma := 0 \cdot \sigma \circ \uparrow$.

The σ-calculus is equipped with a directed set of equational rules over these primitives (Figure 1). We denote $s \equiv_{\sigma'} t$ if s and t are provably equal by rewriting with a fragment σ' of σ-rules. We denote by $\sigma_{\texttt{min}}$ the fragment consisting of the rules inside the box in Figure 1, which characterise instantiation and its compositionality properties.

Schäfer et al. showed that the de Bruijn algebra forms a sound and complete model of the σ-calculus [32], and, as the rewriting system induced by these rules is known to be convergent [10], equality in the de Bruijn algebra is decidable by normalisation with respect to the σ-rules.

We also require notions from σ-matching [13, 7]. A σ-substitution θ is a function from the object variables to σ-expressions such that $\theta\ x \neq x$ for finitely many x. Its domain is $\mathscr{D}(\theta) := \{x \mid \theta\ x \neq x\}$. We write θ as $\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$ where $x_i \in \mathscr{D}(\theta)$ and denote its application to a σ-expression s simply as $\theta\ s$. Two substitutions θ_1 and θ_2 are σ'-equal iff $\theta_1\ x \equiv_{\sigma'} \theta_2\ x$ for all x. Otherwise, θ_1 and θ_2 are σ'-different.

We distinguish between *bound* and *free* object variables. The free variables $\mathscr{F}(s)$ are the ones we need to resolve in a unification problem [13]. We use the notation $s =_{\sigma'} s'$ to denote a matching equation in a fragment σ' of the σ-calculus where s does not have free variables. We say $s =_{\sigma'} t$ has a solution θ iff $s \equiv_{\sigma'} \theta\ t$ and $\mathscr{D}(\theta) \subseteq \mathscr{F}(t)$. We call a σ'-matching algorithm σ-complete iff it computes all σ-different solutions to a σ'-matching problem. In our intended setting, bound variables correspond to abstract meta-variables, while free variables correspond to unresolved existential variables in a matching problem.

$$\boxed{\begin{aligned}(s\ t)[\sigma] &\equiv s[\sigma]\ t[\sigma]\\ (\lambda.s)[\sigma] &\equiv \lambda.(s[\Uparrow \sigma])\\ s[\sigma][\tau] &\equiv s[\sigma \circ \tau]\\ (s \cdot \sigma) \circ \tau &\equiv s[\tau] \cdot \sigma \circ \tau\\ (\sigma \circ \tau) \circ \rho &\equiv \sigma \circ \tau \circ \rho\end{aligned}}\qquad \begin{aligned}s[id] &\equiv s\\ \sigma \circ id &\equiv \sigma\\ id \circ \sigma &\equiv \sigma\\ 0 \cdot \uparrow &\equiv id\end{aligned}\qquad \begin{aligned}0[s \cdot \sigma] &\equiv s\\ \uparrow \circ (s \cdot \sigma) &\equiv \sigma\\ 0[\sigma] \cdot \uparrow \circ \sigma &\equiv \sigma\end{aligned}$$

Figure 1: The rewriting system of the σ-calculus

3 A Pattern Matching Tactic for Autosubst

We describe a general pattern-matching tactic `as_apply` for terms of the λ-calculus. The tactic is implemented in Rocq's tactic language Ltac.

Let P be an n-ary relation, and consider the following Rocq goal:

H : $\forall$ <qvars>, <premises> $\to P\ t_1\ t_2\ \ldots\ t_n$

$P\ s_1\ s_2\ \ldots\ s_n$

with H an assumption in the context we want to apply, where <qvars> and <premises> denote the quantified variables and premises of H.

A call of `as_apply` H proceeds in two phases: preprocessing and matching.

Preprocessing. We first normalise both the goal and the conclusion of H with respect to the σ-calculus using the `asimpl` tactic provided by Autosubst.

Next, all quantified variables are replaced with existential variables (evars) of the corresponding types, which will be instantiated during matching. The premises of H are then turned into subgoals, to be solved after matching with instantiated quantified variables.

The actual matching will be between the corresponding s_i and t_i. In a first step, we try whether this can be solved by pure syntactic matching using Rocq's tactic. Furthermore, if s_i uses Autosubst's explicit renamings (i.e., is of the form $s\langle\xi\rangle$ or $s[\xi]$) but t_i is not, we change s_i to $s[\xi]$ ($s\langle\xi\rangle$) using `substify` (`renamify`). These are Autosubst-provided tactics that perform conversion between instantiation and renaming operations.

Matching Phase. After preprocessing, we obtain a σ-matching problem

$$\mathscr{M} = \{s_i =_\sigma t_i \mid s_i \text{ and } t_i \text{ are } \sigma\text{-expressions}\}$$

We process equations in $\mathscr{M}$ from left to right.

For each equation $s_i =_\sigma t_i$, we first try to match in the $\sigma_{\texttt{min}}$ fragment using a procedure $\sigma_{\texttt{min}}(s_i,\ t_i)$. If successful, this yields a substitution θ, which is applied to all remaining equations t_k for $k \geq i$.

The function $\sigma_{\texttt{min}}(s,\ t)$ is a partial procedure that computes substitutions by matching modulo the $\sigma_{\texttt{min}}$ fragment. It proceeds as follows:

- If s and t match syntactically, a substitution is returned.
- Otherwise, t is rewritten using $\sigma_{\texttt{min}}$ rules (Figure 1), and matching is retried.

In Figure 2, we give an operational description of the $\sigma_{\texttt{min}}$-function. The function performs a backtracking search over possible rewrites of t until a match with s is found. For example, $\sigma_{\texttt{min}}(s[\sigma\circ\tau],\ ?s[\tau])$ matches first with the seventh case, but the execution path stemming from it does not result in a σ-substitution. The execution backtracks and continues by matching with the tenth case, resulting in $\{?s \mapsto s[\sigma]\}$.

If $\sigma_{\texttt{min}}(s,t)$ fails, we apply a small set of heuristics capturing common patterns in developments:

- If $s_i =_\sigma t_i$ is of the form $s[t\cdot\sigma] =_\sigma ?s[t\cdot id]$, instantiate t_k with $\{?s \mapsto s[\Uparrow\ \sigma]\}$, for $k \geq i$.

$$
\begin{aligned}
\sigma_{\mathtt{min}}(s,\ s) &\Rightarrow \{\} \\
\sigma_{\mathtt{min}}(s,\ ?s) &\Rightarrow \{?s \mapsto s\} \\
\sigma_{\mathtt{min}}(\lambda.s,\ \lambda.s') &\Rightarrow \sigma_{\mathtt{min}}(s,\ s') \\
\sigma_{\mathtt{min}}(s\ t,\ s'\ t') &\Rightarrow \sigma_{\mathtt{min}}(t,\ \sigma_{\mathtt{min}}(s,\ s')\ t') \cup \sigma_{\mathtt{min}}(s,\ s') \\
\sigma_{\mathtt{min}}(\sigma \circ \tau,\ \sigma' \circ \tau') &\Rightarrow \sigma_{\mathtt{min}}(\tau,\ \sigma_{\mathtt{min}}(\sigma,\ \sigma')\ \tau') \cup \sigma_{\mathtt{min}}(\sigma,\ \sigma') \\
\sigma_{\mathtt{min}}(s \cdot \sigma,\ t \cdot \tau) &\Rightarrow \sigma_{\mathtt{min}}(\sigma,\ \sigma_{\mathtt{min}}(s,\ t)\ \tau) \cup \sigma_{\mathtt{min}}(s,\ t) \\
\sigma_{\mathtt{min}}(s[\sigma],\ s'[\sigma']) &\Rightarrow \sigma_{\mathtt{min}}(\sigma,\ \sigma_{\mathtt{min}}(s,\ s')\ \sigma') \cup \sigma_{\mathtt{min}}(s,\ s') \\
\sigma_{\mathtt{min}}(\lambda.s[\Uparrow\ \sigma],\ s'[\sigma']) &\Rightarrow \sigma_{\mathtt{min}}((\lambda.s)[\sigma],\ s'[\sigma']) \\
\sigma_{\mathtt{min}}(s[\sigma]\ t[\sigma],\ s'[\sigma]) &\Rightarrow \sigma_{\mathtt{min}}((s\ t)[\sigma],\ s'[\sigma]) \\
\sigma_{\mathtt{min}}(s[\sigma \circ \tau],\ s'[\sigma']) &\Rightarrow \sigma_{\mathtt{min}}(s[\sigma][\tau],\ s'[\sigma]) \\
\sigma_{\mathtt{min}}(\sigma \circ \tau \circ \rho,\ \sigma' \circ \tau') &\Rightarrow \sigma_{\mathtt{min}}((\sigma \circ \tau) \circ \rho,\ \sigma' \circ \tau') \\
\sigma_{\mathtt{min}}(s[\tau] \cdot \sigma \circ \tau,\ \sigma' \circ \tau') &\Rightarrow \sigma_{\mathtt{min}}((s \cdot \sigma) \circ \tau,\ \sigma' \circ \tau')
\end{aligned}
$$

Figure 2: The $\sigma_{\mathtt{min}}$-function for matching in the $\sigma_{\mathtt{min}}$ fragment

- If $s_i =_\sigma t_i$ is of the form $s =_\sigma ?s[?\sigma]$, instantiate t_k with $\{?s \mapsto s,\ \sigma \mapsto id\}$ respectively, for $k \geq i$.
- If $s_i =_\sigma t_i$ is of the form $s =_\sigma ?s[id]$, instantiate t_k with $\{?s \mapsto s\}$ respectively, for $k \geq i$.
- If $s_i =_\sigma t_i$ is of the form $s =_\sigma s[?\sigma]$, instantiate t_k with $\{?\sigma \mapsto id\}$ respectively, for $k \geq i$.

We again σ-normalize H with the potentially instantiated existential variables. The tactic succeeds if the goal and H are syntactically equal at this point.

3.1 Generalisation

Autosubst [36] is a code generator that takes a second-order HOAS [28] representation of a custom syntax and generates a model of extended σ-calculus comprising multiple mutually inductive sorts, vector substitutions, and first-class renamings.

This tactic has been generalised to work for the input language of the Autosubst compiler. The $\sigma_{\mathtt{min}}$ fragment describes the instantiation operation and has compositionality laws. Hence, for a general syntax, the $\sigma_{\mathtt{min}}$-function has congruence cases, and cases corresponding to the instantiation and compositionality laws.

If $\sigma_{\mathtt{min}}(s,t)$ fails, we check whether $s =_\sigma t$ matches with certain equations that are recurring in developments. In the case of the first equation $s[t \cdot \sigma] =_\sigma ?s[t \cdot id]$, the expression $s[t \cdot id]$ denotes the elimination of the abstraction $\lambda.s$ by substituting t for the bound index. We generate a case accordingly for each binder in the syntax. For example, if we have a polyadic binder $\lambda_2 : (tm \to\ tm \to tm) \to tm$, we generate a case for the equation $s[t_1 \cdot t_2 \cdot \sigma] =?s[t_1 \cdot t_2 \cdot id]$. In the solution $\{?s \mapsto s[\Uparrow\ \sigma]\}$ for the λ-abstraction, the substitution $\Uparrow \sigma$ comes from the instantiation law $(\lambda.s)[\sigma] = \lambda.s[\Uparrow\ \sigma]$. For a general binder, this would be the lifted substitution vector formed during instantiation. In the case of λ_2, the solution would be $\{?s \mapsto s[\Uparrow\Uparrow \sigma]\}$ because $(\lambda_2.s)[\sigma] = \lambda_2.s[\Uparrow\Uparrow\ \sigma]$.

Note that as an additional difficulty since Autosubst supports first-class renamings, the $\sigma_{\mathtt{min}}$-function and the specific equations have cases for the renaming operation, which we omit here for conciseness.

3.2 Current Limitations

We summarise some of the limitations of matching in the $\sigma_{\texttt{min}}$ fragment and the `as_apply` tactic.

First, note that, per se, the $\sigma_{\texttt{min}}$ fragment is not a confluent rewrite system. The expression $(\lambda.s)[\sigma][\tau]$ reduces to both $\lambda.s[0[\Uparrow\ \tau]\cdot\sigma\circ\uparrow\circ\Uparrow\tau]$ and $\lambda.s[0\cdot\sigma\circ\tau\circ\uparrow]$ via $\sigma_{\texttt{min}}$ rules. But, they are not joinable as $\sigma_{\texttt{min}}$ lacks the rules $0[s\cdot\sigma]\to s$ and $\uparrow\circ(s\cdot\sigma)\to\sigma$. Of course they are joinable via the whole σ-calculus.

Note that $\sigma_{\texttt{min}}$-matching does not have unitary solutions in general. An example is $s[\sigma\circ\tau]=_{\sigma_{\texttt{min}}}?s[?\sigma]$ which has two solutions $\{?s\mapsto s,\ ?\sigma\mapsto\sigma\circ\tau\}$ and $\{?s\mapsto s[\sigma],\ ?\sigma\mapsto\tau\}$.

Generally, `as_apply` is not σ-complete. In the above example, it would only produce the first solution. `as_apply` is not complete even as a decider. It fails for the equation $t=_\sigma?s[t\cdot\sigma]$ though there is the solution $\{?s\mapsto 0\}$. Note that this would rarely be a problem in a practical development.

4 Case Studies

We have tested the tactic on Rocq solutions of the POPLMark [6] and POPLMark Reloaded [2] challenges, two benchmarks on reasoning with binders. In this section, we present examples from our modified solutions. The linked development contains both the original and new solutions.

Example 2. When proving substitutivity of multi-step in the POPLMark Reloaded, we have to prove:

$$\begin{array}{l}mstep\_inst\ :\ \forall\ s\ t\ \sigma,\ \ s\gg t\to\ s[\sigma]\gg t[\sigma]\\ H\ :\ \forall n,\ \ \sigma\ n\gg\tau\ n\\ \hline (\sigma\ i)\langle\uparrow\rangle\gg(\tau\ i)\langle\uparrow\rangle\end{array}$$

The `apply` tactic fails if we try to apply *mstep_inst* because the renaming operation does not match with the instantiation operation, meaning that originally, the renaming operation has to be manually changed to an instantiation. We apply *mstep_inst* with the `as_apply` tactic. After pre-processing, the goal has changed to $(\sigma\ i)[\uparrow]\gg(\tau\ i)[\uparrow]$. The $\sigma_{\texttt{min}}$-function first solves the equation $(\sigma\ i)[\uparrow]=?s[?\sigma]$ resulting in $\{?s\mapsto\sigma\ i,?\sigma\mapsto\uparrow\}$, and solves next $(\tau\ i)[\uparrow]=?t[\uparrow]$ resulting in $\{?t\mapsto\tau\ i\}$.

Example 3. Consider the following case in the context morphism lemma in the POPLMark challenge:

$$\begin{array}{l}T\_Tapp\ :\ \forall\ \Delta\ \Gamma\ s\ A\ A'\ B,\ \ \Delta;\Gamma\vdash s:\forall A.B\to\ \Delta\vdash A'<:A\to\ \Delta;\Gamma\vdash s\ A':B[A'\cdot id]\\ H_1\ :\ s[\sigma;\tau]\ :\ (\forall A.B)[\sigma]\\ H_2\ :\ \Delta\vdash A'[\sigma]:A[\sigma]\\ \hline \Delta;\Gamma\vdash s[\sigma;\tau]\ A'[\sigma]:B[A'[\sigma]\cdot\sigma]\end{array}$$

The `apply` tactic fails when applying *T_Tapp*, and this is the reason for the originally indirect definition of *T_Tapp*. We apply *T_Tapp* with `as_apply`. After the pre-processing steps, we have to match $\Delta;\Gamma\vdash s[\sigma;\tau]\ A'[\sigma]:B[A'[\sigma]\cdot\sigma]$ with $?\Delta;?\Gamma\vdash?s\ ?A':?B[?A'\cdot id]$. The equation $s[\sigma;\tau]\ A'[\sigma]=?s\ ?A'$ matches syntactically and $\sigma_{\texttt{min}}$-function results in $\{?s\mapsto s[\sigma;\tau],?A'\mapsto A'[\sigma]\}$. But, $B[A'[\sigma]\cdot\sigma]=?B[A[\sigma]\cdot id]$ can't be matched with $\sigma_{\texttt{min}}$-rules and hence $\sigma_{\texttt{min}}$-function fails. However, it has the form $s[t\cdot\sigma]=?s[t\cdot id]$ described in the last section. Hence, we resolve $?B$ as $B[\Uparrow\sigma]$, and $B[\Uparrow\sigma][A'[\sigma]]$ normalizes to $B[A'[\sigma]\cdot\sigma]$.

Example 4. When proving transitivity of subtyping (POPLMark), we have:

$$\begin{array}{l}sub\_weak\ :\ \forall\Delta\ \Delta'\ \xi\ A_1\ A_2,\ \ \Delta\vdash A_1<:A_2\to\ (\forall x.(\Delta\ x)\langle\xi\rangle=\Delta'(\xi\ x))\to\ \Delta'\vdash A_1\langle\xi\rangle<:A_2\langle\xi\rangle\\ H\ :\ \Delta\vdash B\langle\uparrow\rangle<:B'\langle\uparrow\rangle\\ \hline (B\langle\uparrow\rangle\cdot\Delta\circ\uparrow)\vdash B\langle\uparrow\rangle<:B'\langle\xi\circ\uparrow\rangle\end{array}$$

We need to apply *sub_weak*, which is the weakening lemma for subtyping. Originally, *sub_weak* has two versions: one with the above statement, and an auxiliary lemma that has the conclusion $\Delta' \vdash A'_1 <: A'_2$ with the extra premises $A'_1 = A_1\langle\xi\rangle$ and $A'_2 = A_2\langle\xi\rangle$ which is applied in this case. We proceed with `as_apply`. All equations match syntactically except $B'\langle\xi \circ \uparrow\rangle =?A_2\langle\uparrow\rangle$. Since $B'\langle\xi \circ \uparrow\rangle$ can be decomposed as $B'\langle\xi\rangle\langle\uparrow\rangle$, the $\sigma_{\texttt{min}}$-function resolves $B'\langle\xi \circ \uparrow\rangle =?A_2\langle\uparrow\rangle$ as $\{?A_2 \mapsto B'\langle\xi\rangle\}$.

Example 5. We look at the proof of *ty_subst* from the POPLMark challenge:

$$\frac{\begin{array}{l} \mathit{sub\_subst} \;:\; \forall \Delta\, \Delta'\, \sigma\, A\, B,\;\; \Delta \vdash A <: B \to\; (\forall x. \Delta' \vdash \sigma\, x <: (\Delta\, x)[\sigma]) \to\; \Delta' \vdash A[\sigma] <: B[\sigma] \\ H \;:\; \Delta \vdash A' <: A \end{array}}{\Delta' \vdash A' <: A}$$

Here, we have a goal in uninstantiated form and an assumption in instantiated form. Originally, A and A' are manually changed to $A[id]$ and $A'[id]$ before applying *sub_subst*. We apply *sub_subst* with `as_apply`. Eventually, we need to match $A' =?A[?\sigma]$ and $A =?B[?\sigma]$. While the $\sigma_{\texttt{min}}$-function fails in both cases, our additional heuristics resolve $?A$, $?B$ and $?\sigma$ as A', A and id respectively.

Discussion. Even in relatively short proof scripts such as the POPLMark challenge (642 lines) and POPLMark Reloaded (683 lines), the `as_apply` is used frequently: 15 times in the POPLMark B and 10 times in the POPLMark Reloaded. In all cases, it successfully solves the intended goal. Compared to the previous solutions, the proposed solutions with the matching tactics avoid the need for indirect definitions, auxiliary lemmas, and the need to manually transform the goal or find the instances for existential variables.

Additional (single) examples, for example, as one in a formalisation of Martin-Löf Type Theory [3], can be found in the appendix. These developments require a customised matching tactic due to manual adaptations of the substitution primitives in the original developments. Apart from these technical problems, both in this case study and in a development of the call-by-push-value [15], we have so far not found any essential problems with the tactic.

5 Related Work

Mechanizing syntax has a long history, leading to many syntax representations [11, 4, 23, 28, 30, 19, 31] and supporting tools [33, 36, 5, 21, 35, 37, 8, 29, 17, 34]. Autosubst [33, 36] provides automation for de Bruijn syntax and has been used in several mechanizations [2, 15, 3, 14].

Unification in languages with binders has been studied extensively [12]. Traditionally, unification is defined modulo $\beta\eta$-equivalence, known as higher-order unification. While higher-order unification is undecidable in general, Huet's algorithm works well in practice [20]. Miller et al. [24] identify a fragment named *higher-order patterns* for which higher-order unification is decidable and is unitary, i.e. admits general unifiers. Higher-order matching is decidable up to fourth order [27] while for higher orders, decidability remains open.

Calculi for explicit substitutions such as the σ-calculus were introduced by Abadi et al. [1] to bridge the gap between the λ-calculus and its implementations. Curien et al. [10] prove that the σ-calculus is a convergent rewriting system, and later Schäfer et al. [32] show that the σ-calculus is a sound and complete model for the de Bruijn algebra. Together, this means the σ-calculus has practical usage as a rewriting system to decide equality in languages with binders as in the Autosubst library [33, 36]. However, these techniques do not directly extend to matching problems.

Dowek et al. [13] show that higher-order unification is reducible to unification in the first-order equational theory of $\lambda\sigma$-calculus (σ-calculus with β rule), and provide a general unification method for the $\lambda\sigma$-calculus. Moura et al. [25] show that this method does not decide second-order matching in the $\lambda\sigma$-calculus by providing a non-terminating counter-example. They characterise a fragment for which the method terminates and provide a second-order matching algorithm for this fragment. This is not directly applicable as it covers the σ-calculus, including the β rule.

Nominal syntax [16] provides an alternative representation of languages with binders. Urban et al. [38] show that *nominal unification* is both decidable and unitary. Cheney [9] shows that higher-order pattern unification problems can be solved by encoding them as nominal unification problems, and Levy et al. [22] show the reverse. Nantes-Sobrinho et al. [26] generalise nominal unification problems to *nominal equational problems* and provide a rule-based algorithm to find solutions in the ground nominal algebra [16].

Rocq's original unification algorithm relies on heuristics [39]. Ziliani et al. [39] give a new unification algorithm along with an implementation for Calculus of Inductive Constructions, incorporating canonical structures and universe polymorphism. They formally describe a heuristic called *controlled backtracking* used in the unification algorithm of Rocq.

6 Conclusion and Ongoing Work

We extended the Autosubst framework to generate a pattern-matching tactic for custom syntax. The tactic is intended as a substitute for the `apply` tactic when matching with an assumption up to the substitution calculus. We tested this tactic on standard benchmarks, such as the POPLMark [6] and POPLMark Reloaded [2] challenges, and were able to simplify previous solutions.

The current tactic operates heuristically for pragmatic reasons – the σ-matching problem could potentially have many and even infinite solutions. However, in case the solution is not unique, it could produce an incorrect solution when there is a correct solution. Eventually, we are interested in having a matching tactic with proven guarantees; for example, we are interested in identifying the biggest fragment σ' for which the uniqueness criterion holds: If θ is a solution of $s =_{\sigma'} t$, then any other solution θ' is σ-equal to θ.

References

[1] Martin Abadi, Luca Cardelli, P-L Curien & J-J Lévy (1989): *Explicit substitutions*. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 31–46, doi:10.1145/96709.96712.

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

[3] Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot & Loïc Pujet (2024): *Martin-Löf à la Coq*. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 230–245, doi:10.1145/3636501.3636951.

[4] Brian Aydemir, Arthur Charguéraud, Benjamin C Pierce, Randy Pollack & Stephanie Weirich (2008): *Engineering formal metatheory*. *Acm sigplan notices* 43(1), pp. 3–15, doi:10.1145/1328897.1328443.

[5] Brian Aydemir & Stephanie Weirich (2010): *LNgen: Tool support for locally nameless representations*.

[6] Brian E Aydemir, Aaron Bohannon, Matthew Fairbairn, J Nathan Foster, Benjamin C Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich & Steve Zdancewic (2005): *Mechanized metatheory for the masses: the POPLMark challenge*. In: *Theorem Proving in Higher Order Logics: 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005. Proceedings 18*, Springer, pp. 50–65, doi:10.1007/11541868_4.

[7] Franz Baader & Tobias Nipkow (1998): *Term rewriting and all that*. Cambridge university press, doi:10.1017/CBO9781139172752.

[8] Jan van Brügge, Andrei Popescu & Dmitriy Traytel (2025): *Animating MRBNFs: Truly modular binding-aware datatypes in Isabelle/HOL*. In: *16th International Conference on Interactive Theorem Proving (ITP 2025)*, 352, Sheffield, pp. 11:1–11:20, doi:10.4230/LIPIcs.ITP.2025.11.

[9] James Cheney (2005): *Relating nominal and higher-order pattern unification*. In: *Proceedings of the 19th international workshop on Unification (UNIF 2005)*, LORIA research report A05, pp. 104–119.

[10] Pierre-Louis Curien, Thérese Hardin & Jean-Jacques Lévy (1996): *Confluence properties of weak and strong calculi of explicit substitutions*. *Journal of the ACM (JACM)* 43(2), pp. 362–397, doi:10.1145/226643.226675.

[11] Nicolaas Govert De Bruijn (1972): *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*. *Indagationes mathematicae (proceedings)* 75(5), pp. 381–392, doi:10.1016/1385-7258(72)90034-0.

[12] Gilles Dowek (2001): *Higher-order unification and matching*. *Handbook of automated reasoning* 2, p. 1009, doi:10.1016/B978-044450813-3/50018-7.

[13] Gilles Dowek, Thérese Hardin & Claude Kirchner (1995): *Higher-order unification via explicit substitutions*. In: *Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science*, IEEE, pp. 366–374, doi:10.1006/inco.1999.2837.

[14] Yannick Forster, Dominik Kirst & Dominik Wehr (2021): *Completeness theorems for first-order logic analysed in constructive type theory: Extended version*. *Journal of Logic and Computation* 31(1), pp. 112–151, doi:10.1093/logcom/exaa073.

[15] Yannick Forster, Steven Schäfer, Simon Spies & Kathrin Stark (2019): *Call-by-push-value in Coq: operational, equational, and denotational theory*. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 118–131, doi:10.1145/3293880.3294097.

[16] Murdoch J Gabbay & Andrew M Pitts (2002): *A new approach to abstract syntax with variable binding*. *Formal aspects of computing* 13(3), pp. 341–363, doi:10.1007/s001650200016.

[17] Andrew Gacek (2008): *The Abella interactive theorem prover (system description)*. In: *International Joint Conference on Automated Reasoning*, Springer, pp. 154–161, doi:10.1007/978-3-540-71070-7_13.

[18] Mathews George: *Extended Autosubst compiler, Case studies and Work-in-Progress developments*. `https://github.com/mthwsgrg/lfmtp2026`. (visited on May 1 2026).

[19] Andrew D Gordon (1993): *A mechanisation of name-carrying syntax up to alpha-conversion*. In: *HOL Users' Group Workshop*, Springer, pp. 413–425, doi:10.1007/3-540-57826-9_152.

[20] Gerard P. Huet (1975): *A unification algorithm for typed λ-calculus*. *Theoretical Computer Science* 1(1), pp. 27–57, doi:10.1016/0304-3975(75)90011-0.

[21] Steven Keuchel, Stephanie Weirich & Tom Schrijvers (2016): *Needle & Knot: Binder boilerplate tied up*. In: *European Symposium on Programming*, Springer, pp. 419–445, doi:10.1007/978-3-662-49498-1_17.

[22] Jordi Levy & Mateu Villaret (2008): *Nominal unification from a higher-order perspective*. In: *International Conference on Rewriting Techniques and Applications*, Springer, pp. 246–260, doi:10.1007/978-3-540-70590-1_17.

[23] Conor McBride & James McKinna (2004): *Functional pearl: i am not a number–i am a free variable*. In: *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, pp. 1–9, doi:10.1145/1017472.1017477.

[24] Dale Miller (1991): *A logic programming language with lambda-abstraction, function variables, and simple unification*. Journal of logic and computation 1(4), pp. 497–536, doi:10.1093/logcom/1.4.497.

[25] Flávio LC de Moura, Fairouz Kamareddine & Mauricio Ayala-Rincón (2005): *Second-order matching via explicit substitutions*. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning, Springer, pp. 433–448, doi:10.1007/978-3-540-32275-7_29.

[26] Daniele Nantes-Sobrinho, Maribel Fernandez, Deivid Vale & Mauricio Ayala-Rincón (2025): *A Nominal Approach to Equational Problems in Languages with Binders*. ACM Transactions on Computational Logic 27(1), pp. 1–46, doi:10.1145/3767744.

[27] Vincent Padovani (2000): *Decidability of fourth-order matching*. Mathematical Structures in Computer Science 10(3), pp. 361–372, doi:10.1017/S0960129500003108.

[28] Frank Pfenning & Conal Elliott (1988): *Higher-order abstract syntax*. ACM sigplan notices 23(7), pp. 199–208, doi:10.1145/960116.54010.

[29] Brigitte Pientka & Jana Dunfield (2010): *Beluga: A framework for programming and reasoning with deductive systems (system description)*. In: International Joint Conference on Automated Reasoning, Springer, pp. 15–21, doi:10.1007/978-3-642-14203-1_2.

[30] Andrew M Pitts (2001): *Nominal logic: A first order theory of names and binding*. In: International Symposium on Theoretical Aspects of Computer Software, Springer, pp. 219–242, doi:10.1007/3-540-45500-0_11.

[31] Piotr Polesiuk & Filip Sieczkowski (2024): *Functorial Syntax for All*.

[32] Steven Schäfer, Gert Smolka & Tobias Tebbi (2015): *Completeness and decidability of de Bruijn substitution algebra in Coq*. In: Proceedings of the 2015 Conference on Certified Programs and Proofs, Association for Computing Machinery, pp. 67–73, doi:10.1145/2676724.2693163.

[33] Steven Schäfer, Tobias Tebbi & Gert Smolka (2015): *Autosubst: Reasoning with de Bruijn terms and parallel substitutions*. In: Interactive Theorem Proving: 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings 6, Springer, pp. 359–374, doi:10.1007/978-3-319-22102-1_24.

[34] Carsten Schürmann (2009): *The Twelf proof assistant*. In: International Conference on Theorem Proving in Higher Order Logics, Springer, pp. 79–83, doi:10.1007/978-3-642-03359-9_7.

[35] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar et al. (2010): *Ott: Effective tool support for the working semanticist*. Journal of functional programming 20(1), pp. 71–122, doi:10.1017/S0956796809990293.

[36] Kathrin Stark, Steven Schäfer & Jonas Kaiser (2019): *Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions*. In: Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 166–180, doi:10.1145/3293880.3294101.

[37] Christian Urban & Cezary Kaliszyk (2012): *General bindings and alpha-equivalence in Nominal Isabelle*. Logical methods in computer science 8, doi:10.2168/LMCS-8(2:14)2012.

[38] Christian Urban, Andrew M Pitts & Murdoch J Gabbay (2004): *Nominal unification*. Theoretical Computer Science 323(1-3), pp. 473–497, doi:10.1016/j.tcs.2004.06.016.

[39] Beta Ziliani & Matthieu Sozeau (2015): *A unification algorithm for Coq featuring universe polymorphism and overloading*. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, pp. 179–191, doi:10.1145/2784731.2784751.

A More Examples

We look at more examples from the MLTT mechanization [3][1], POPLMark Reloaded, and POPLMark to demonstrate the usefulness of our tactic.

Example 6. In the MLTT mechanization, in the proof of *complete_Pi*, we come across the following.

[1] https://github.com/CoqHott/logrel-coq/blob/coq-8.19/theories/LogicalRelation/Neutral.v

$ty\_app\ :\ \forall\ \Gamma\ (f\ d\ dom\ cod : tm),\ \ \Gamma \vdash f : \Pi_{dom}\ cod\ \to \Gamma \vdash d : dom \to \Gamma \vdash tApp\ f\ d : cod[d \cdot id]$
$H\ :\ \Gamma \vdash a : dom\langle\xi\rangle$

$\Gamma \vdash tApp\ n\langle\xi\rangle\ a\ :\ cod[a \cdot \xi]$

Here, we need to apply the type application rule *ty_app* for the Π type. The `apply` tactic would fail, and we resort to `as_apply`. First, we do the pre-processing steps, and all the equations are matched except $cod[a \cdot \xi] = ?cod[a \cdot id]$. This equation can't be matched in the $\sigma_{\texttt{min}}$ fragment. But, it's of the form $s[t \cdot \sigma] = ?s[t \cdot id]$. Since ξ is a renaming, we resolve $?cod$ as $cod\langle \Uparrow \xi\rangle$. After, $cod\langle \Uparrow\ \xi\rangle[a \cdot id]$ is reduced to $cod[a \cdot \xi]$.

Example 7. This example is a continuation from the above example, where we have to prove the subgoal (first premise of *ty_app*) by applying the weakening lemma *ty_wk*.

$ty\_wk\ :\ \forall\ \Gamma\ \Delta\ \xi,\ \vdash \Delta\ \to \Gamma \vdash t : A \to \Delta \vdash t\langle\xi\rangle : A\langle\xi\rangle$

$\Gamma \vdash n\langle\xi\rangle : \Pi_{dom\langle\xi\rangle}\ cod\langle \Uparrow \xi\rangle$

The `apply` tactic fails as expected. We apply the `as_apply` tactic on *ty_wk* and eventually we face the σ-matching equation $\Pi_{dom\langle\xi\rangle}\ cod\langle \Uparrow \xi\rangle = ?A\langle\xi\rangle$. This equation matches in the $\sigma_{\texttt{min}}$ fragment because we have the instantiation law for renaming $(\Pi_{dom}\ cod)\langle\xi\rangle = \Pi_{dom\langle\xi\rangle}\ cod\langle \Uparrow \xi\rangle$. Hence, the $\sigma_{\texttt{min}}$-function resolves $?A \mapsto \Pi_{dom}\ cod$.

Example 8. In the *step_inst* lemma from the POPLMark Reloaded, we face the following.

$\beta\ :\ \forall s\ t,\ \ (\lambda.s)\ t \succ s[t \cdot id]$

$\lambda.s[\Uparrow \sigma]\ t[\sigma] \succ s[t[\sigma] \cdot \sigma]$

The `apply` tactic can syntactically match the reducible expression $(\lambda.s[\Uparrow\ \sigma])\ t[\sigma]$ with $(\lambda.s)\ t$ in the β-rule. But `apply` fails to match $s[t[\sigma] \cdot \sigma]$ with $s[\Uparrow\ \sigma][t[\sigma] \cdot id]$ because it relies on the Rocq unification engine that doesn't convert up to σ-rules. In the case of `as_apply`, we resolve the variables $?s$ and $?t$ as $s[\Uparrow\ \sigma]$ and $t[\sigma]$ respectively. Finally, we do a σ-normalization that reduces $s[\Uparrow\ \sigma][t[\sigma] \cdot id]$ to $s[t[\sigma] \cdot \sigma]$.

Example 9. We look at the proof of the context morphism lemma from the POPLMark. At one point in the proof, we have the context and goal as follows:

$context\_renaming\_lemma\ :\ \forall \Delta'\ \Delta\ \Gamma\ \Gamma'\ s\ A\ \xi\ \zeta\ \ ...\ \ \to \Delta';\Gamma' \vdash s : A \to \Delta;\Gamma \vdash s\langle\xi;\zeta\rangle : A\langle\xi\rangle$

$\Delta';A[\sigma] \cdot \Gamma'\ \vdash\ (\tau\ f)\langle id;\uparrow\rangle : (\Gamma\ f)[\sigma]$

The `apply` tactic fails when we try to apply the context renaming lemma. The `as_apply` proceeds as expected. The equation $(\tau\ f)\langle id;\uparrow\rangle = ?s\langle ?\xi;?\zeta\rangle$ matches syntactically and we resolve $\{?s \mapsto \tau\ f, ?\xi \mapsto id, ?\zeta \mapsto \uparrow\}$. Now, the $\sigma_{\texttt{min}}$-function fails for the equation $(\Gamma\ f)[\sigma] = ?A\langle id\rangle$. But it has the form $s = ?t\langle id\rangle$ (for renamings). Hence, we resolve $?A$ as $(\Gamma\ f)[\sigma]$

A Strategy Language for Controlled Proof Search

Romain Sidhoum Simon Robillard David Delahaye

LIRMM, Univ. Montpellier, CNRS, Montpellier, France
Firstname.Lastname@lirmm.fr

This paper introduces the strategy language of Pgeon, a meta-prover with a clear separation between inference rules and proof search. We give the semantics of strategies as functions over proof states, and of the operators that are used to combine them, allowing for sequential composition, choice, repetition and interleaving of strategies. This language is designed to handle the challenge of fair proof search in semi-decidable logics, where simple depth-first exploration of the proof space is not guaranteed to achieve completeness. We showcase the expressiveness and effectiveness of the approach through case studies in first-order and modal logics.

1 Introduction

In automated reasoning, the completeness of a logical calculus does not guarantee that a proof will be found. In semi-decidable logics, some rules may generate infinitely many successors, and unfair exploration may indefinitely postpone relevant inferences, preventing the discovery of valid proofs. The completeness of the calculus is sometimes referred to as *static completeness*, while *dynamic completeness* describes the property of a prover, i.e., a proof search algorithm applied to a given calculus, to find a proof for any theorem [8].

These issues are particularly visible in systems that separate inference rules from proof-search strategies. While rules describe admissible transformations, strategies determine how they are applied during search, making the design of effective exploration policies non-trivial.

Pgeon [7] is a framework that separates logical rules from proof-search strategies and compiles them into executable provers. This separation highlights the impact of strategy design: the same set of rules may lead to different outcomes depending on how they are explored.

We introduce a formal strategy language for Pgeon, together with a compositional semantics that models proof search as the enumeration of proof states. Strategies are interpreted as functions mapping states to (possibly infinite) streams of successors, capturing both deterministic and non-deterministic behavior. The language provides combinators for sequential composition, choice, repetition, and fair interleaving of strategies.

We illustrate the approach through case studies in first-order and modal logics, showing how naive strategies may fail to terminate and how fair combinators enable systematic exploration of the search space. More generally, the framework provides a basis for reasoning about proof-search procedures at an abstract level, independently of implementation details.

Related Work Our strategy language relates to work on compositional stream processing, monadic nondeterminism, and strategy languages.

Monoidal stream semantics [3] models stream processors within a symmetric monoidal category, providing a principled account of compositionality where complex transformations are built from sim-

Hermant, Tourret (Eds.): International Workshop on Logical Frameworks
and Meta-Languages, Theory and Practice (LFMTP 26)
EPTCS 448, 2026, pp. 58–64, doi:10.4204/EPTCS.448.5

pler ones. Similarly, functional reactive programming (FRP) [4] treats stream transformers as compositional entities combined via structured interfaces. However, these approaches are typically deterministic, whereas our setting is inherently nondeterministic and allows multiple (possibly infinite) results, requiring explicit control over exploration.

The treatment of nondeterministic computations over streams is closely related to the LogicT framework [6], which provides a monadic encoding of backtracking with both biased and fair search. Sequential composition in our language corresponds to Kleisli composition, while our choice operators capture biased and fair exploration. In contrast to LogicT, which provides an operational abstraction, we reify computations as lazy streams and introduce an explicit algebra of strategy combinators.

Our work is also related to tactic languages such as $\mathscr{L}_{tac}$ [2], used in the prover Rocq, where tactics act as transformations over proof states with failure and backtracking. While the analogy is structural, tactic languages typically provide an operational notion of search. In our approach, the search space is represented explicitly as a stream, and exploration strategies are expressed compositionally.

Rewriting-based systems such as Maude [1] similarly model computation as nondeterministic transitions and provide strategy languages for controlling rule application. However, rewriting logic is fundamentally relational, whereas our approach is functional and stream-based, reifying the search space as a first-class object and enabling compositional reasoning about fairness and enumeration.

2 Pgeon

Pgeon [7] generates automated theorem provers from declarative specifications of logical calculi. It is intended to generate provers based on the tableaux method. The specification of a prover defines the syntax, inference rules, and a strategy that describes how rules are applied. From this specification, an executable prover is produced.

This separation allows different proof-search behaviors to be defined over the same calculus, as the rules specify admissible steps while strategies control their exploration. In particular, rule applications are inherently non-deterministic, as they may match multiple formulas, branches, or terms, potentially generating infinitely many successors.

In Pgeon, this search policy is expressed by a strategy. Strategies control the order of rule applications, the exploration of alternatives, and the use of backtracking, making them a central component of the generated prover. The strategy language studied in Section 3 abstracts away from the concrete implementation of this mechanism. We treat a rule application as a function from proof states to streams of successor states. The stream represents the possible outcomes of applying the rule, ordered according to the intended exploration policy. This view makes non-determinism explicit since all possible outcomes are enumerated rather than implicitly explored. A deterministic rule returns a singleton stream, a failing rule returns the empty stream, and a non-deterministic rule returns a stream with several successors. A rule may yield an infinite stream of successors.

This abstraction is useful because it allows strategy operators to be defined compositionally. Sequential composition, choice, repetition, and fair interleaving can all be described as operations on streams of proof states. The resulting semantics does not depend on the stack discipline or continuation representation used by particular implementations.

The need for such a semantics it becomes clear when considering fairness. Suppose a strategy repeatedly applies a universally quantified rule before considering other instances. Even if a contradiction can be found after a finite number of different instantiations, a depth-first strategy may keep generating redundant or irrelevant instances forever. The problem is not that the calculus is incomplete, but that the

strategy unfairly gives unbounded priority to one part of the search space.

For example, consider the clause set:

$$\Gamma = \{\ \forall x.\neg P(x) \vee \neg Q(x),\ P(a),\ P(b),\ Q(b)\ \}$$

A closing derivation requires selecting useful instances of the universal formula and combining them with the atomic facts already present in the branch. However, repeated instantiation with $x \mapsto a$ produces an infinite expansion without progress. The search then follows an infinite path although the contradiction is reachable in the proof space. This phenomenon can be sketched as follows:

$$\dfrac{\dfrac{\Gamma}{\Gamma, \neg P(a)} \quad \dfrac{\dfrac{\Gamma}{\Gamma, \neg Q(a)}\ \forall, \vee, x \mapsto a}{\dfrac{\Gamma, \neg Q(a), \neg P(a)}{\vdots} \quad \dfrac{\Gamma, \neg Q(a), \neg Q(a)}{\vdots}}\ \forall, \vee, x \mapsto a}$$

This motivates the fair strategy combinators introduced in the rest of the paper. Rather than relying on depth-first exploration alone, the language provides operators that interleave the results of several strategies. Interleaving ensures that if a successor state occurs at some finite position in one of the component searches, then it also occurs at some finite position in the combined search, so no branch that can produce a result is postponed indefinitely. In this way, fair composition prevents one infinite branch from starving the exploration of another.

3 Strategy Language

We formalize proof search as the transformation of proof states. A proof state represents a partially constructed proof tree, whose open branches correspond to sets of formulas to be closed. Applying an inference rule expands the proof tree, producing new proof states.

Let Σ be the set of proof states. In general, a proof state can be seen as a tree of formulas, whose branches are either open or closed. The precise structure of Σ is left abstract, as our framework is parametric in the underlying proof system.

Rather than modeling proof search as a relation on states, we adopt a view where a strategy enumerates all possible outcomes of its application. Formally, we write $\mathrm{Str}(\Sigma)$ for the set of (finite or infinite) streams over Σ, and define a strategy as a function:

$$s : \Sigma \to \mathrm{Str}(\Sigma)$$

Given a state $\sigma \in \Sigma$, the stream $s(\sigma)$ represents all possible successor states obtained by applying s to σ. The empty stream denotes failure, a singleton stream denotes deterministic success, and longer streams represent multiple possible successor states (non-deterministic branching).

Each inference rule r induces a primitive strategy:

$$[\![r]\!] : \Sigma \to \mathrm{Str}(\Sigma)$$

which returns all proof states obtained by applying r for each admissible match of premises. Depending on the rule, the matched premises may either be removed from the state, or preserved. In the latter case, it is often useful to apply the rule exhaustively to all applicable matches. To this end, we introduce a derived operator $r!$, which applies r to all admissible matches in a given state. Its semantics is given by:

$$[\![r!]\!] : \Sigma \to \mathrm{Str}(\Sigma)$$

where $[\![r!]\!](t)$ returns the unique state obtained by exhaustively applying r to all applicable premises in t, if at least one application is possible, and the empty stream otherwise.

Stream operators We assume the following operators on streams:

- The stream operator id such that $\mathrm{id}(t) = \langle t \rangle$
- For a function $f : \Sigma \to \mathrm{Str}(\Sigma)$ and a stream X, $X \gg\!= f$ is the stream obtained by applying f to each element of X and concatenating the results.
- Given a stream of streams $S = \langle S_0, S_1, S_2, \ldots \rangle$ with $S_i = \langle \sigma_{i,0}, \sigma_{i,1}, \ldots \rangle$, we define $\triangle(S)$ the stream that contains every $\sigma_{i,j}$, ordered such that $\sigma_{i,j}$ appears before $\sigma_{i',j'}$ if
  (a) $i + j < i' + j'$, or
  (b) $i + j = i' + j'$ and $i < i'$.

  This construction ensures that every element $\sigma_{i,j}$ appears after finitely many elements in $\triangle(S)$. Thus, if a result appears at a finite depth in any component stream, it appears at a finite depth in the interleaved stream.

Strategy combinators Strategies are built from primitive rules (r) using the following grammar:

$$s \quad ::= \quad r \mid r! \mid s\,;s \mid s \,\|\, s \mid s\,\&;s \mid s\,\&|\,s \mid s^*$$

Let $s, s_1, s_2 : \Sigma \to \mathrm{Str}(\Sigma)$. The semantics of the combinators is defined as follows.

- Sequential composition:
$$[\![s_1\,;s_2]\!](t) = [\![s_2]\!] \gg\!= [\![s_1]\!](t)$$
  This corresponds to applying s_1 to t, and then applying s_2 to each resulting state.
- Left-biased choice:
$$[\![s_1 \,\|\, s_2]\!](t) = \begin{cases} [\![s_1]\!](t) & \text{if } [\![s_1]\!](t) \neq \emptyset, \\ [\![s_2]\!](t) & \text{otherwise} \end{cases}$$
  This operator models deterministic choice with fallback.
- Fair sequential composition:
$$[\![s_1\,\&;s_2]\!](t) = \triangle([\![s_1]\!](t) \gg\!= [\![s_2]\!])$$
  Unlike $\|$, this operator interleaves the exploration of all intermediate results, ensuring that no branch is indefinitely delayed. If $s_2(s_1(t))$ can produce a result in finite steps, then $s_1\,\&;s_2$ will eventually produce it.
- Fair choice:
$$[\![s_1\,\&|\,s_2]\!](t) = \triangle(\langle [\![s_1]\!](t), [\![s_2]\!](t) \rangle)$$
  This operator fairly interleaves the results of both strategies.
- Repetition:
$$s^* = \mu X.\,\mathrm{id}\,\&;(s\,\&;X)$$
  This defines the closure under repeated application of s, with fairness ensuring that all finite iteration depths are explored. Unlike the usual Kleene star based on sequential composition, this definition uses fair sequential composition. As a result, the exploration of derivations is interleaved across all iteration depths, rather than proceeding depth-first. The fixpoint is understood as the least fixpoint on strategies ordered by prefix inclusion of their output streams.

The operators ; and $\|$ induce a depth-first exploration of the search space, as they process results sequentially and may indefinitely delay alternative branches. The operators &; and &| use diagonalization to ensure a fair exploration, guaranteeing that every branch that produces results is eventually explored.

4 Case Studies

4.1 First-Order Logic (LK)

We illustrate the expressiveness of the strategy language on a tableau calculus for classical first-order logic (LK). Inferences are classified into four families: non branching rules α, branching rules β, universal rules γ, and existential rules δ. The rules are given in Figure 1. This calculus uses a destructive closure rule $\odot$: the closing substitution is applied to all branches, potentially forbidding inferences that would have previously been possible. The closing rule is the only source of instantiation, i.e., there is no γ rule that can provide an instance $P(t)$ of some universally quantified formula.

$$\frac{P \qquad \neg P}{\odot}\ \odot \qquad \frac{\neg\neg P}{P}\ \alpha\neg\neg$$

$$\frac{P \wedge Q}{P, Q}\ \alpha\wedge \qquad \frac{\neg(P \vee Q)}{\neg P, \neg Q}\ \alpha\neg\vee$$

$$\frac{\neg(P \Rightarrow Q)}{P, \neg Q}\ \alpha\neg\Rightarrow \qquad \frac{P \vee Q}{P \mid Q}\ \beta\vee$$

$$\frac{\neg(P \wedge Q)}{\neg P \mid \neg Q}\ \beta\neg\wedge \qquad \frac{P \Rightarrow Q}{\neg P \mid Q}\ \beta\Rightarrow$$

$$\frac{\exists x.P(x)}{P(f(X_1,\ldots,X_n))}\ \delta\exists \qquad \frac{\neg\forall x.P(x)}{\neg P(f(X_1,\ldots,X_n))}\ \delta\neg\forall$$

$$\frac{\forall x.P(x)}{P(X)}\ \gamma\forall \qquad \frac{\neg\exists x.P(x)}{P(X)}\ \gamma\neg\exists$$

$$\frac{P \qquad \neg Q}{\odot, \text{ apply } \sigma \text{ to the tree if } P \text{ and } Q \text{ are unifiable and } \sigma = \mathrm{mgu}(P,Q)}\ \odot$$

Figure 1: LK Rules for First-Order Classical Logic

A common heuristic is to separate rule applications into phases: First, perform all applications of α and δ rules, then applications of β rules. Interleave this process with instantiations of universal formulas. A naive strategy would consist in saturating the branch and repeatedly applying γ: $(\alpha \,\|\, \delta \,\|\, \beta \,\|\, \gamma)^*$.

The main difficulty lies in the interaction between closure and the rest of the proof search. Failing to consider certain closure choices may delay the discovery of useful instantiations. An effective strategy must therefore ensure that all relevant closure-induced substitutions are eventually explored.

To address this issue, we separate closure from the rest of the search and combine them using fair composition:

$$(\odot^* \,\&;\, (\alpha \,\|\, \delta \,\|\, \beta \,\|\, \gamma!)^*)^*$$

The sub-strategy $\odot^*$ enumerates all finite streams of closure applications, thereby exploring different possible instantiations induced by unification. The sub-strategy $(\alpha \,\|\, \delta \,\|\, \beta \,\|\, \gamma!)^*$ performs saturation using the remaining rules. Because $\|$ is left-biased, it can be used to prioritize certain rules. The fair composition operator &; interleaves these two processes, ensuring that closure attempts are distributed

across all depths of the search. As a result, the strategy avoids committing prematurely to a particular instantiation: if a closing derivation exists, it will eventually be explored.

4.2 Modal Logic (S4)

We now consider a second case study based on propositional modal logic S4 [5]. We consider a standard classification of tableau rules for propositional S4. In addition to the usual proposition rules (α and β), we use the three modal rules described in Figure 2.

$$\frac{P}{}\ \theta^a \qquad \frac{\Box P}{\Box P; P}\ T \qquad \frac{\Box X; \neg\Box P}{\Box X; \neg P}\ S4^b$$

[a]This rule operates on a single branch. In pgeon, the tableau is represented as a forest of branches, enabling local transformation.

[b]This rule is non-local, as it requires access to the entire branch, unlike the other rules which are applied to individual formulas.

Figure 2: S4 Rules for Propositional Modal Logic

The main source of non-determinism is the rule θ, that allows arbitrary removal of formulas from the branch. θ is necessary to expose formulas to the modal rule $S4$ by removing blocking formulas so that $S4$ becomes applicable, but it may also discard formulas that are needed later to close the branch. An uncontrolled use of θ may therefore lead to dead ends in the search space.

The key observation is that θ should not be used arbitrarily, but only in order to enable applications of $S4$. We capture this idea by grouping θ and $S4$ into a combined strategy: $\theta^* \,\&;\, S4$. We combine this construction with the other rules to obtain the following strategy:

$$((\alpha \,\|\, \beta \,\|\, T)^*;(\theta^* \,\&;\, S4))^*$$

At each iteration, the strategy first saturates using α, β, and T, which are terminating using loop checking. If no such rule is applicable, it applies the combined strategy which explores all ways of applying $S4$ with the removal of some formulas. The use of fair composition ensures that different choices of formulas to discard are explored without starvation. If a successful application of $S4$ requires removing a specific subset of formulas, this configuration will eventually be reached. If closing the tableau additionally requires not discarding a specific subset of formulas, this configuration will also be reached.

5 Conclusion

This work introduced a formal strategy language for controlling proof search in tableau-based automated theorem proving, as implemented in Pgeon. By modeling strategies as functions from proof states to streams of successor states, the framework makes non-determinism explicit and enables a compositional semantics for combining search behaviors. The language provides both biased and fair combinators, with fairness achieved through interleaving constructions that prevent the starvation of branches in potentially infinite search spaces. Through case studies in first-order and modal logics, we demonstrated

how naive depth-first strategies may fail to find proofs despite the completeness of the underlying calculus, and support dynamic completeness by enabling systematic exploration of the search space. Overall, this approach offers a principled and modular foundation for specifying, analyzing, and reasoning about proof-search procedures independently of implementation details, while addressing fundamental challenges inherent to semi-decidable logics.

References

[1] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer & Carolyn L. Talcott, editors (2007): *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Lecture Notes in Computer Science 4350, Springer, doi:10.1007/978-3-540-71999-1.

[2] David Delahaye (2000): *A Tactic Language for the System Coq*. In: *LPAR*, Lecture Notes in Computer Science 1955, Springer, pp. 85–95, doi:10.1007/3-540-44404-1_7.

[3] Elena Di Lavore, Giovanni de Felice & Mario Román (2022): *Monoidal Streams for Dataflow Programming*. In: *LICS*, ACM, pp. 51:1–51:14, doi:10.1145/3531130.3533365.

[4] Conal Elliott & Paul Hudak (1997): *Functional Reactive Animation*. In: *ICFP*, ACM, pp. 263–273, doi:10.1145/258948.258973.

[5] Rajeev Goré (1999): *Tableau Methods for Modal and Temporal Logics*, pp. 297–396. Springer, doi:10.1007/978-94-017-1754-0_6.

[6] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman & Amr Sabry (2005): *Backtracking, interleaving, and terminating monad transformers: (functional pearl)*. In: *ICFP*, ACM, pp. 192–203, doi:10.1145/1086365.1086390.

[7] Romain Sidhoum, Simon Robillard & David Delahaye (2026): *Pgeon: Generating Tableau-Based Provers from Declarative Specifications of Logical Calculi*. In: International Joint Conference on Automated Reasoning (IJCAR). To appear.

[8] Uwe Waldmann, Sophie Tourret, Simon Robillard & Jasmin Blanchette (2020): *A Comprehensive Framework for Saturation Theorem Proving*. In: *IJCAR (1)*, Lecture Notes in Computer Science 12166, Springer, pp. 316–334, doi:10.1007/978-3-030-51074-9_18.