Informal pre-proceedings of 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2019)

Preface

This volume contains the papers presented at LOPSTR 19: 29th International Symposium on Logic-based Program Synthesis and Transformation held on October 7-10, 2019 in Porto.

There were 32 submissions and each submission was reviewed by at least 3 program committee members. The committee decided to accept 18 papers which are included in these informal proceedings. The program also includes 3 invited talks, two of which are contained in this volume.

Thanks to Tong Liu for his work as publicity chair, to the organizers of FM Week for their help with local (and non-local) organization and to Springer for sponsoring the best paper award. Thanks also to EasyChair for making very easy the management of the conference.

Maurizio Gabbrielli
Program chair

Program Committee

Sabine Broda	University of Porto
Manuel Carro	Technical University of Madrid (UPM) and IMDEA Software Institute
Ugo Dal Lago	Università di Bologna & INRIA Sophia Antipolis
Daniel De Schreye	Katholieke Universiteit Leuven
Santiago Escobar	Universitat Politècnica de València
Moreno Falaschi	Dipartimento di Ingegneria dell'Informazione e Scienze Matematiche, University of Siena
Maurizio Gabbrielli	University of Bologna
Arnaud Gotlieb	SIMULA Research Laboratory
Gopal Gupta	The University of Texas at Dallas
Andy King	University of Kent
Herbert Kuchen	University of Muenster
Fribourg Laurent	CNRS
Jacopo Mauro	University of Southern Denmark
Hernan Melgratti	Departamento Computación, Universidad de Buenos Aires
Maria Chiara Meo	Università degli studi G. D'Annunzio Chieti Pescara
Carlos Olarte	Universidade Federal do Rio Grande do Norte
Hirohisa Seki	Nagoya Inst. of Technology
Caterina Urban	INRIA
Herbert Wiklicky	Imperial College London

Table of Contents

Invited papers

Reversibilization in Functional and Concurrent Programming 1
German Vidal

Horn clauses and tree automata for imperative program verification 2
John P. Gallagher

Static Analysis

On fixpoint/iteration/variant induction principles for proving total correctness of programs with denotational semantics 3
Patrick Cousot

A General Framework for Static Cost Analysis of Parallel Logic Programs 23
Maximiliano Klemen, Pedro Lopez-Garcia, John P. Gallagher, Jose F. Morales and Manuel V. Hermenegildo

Incremental Analysis of Logic Programs with Assertions and Open Predicates 43
Isabel Garcia-Contreras, Jose F. Morales and Manuel V. Hermenegildo

Computing Abstract Distances in Logic Programs 63
Ignacio Casso, Jose F. Morales, Pedro López, Roberto Giacobazzi and Manuel Hermenegildo

Program synthesis

Synthesizing Imperative Code from Answer Set Programming Specifications 83
Sarat Chandra Varanasi, Elmer Salazar, Neeraj Mittal and Gopal Gupta

Logic-Based Synthesis of Fair Voting Rules Using Composable Modules 103
Karsten Diekhoff, Michael Kirsten and Jonas Krämer

Constraints and unification

Solving Proximity Constraints 123
Temur Kutsia and Cleo Pau

A Certified Functional Nominal C-Unification Algorithm 143
Mauricio Ayala-Rincon, Maribel Fernández, Gabriel Silva and Daniele Nantes-Sobrinho

Modeling and Reasoning in Event Calculus Using Goal-Directed Constraint Answer Set Programming 163
Joaquin Arias, Zhuo Chen, Manuel Carro and Gopal Gupta

Blockchain Superoptimizer 183
Julian Nagele and Maria Anna Schett

Debugging and verification

An Integrated Approach to Assertion-Based Random Testing in Prolog 203
Ignacio Casso, Jose F. Morales, Pedro Lopez-Garcia and Manuel V. Hermenegildo

Trace analysis using an Event-driven Interval Temporal Logic .....
María-Del-Mar Gallardo and Laura Panizo

The Prolog debugger and declarative programming .....
Włodek Drabent

An Optional Static Type System for Prolog .....
Isabel Wingen and Philipp Koerner

Program transformation

A Port Graph Rewriting Approach to Relational Database Modelling .....
Maribel Fernandez, Bruno Pinaud and Janos Varga

Generalization-driven semantic clone detection in CLP (Extended Abstract) .....
Wim Vanhoof and Gonzague Yernaux

Semi-Inversion of Conditional Constructor Term Rewriting Systems .....
Maja Kirkeby and Robert Glück

Natural language compositionality in Coq .....
Erkki Luuk

Reversibilization in Functional and Concurrent Programming*

Germán Vidal

MiST, VRAIN, Universitat Politècnica de València
Camino de Vera, s/n, 46022 Valencia, Spain
gvidal@dsic.upv.es

Landauer's seminal work [4] states that a computation principle can be made reversible by adding the history of the computation—a so-called Landauer embedding—to each state. Although it may seem impractical at first, there are several useful reversibilization techniques that are roughly based on this idea (e.g., [1, 8, 11, 13]).

In this talk, we first introduce a Landauer embedding for a simple (first-order) eager functional language [9, 11] and illustrate its usefulness to define an automatic *bidirectionalization* technique [12] in the context of bidirectional programming. This framework often considers two representations of some data and the functions that convert one representation into the other and vice versa (see, e.g., [3] for an overview). For instance, we may have a function called "get" that takes a *source* and returns a *view*. In turn, a function "put" takes a possibly updated view (together with the original source) and returns the corresponding, updated source. In this context, *bidirectionalization* [8] usually aims at automatically producing a function put from the corresponding function get (but the opposite approach is also possible, see, e.g., [2]).

Then, we extend the language with some primitives for message-passing concurrency and present an appropriate Landauer embedding to make its computations reversible [10, 6]. In this case, we consider *reversible debugging* as a promising application of reversible computing. Essentially, we allow the user to record an execution of a running program and, then, use the reversible semantics to reproduce some visible misbehavior inside the debugger. Here, the user can explore a computation back and forth in a *causal-consistent* way (i.e., so that an action is not undone until all the actions that depend on it have already been undone) until the source of a misbehavior is found [5, 7].

*This research has been partially supported by the EU (FEDER) and the *Spanish Ministerio de Ciencia, Innovación y Universidades*/AEI under grant TIN2016-76843-C4-1-R, by the *Generalitat Valenciana* under grant Prometeo/2019/098 (DeepTrust), and by the COST Action IC1405 on Reversible Computation - extending horizons of computing.

References

[1] Vincent Danos and Jean Krivine. Reversible communicating systems. In Philippa Gardner and Nobuko Yoshida, editors, *Proc. of the 15th International Conference on Concurrency Theory (CONCUR 2004)*, volume 3170 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 2004.

[2] Sebastian Fischer, Zhenjiang Hu, and Hugo Pacheco. The essence of bidirectional programming. *SCIENCE CHINA Information Sciences*, 58(5):1–21, 2015.

[3] Zhenjiang Hu, Andy Schürr, Perdita Stevens, and James F. Terwilliger. Bidirectional transformation "bx" (dagstuhl seminar 11031). *Dagstuhl Reports*, 1(1):42–67, 2011. Available from URL: `http://drops.dagstuhl.de/volltexte/2011/3144/`.

[4] Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5:183–191, 1961.

[5] Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. CauDEr: A Causal-Consistent Reversible Debugger for Erlang (system description). In John P. Gallagher and Martin Sulzmann, editors, *Proceedings of the 14th International Symposium on Functional and Logic Programming (FLOPS'18)*, volume 10818 of *Lecture Notes in Computer Science*, pages 247–263. Springer, 2018.

[6] Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. A theory of reversibility for Erlang. *Journal of Logical and Algebraic Methods in Programming*, 100:71–97, 2018.

[7] Ivan Lanese, Adrián Palacios, and Germán Vidal. Causal-consistent replay debugging for message passing programs. In Jorge A. Pérez and Nobuko Yoshida, editors, *Proceedings of the 39th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2019)*, volume 11535 of *Lecture Notes in Computer Science*, pages 167–184. Springer, 2019.

[8] Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In Ralf Hinze and Norman Ramsey, editors, *Proc. of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, pages 47–58. ACM, 2007.

[9] Naoki Nishida, Adrián Palacios, and Germán Vidal. Reversible term rewriting. In Delia Kesner and Brigitte Pientka, editors, *Proc. of the 1st International Conference on Formal Structures for Computation and Deduction (FSCD'16)*, volume 52 of *LIPIcs*, pages 28:1–28:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[10] Naoki Nishida, Adrián Palacios, and Germán Vidal. A reversible semantics for Erlang. In Manuel Hermenegildo and Pedro López-García, editors, *Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016)*, volume 10184 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 2017.

[11] Naoki Nishida, Adrián Palacios, and Germán Vidal. Reversible computation in term rewriting. *J. Log. Algebr. Meth. Program.*, 94:128–149, 2018.

[12] Naoki Nishida and Germán Vidal. Characterizing Compatible View Updates in Syntactic Bidirectionalization. In Michael Kirkedal Thomsen and Mathias Soeken, editors, *Proceedings of the 11th International Conference on Reversible Computation (RC 2019)*, volume 11497 of *Lecture Notes in Computer Science*, pages 67–83. Springer, 2019.

[13] Iain C.C. Phillips and Irek Ulidowski. Reversing algebraic process calculi. *J. Log. Algebr. Program.*, 73(1-2):70–96, 2007.

Horn clauses and tree automata for imperative program verification

John P. Gallagher[1,2]

[1] Roskilde University, Denmark
[2] IMDEA Software Institute, Madrid, Spain

Automatic program verification is one of the oldest challenges in computer science. The formalism of constrained Horn clauses (CHCs) has emerged in recent years as a common representation language for the semantics of imperative programming languages and the modelling of sequential, concurrent and reactive systems. This has opened up possibilities for using CHCs in verification tools [5,2,1]. Since CHCs are syntactically and semantically the same as constraint logic programs (CLP) we look at how techniques for analysis and transformation of CLP can play a role in verification of imperative programs. These techniques include partial evaluation and abstract interpretation of logic programs, and the exploitation of the connection between tree automata and Horn clauses.

The talk will summarise recent work exploiting tools, techniques and theory developed for the analysis of constraint logic programs. A set of CHCs corresponds directly to a finite tree automaton that recognises the set of derivation trees based on the CHCs. Well established properties and operations on tree automata can then be exploited to manipulate sets of CHCs. We will see how infeasible derivations can be pruned from a set of CHCs by computing the difference of tree automata. This leads to a refinement operation in a CHC verification tool [6,8]. Secondly we show how partial evaluation can be used to specialise a set of CHCs with respect to a property to be verified. In this way, verification problems can be simplified by eliminating derivations that are irrelevant to the property being verified. Furthermore, CHCs that require disjunctive invariants can often be transformed by polyvariant specialisation [4] to CHCs that can be verified with simpler invariants [7]. The same technique is applied to achieve control flow refinement of CHCs; the resulting transformed CHCs often allow termination proofs that use simple ranking functions instead of requiring more complex ranking functions such as lexicographic ranking functions, and can improve the results of automatic complexity analysis [3].

References

1. N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. Horn clause solvers for program verification. In L. D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner, and W. Schulte, editors, *Fields of Logic and Computation II*, volume 9300 of *LNCS*, pages 24–51. Springer, 2015.
2. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Program verification via iterated specialization. *Sci. Comput. Program.*, 95:149–175, 2014.

3. J. J. Doménech, J. P. Gallagher, and S. Genaim. Control-flow refinement by partial evaluation, and its application to termination and cost analysis. *CoRR*, abs/1907.12345, 2019. Accepted for TPLP.
4. J. P. Gallagher. Polyvariant program specialisation with property-based abstraction. In A. Lisitsa and A. P. Nemytykh, editors, *Proceedings Seventh International Workshop on Verification and Program Transformation (VPT-19)*, volume 299 of *EPTCS*, 2019.
5. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In J. Vitek, H. Lin, and F. Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*, pages 405–416. ACM, 2012.
6. B. Kafle and J. P. Gallagher. Horn clause verification with convex polyhedral abstraction and tree automata-based refinement. *Computer Languages, Systems & Structures*, 2015.
7. B. Kafle, J. P. Gallagher, G. Gange, P. Schachte, H. Søndergaard, and P. J. Stuckey. An iterative approach to precondition inference using constrained Horn clauses. *TPLP*, 18(3-4):553–570, 2018.
8. B. Kafle, J. P. Gallagher, and J. F. Morales. RAHFT: A tool for verifying Horn clauses using abstract interpretation and finite tree automata. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, pages 261–268, 2016.

On fixpoint/iteration/variant induction principles for proving total correctness of programs with denotational semantics

Patrick Cousot

CIMS, NYU

Abstract. We study partial and total correctness proof methods based on generalized fixpoint and iteration induction principles applied to the denotational semantics of first-order functional and iterative programs.

Keywords: Induction principles · Denotational semantics · Partial and total correctness · Verification

1 Introduction

Imperative and functional programming are very often separate worlds, even in languages like OCaml [14] which combines both style. Most programmers definitely prefer one style to the other. This reflects in semantics mostly denotational for functional and operational for imperative. This also reflects on verification, mostly Turing/Floyd/Naur/Hoare for invariance and Turing/Floyd/Manna-Pnueli variant/convergence function for termination of imperative languages while Scott proof method is preferred for functional programming.

We show that after appropriate generalization the principles underlying the verification of these programming styles boils down to the same unified verification (hence analysis) methods.

2 Basic notions in denotational semantics

The denotational semantics of first-order functions $f \in \mathcal{D} \to \mathcal{D}_\bot$ uses a complete partial order (cpo) $\langle \mathcal{D}_\bot, \sqsubseteq, \bot, \sqcup \rangle$ where $\bot$ denotes non-termination and $\mathcal{D}_\bot = \mathcal{D} \cup \{\bot\}$ is the flat domain ordered by $\bot \sqsubseteq \bot \sqsubsetneq d \sqsubseteq d$ for all $d \in \mathcal{D}$. $\sqcup$ is the least upper bound (lub) in $\mathcal{D}_\bot$. This is extended pointwise to $\langle \mathcal{D} \to \mathcal{D}_\bot, \dot{\sqsubseteq}, \dot{\bot}, \dot{\sqcup} \rangle$ by $f \mathrel{\dot{\sqsubseteq}} g$ if and only if $\forall d \in \mathcal{D} \,.\, f(d) \sqsubseteq g(d)$, $\dot{\bot} \triangleq \lambda x \bullet \bot$, and $\dot{\bigsqcup_{i\in\Delta}} f_i \triangleq \lambda x \bullet \bigsqcup_{i\in\Delta} f_i(x)$. First-order functions f are defined recursively $f(x) = F(f)x$ as least fixpoints $f = \mathsf{lfp}^{\dot{\sqsubseteq}} F$ of continuous transformers $F \in (\mathcal{D} \to \mathcal{D}_\bot) \xrightarrow{uc} (\mathcal{D} \to \mathcal{D}_\bot)$. The iterates of F from f are $F^0(f) = f$ and $F^{i+1}(f) = F(F^I(f))$. F is continuous if and only iff for every denumerable chain $f_0 \mathrel{\dot{\sqsubseteq}} f_1 \mathrel{\dot{\sqsubseteq}} \ldots \mathrel{\dot{\sqsubseteq}} f_i \mathrel{\dot{\sqsubseteq}} \ldots$, $\dot{\bigsqcup_{i\in\mathbb{N}}} F(f_i) = F(\dot{\bigsqcup_{i\in\mathbb{N}}} f_i)$. Continuity implies monotonically increasing ($f \mathrel{\dot{\sqsubseteq}} g \Rightarrow F(f) \mathrel{\dot{\sqsubseteq}} F(g)$). Since $F^0(\dot{\bot}) = \dot{\bot}$ and F is monotonically increasing, it follows that the iterates of F from $\dot{\bot}$ form an increasing chain. Then continuity guarantees that $\mathsf{lfp}^{\dot{\sqsubseteq}} F = \dot{\bigsqcup_{i\in\mathbb{N}}} F^i(\dot{\bot})$ is the limit of the iterates $F^i(\dot{\bot})$ of F from $\dot{\bot}$. By def. of $\dot{\sqsubseteq}$ and $\dot{\sqcup}$, $(\mathsf{lfp}^{\dot{\sqsubseteq}} F)x = y$ if and only if $\exists i \in \mathbb{N} \,.\, (\forall j < i \,.\, F^j(\dot{\bot})(x) = \bot) \wedge (\forall j \geqslant i \,.\, F^j(\dot{\bot})(x) = y)$.

Example 1 (while iteration). The iteration $\mathtt{W} = \mathtt{while\ (B)\ S}$ operating on a vector $x \in \mathcal{D}$ of values of variables has denotational semantics $[\![\mathtt{W}]\!] = \mathsf{lfp}^{\sqsubseteq} F_{\mathtt{W}}$ where $F_{\mathtt{W}}(f)x = (\!|\, \neg B(x) \mathbin{?} x : f(S(x)) \,|\!)$, $B \in \mathcal{D} \to \{\mathtt{tt}, \mathtt{ff}\}$ is the semantics of boolean expression $\mathtt{B}$, $S \in \mathcal{D} \to \mathcal{D}_\perp$ that of statement $\mathtt{S}$ (which may contain conditionals and inner loop), and $(\!|\, \mathtt{tt} \mathbin{?} a : b \,|\!) = a$ and $(\!|\, \mathtt{ff} \mathbin{?} a : b \,|\!) = b$ is the conditional. The iterates of $F_{\mathtt{W}}$ from $\dot{\perp}$ are

$$
\begin{aligned}
&F^0_{\mathtt{W}}(\dot{\perp})x = \perp\\
&F^1_{\mathtt{W}}(\dot{\perp})x = F_{\mathtt{W}}(F^0_{\mathtt{W}}(\dot{\perp}))x \;=\; (\!|\, \neg B(x) \mathbin{?} x : \perp \,|\!)\\
&F^2_{\mathtt{W}}(\dot{\perp})x = F_{\mathtt{W}}(F^1_{\mathtt{W}}(\dot{\perp}))x \;=\; (\!|\, \neg B(x) \mathbin{?} x : F^1_{\mathtt{W}}(\dot{\perp})(S(x)) \,|\!) \;=\; (\!|\, \neg B(x) \mathbin{?} x : (\!|\, \neg B(S(x)) \mathbin{?} S(x) : \perp \,|\!) \,|\!)\\
&\qquad\quad = (\!|\, \neg B(x) \mathbin{?} x : \perp \,|\!) \sqcup (\!|\, B(x) \wedge \neg B(x) \mathbin{?} S(x) : \perp \,|\!)
\end{aligned}
$$

...

$$
F^n_{\mathtt{W}}(\dot{\perp})x = \bigsqcup_{i=0}^{n-1} (\!|\, \bigwedge_{j=0}^{i-1} B(S^j(x)) \wedge \neg B(S^i(x)) \mathbin{?} S^i(x) : \perp \,|\!) \qquad \text{(where } S^0(x) \triangleq x \text{ and } S^{i+1}(x) \triangleq S(S^i(x))\text{)}
$$

...

$$
(\mathsf{lfp}^{\sqsubseteq} F_{\mathtt{W}})x = \bigsqcup_{I \in \mathbb{N}} F^n_{\mathtt{W}}(\dot{\perp})x \;=\; \bigsqcup_{i=0}^{n-1} (\!|\, \bigwedge_{j=0}^{i-1} B(S^j(x)) \wedge \neg B(S^i(x)) \mathbin{?} S^i(x) : \perp \,|\!)
$$

Note that in the lub, at most one condition is true, none if the iteration does not terminate. Moreover, if $(\mathsf{lfp}^{\sqsubseteq} F_{\mathtt{W}})x \neq \perp$, then, by def. $\sqcup$, $\exists j \in \mathbb{N}$. $(\mathsf{lfp}^{\sqsubseteq} F_{\mathtt{W}})x = F^j_{\mathtt{W}}(\dot{\perp})x$ and so $\neg B(FW^j(\dot{\perp})x)$ holds proving $\neg B(\mathsf{lfp}^{\sqsubseteq} F_{\mathtt{W}})$. □

3 Termination specification

The termination of function $f \in \mathcal{D} \to \mathcal{D}_\perp$ on a termination domain $T \in \wp(\mathcal{D})$ can be specified as $f \in \mathcal{P}_T$ where $\mathcal{P}_T \triangleq \{f \mid \forall x \in T \,.\, f(x) \neq \perp\}$. So $\mathcal{P}_T$ is the property of functions that terminate on domain T.

Example 2 (termination). For imperative program the termination problem is usually solved by the Turing [24]/Floyd [8]/Manna-Pnueli [16] variant/convergence function method. For first-order functions one can consider Jones size-change termination method [9,13]. □

4 Fixpoint induction principle

In case $\langle \mathcal{D}_\perp, \sqsubseteq, \perp, \sqcup, \sqcap \rangle$ is a complete lattice (*e.g.* by adding a supremum $\top$ as in Scott's original papers [22]), we can make proofs by fixpoint induction. [6, 3.4.1] and [18, (2.3)] observed that fixpoint induction directly follows from Tarski's fixpoint theorem [23].

Theorem 1 (Tarski fixpoint theorem [23]) *A monotonically increasing function $F \in L \xrightarrow{\nearrow} L$ on a complete lattice $\langle L, \sqsubseteq, \perp, \top, \sqcap, \sqcup \rangle$ has a least fixpoint $\mathsf{lfp}^{\sqsubseteq} F = \bigsqcap \{x \in L \mid F(x) \sqsubseteq x\}$.*

Fixpoint induction relies on properties of F above its least fixpoint *i.e.* the $x \in L$ such that $F(x) \sqsubseteq x$ and therefore $\mathsf{lfp}^{\sqsubseteq} F \sqsubseteq x$.

Theorem 2 (Fixpoint induction I) *Let $F \in \mathcal{L} \xrightarrow{\nearrow} \mathcal{L}$ be a monotonically increasing function on a complete lattice $\langle \mathcal{L}, \sqsubseteq, \bot, \top, \sqcap, \sqcup \rangle$ and $P \in \mathcal{L}$. We have*

$$\mathsf{lfp}^{\sqsubseteq} F \sqsubseteq P \Leftrightarrow \exists I \in \mathcal{L} \,.\, \begin{array}{ll} F(I) \sqsubseteq I & (2.\text{a}) \\ \wedge\ I \sqsubseteq P & (2.\text{b}) \end{array} \quad \square$$

I is called an *invariant* of F when $\mathsf{lfp}^{\sqsubseteq} F \sqsubseteq I$ and an *inductive invariant* when satisfying $F(I) \sqsubseteq I$.

Soundness ($\Leftarrow$) states that if a statement is proved by the proof method then that statement is true. Completeness ($\Rightarrow$) states that the proof method is always applicable to prove a true statement.

Proof (of Th. 2). By Tarski fixpoint Th. 1, $\mathsf{lfp}^{\sqsubseteq} F = \bigsqcap \{x \in L \mid F(x) \sqsubseteq x\}$.

Soundness ($\Leftarrow$): If $I \in \mathcal{L}$ satisfies $f(I) \sqsubseteq I$ then $I \in \{x \in L \mid f(x) \sqsubseteq x\}$ so by definition of the glb $\bigsqcap$, $\mathsf{lfp}^{\sqsubseteq} f = \bigsqcap \{x \in L \mid f(x) \sqsubseteq x\} \sqsubseteq I \sqsubseteq P$.

Completeness ($\Rightarrow$): If $\mathsf{lfp}^{\sqsubseteq} f \sqsubseteq P$ then take $I = \mathsf{lfp}^{\sqsubseteq} f$ then $I = f(I)$ so $f(I) \sqsubseteq I$ by reflexivity and $I \sqsubseteq P$ by hypothesis, proving $\exists I \in \mathcal{L} \,.\, F(I) \sqsubseteq I \wedge I \sqsubseteq P$. $\square$

Usually, proofs are done using logics of limited expressive power so completeness is relative to the existence of a logic formula expressing the stronger invariant $I = \mathsf{lfp}^{\sqsubseteq} f$ [3,4]. In Th. 2, we consider invariants to be sets in order to make expressivity a separate problem.

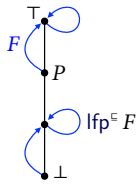
The fixpoint induction principle Th. 2 has been used to justify invariance proof methods for small-step operational semantics/transition systems, including their contrapositive, backward, *etc.* variants [5]. It can also be used with a denotational semantics.

Example 3 (Partial correctness of the factorial). Define $F_!(f) \triangleq \boldsymbol{\lambda}\, n \bullet (\!| n = 0 \mathbin{?} 1 : n \times f(n-1) |\!)$. Let us prove that $\mathsf{lfp}^{\sqsubseteq} F_! \sqsubseteq f_! \triangleq \boldsymbol{\lambda}\, n \bullet (\!| x \geqslant 0 \mathbin{?} n! : \bot |\!)$ where $n!$ is the mathematical factorial function. Applying Th. 2 with $I = P = f_!$ so that (2.b) holds, we have

$$\begin{array}{ll} & F_!(f_!)n \\ = & (\!| n = 0 \mathbin{?} 1 : n \times f_!(n-1) |\!) & \wr \text{def. } F_! \wr \\ = & (\!| n = 0 \mathbin{?} f_!(n) : f_!(n) |\!) & \wr \text{def. } f_! \wr \\ \sqsubseteq & f_!(n) & \wr \text{def. conditional and } \sqsubseteq \text{ reflexive} \wr \end{array}$$

so $F_!(f_!) \mathbin{\dot{\sqsubseteq}} f_!$ by pointwise def. of $\mathbin{\dot{\sqsubseteq}}$, proving (2.a). Obviously this is a partial correctness proof since *e.g.* $\boldsymbol{\lambda}\, n \bullet \bot \mathbin{\dot{\sqsubseteq}} f_!$. By definition of $\mathbin{\dot{\sqsubseteq}}$, we have $\forall n \in \mathbb{Z} \,.\, (\mathsf{lfp}^{\sqsubseteq} F_!)n \neq \bot \Rightarrow \mathsf{lfp}^{\sqsubseteq} F_!(n) = f_!(n)$ *i.e.* if a call $(\mathsf{lfp}^{\sqsubseteq} F_!)n$ terminates then it returns $n!$. $\square$

Notice that if $P = \mathsf{lfp}^{\sqsubseteq} f$, fixpoint induction requires to prove that $f(\mathsf{lfp}^{\sqsubseteq} f) \sqsubseteq \mathsf{lfp}^{\sqsubseteq} f$ and $\mathsf{lfp}^{\sqsubseteq} f \sqsubseteq P$. So to prove $\mathsf{lfp}^{\sqsubseteq} f \sqsubseteq P$, we have to prove $\mathsf{lfp}^{\sqsubseteq} f \sqsubseteq P$! In that case fixpoint induction cannot help. In general, we have to prove $\mathsf{lfp}^{\sqsubseteq} F \sqsubsetneq P$ but nevertheless the only inductive invariant might be $\mathsf{lfp}^{\sqsubseteq} F$, as shown below where P is not inductive.



In such cases fixpoint induction is not useful but it is possible to reason on the iterates of F, as shown in Sect. **9**.

5 Impossibility to prove termination by fixpoint induction with a denotational semantics

One can use a function $P \in \mathcal{D} \to \mathcal{D}_\bot$ to specify a termination domain $\mathsf{dom}(P) \triangleq \{x \in \mathcal{D} \mid P(x) \neq \bot\}$. However, by definition of $\dot{\sqsubseteq}$, $\mathsf{lfp}^{\dot{\sqsubseteq}}\, F \mathrel{\dot{\sqsubseteq}} P$ means that $\mathsf{lfp}^{\dot{\sqsubseteq}}\, F$ terminates less often that P that is $\mathsf{dom}(\mathsf{lfp}^{\dot{\sqsubseteq}}\, F) \subseteq \mathsf{dom}(P)$. This is not a specification of definite termination but of definite non-termination. So fixpoint induction can be used to prove non-termination but not termination. Of course $P \mathrel{\dot{\sqsubseteq}} \mathsf{lfp}^{\dot{\sqsubseteq}}\, F$ would do but this is not what fixpoint induction is intended to prove. Considering the order-dual of Th. 2 will not work either (although it would work for greatest fixpoints) since, in general, $\mathsf{gfp}^{\sqsubseteq}\, F \neq \mathsf{lfp}^{\sqsubseteq}\, F$.

6 Generalized fixpoint induction principle and total correctness proofs

The termination problem can be formulated as $\mathsf{lfp}^{\dot{\sqsubseteq}}\, F \in \{f \in \mathcal{D} \to \mathcal{D}_\bot \mid \forall x \in D \,.\, f(x) \neq \bot\}$. Fixpoint induction can be generalized to prove such properties.

Theorem 3 (Fixpoint induction II) *Let $f \in \mathcal{L} \xrightarrow{\nearrow} \mathcal{L}$ be a monotonically increasing function on a complete lattice $\langle \mathcal{L}, \sqsubseteq, \bot, \top, \sqcap, \sqcup \rangle$ and $P \in \mathcal{L}$. Let $\mathcal{P} \in \wp(\mathcal{L})$ be a property.*

$$\mathsf{lfp}^{\sqsubseteq}\, f \in \mathcal{P} \Leftrightarrow \exists \mathcal{Q} \in \wp(\mathcal{L}) \,.\, \begin{array}{ll} & \forall I \in \mathcal{L} \,.\, f(I) \sqsubseteq I \Rightarrow I \in \mathcal{Q} \quad (3.\text{a}) \\ \wedge & \forall S \subseteq \mathcal{Q} \,.\, \bigsqcap S \in \mathcal{Q} \quad (3.\text{b}) \\ \wedge & \bigsqcap \mathcal{Q} \in \mathcal{P} \quad (3.\text{c}) \end{array} \qquad \square$$

Proof (of Th. 3). By Tarski fixpoint Th. 1, $\mathsf{lfp}^{\sqsubseteq}\, F = \bigsqcap \{x \in L \mid F(x) \sqsubseteq x\}$.

Soundness ($\Leftarrow$): By (3.a), we have $S = \{I \mid F(I) \sqsubseteq I\} \subseteq \mathcal{Q}$ so $\bigsqcap S \sqsubseteq \bigsqcap \mathcal{Q}$ and, by (3.b), $\bigsqcap S \in \mathcal{Q}$ so $\bigsqcap S = \bigsqcap \mathcal{Q}$. Therefore, by (3.c), $\mathsf{lfp}^{\sqsubseteq}\, F = \bigsqcap S = \bigsqcap \mathcal{Q} \in \mathcal{P}$.

Completeness ($\Rightarrow$): Assume $\mathsf{lfp}^{\sqsubseteq}\, F \in \mathcal{P}$ so that $\bigsqcap \{x \in L \mid F(x) \sqsubseteq x\} \in \mathcal{P}$. Define $\mathcal{Q} = \{x \in L \mid F(x) \sqsubseteq x\}$, which implies (3.a). Assume that $S \subseteq \mathcal{Q}$ which can be written as $S = \{x_i \in L \mid i \in \Delta \wedge F(x_i) \sqsubseteq x_i\}$. We have $\bigsqcap S = \bigsqcap_{i \in \Delta} x_i \subseteq x_i$ by def. glb. Since F is increasing $F(\bigsqcap S) \sqsubseteq F(x_i)$ and so, by def. glb, $F(\bigsqcap S) \sqsubseteq \bigsqcap_{i \in \Delta} F(x_i) \sqsubseteq \bigsqcap_{i \in \Delta} x_i = \bigsqcap S$ since $F(x_i) \sqsubseteq x_i$. This proves (3.b). Finally, $\bigsqcap \mathcal{Q} = \mathsf{lfp}^{\sqsubseteq}\, F \in \mathcal{P}$ proving (3.c). $\square$

Remark 1. Th. 3 generalizes Th. 2 in that any proof by the method of Th. 2 implies, by soundness of Th. 2, that $\mathsf{lfp}^{\sqsubseteq}\, f \sqsubseteq P$, which is equivalent to $\mathsf{lfp}^{\sqsubseteq}\, f \in \mathcal{P}$ where $\mathcal{P} = \{x \mid x \sqsubseteq P\}$, and so, by completeness of Th. 3, can be done by the proof method of Th. 3.

Th. 3 can be used to prove termination of a recursive definition on a termination domain $T \in \wp(\mathcal{D})$ by adding a supremum to $\mathcal{D}_\bot$ to make it a complete lattice and choosing $\mathcal{P} \triangleq \{f \in \mathcal{D} \to \mathcal{D}_\bot \mid \forall x \in T \,.\, f(x) \in \mathcal{D}\}$.

Example 4 (Total correctness I). Continuing Ex. 3, we want to prove that $\mathsf{lfp}^{\sqsubseteq}\, F_! \in \mathcal{P}_{\mathbb{N}} \triangleq \{f \mid \forall n \in \mathbb{N} \,.\, f(n) \neq \bot\}$. We apply Th. 3 for the complete lattice $\langle \mathbb{Z} \to \mathbb{Z}^\top_\bot, \dot{\sqsubseteq} \rangle$ which is the pointwise extension of the complete lattice $\langle \mathbb{Z}^\top_\bot, \sqsubseteq \rangle$ with $\mathbb{Z}^\top_\bot \triangleq \mathbb{Z} \cup \{\bot, \top\}$, $\bot, \top \notin \mathbb{Z}$ and $\bot \sqsubseteq \bot \sqsubsetneq z \sqsubseteq z \sqsubsetneq \top \sqsubseteq \top$ for all $z \in \mathbb{Z}$. We let $\mathcal{Q} = \{f \in \mathbb{Z} \to \mathbb{Z}^\top_\bot \mid \forall n \in \mathbb{N} \,.\, f(n) = n!\}$.

- Assume $f \in \mathcal{Q}$, then $F_!(f) = \lambda n \bullet \llbracket n = 0 \,?\, 1 : n \times f_!(n-1) \rrbracket \in \mathcal{Q}$ since for any $n \in \mathbb{N}$ either $n = 0$ and $F_!(f)n = 1 = 0!$ or else $F_!(f)n = n \times f_!(n-1) = n \times (n-1)! = n!$, proving (a).
- If $S \subseteq \mathcal{Q}$ and $\forall n \in \mathbb{N}$ then $(\dot{\bigsqcap} S)n = \bigsqcap_{f \in S} f(n) = n!$, proving that $S \in \mathcal{Q}$ that is (b).
- In particular for $S = Q$, we have $n \in \mathbb{N} \,.\, (\dot{\bigsqcap} Q)n = n! \neq \bot$ so that $\bigsqcap \mathcal{Q} \in \mathcal{P}_{\mathbb{N}}$, proving (c). $\square$

7 Iteration induction principle

As observed by [21,17,15], iteration induction directly follows from Kleene/Scott's fixpoint theorem below (which we used in Sect. **2** with $\mathcal{L} = \mathcal{D} \to \mathcal{D}_\bot$). (Th. 4 is often attributed to Stephen Cole Kleene, after its first recursion theorem [12, p. 348].)

Theorem 4 (Kleene/Scott iterative fixpoint theorem [21]) *If $F \in \mathcal{L} \xrightarrow{uc} \mathcal{L}$ is an upper continuous function on a cpo $\langle \mathcal{L}, \sqsubseteq, \bot, \sqcup \rangle$ then F has a least fixpoint $\mathsf{lfp}^\sqsubseteq F = \bigsqcup_{n \in \mathbb{N}} F^n(\bot)$.*

Iteration induction relies on properties of F below its least fixpoint. It is usually referred to as Scott induction and formalized as

> "If $\mathcal{P} \in \wp(\mathcal{D})$ is an admissible predicate, $\bot \in \mathcal{P}$, and $\forall d \in \mathcal{P} \,.\, F(d) \in \mathcal{P}$ then $\mathsf{lfp}^\sqsubseteq F \in \mathcal{P}$". (5)

The predicate $\mathcal{P}$ is said to be admissible [15] or inclusive [20, p. 118] if and only if it holds for an increasing enumerable chain, it also holds for its limit, that is for all increasing enumerable chains $F_0 \sqsubseteq F_1 \sqsubseteq \ldots \sqsubseteq F_i \sqsubseteq \ldots$, if $\forall i \in \mathbb{N} \,.\, F_i \in \mathcal{P}$ then $\bigsqcup_{i \in \mathbb{N}} F_i \in \mathcal{P}$.

8 Impossibility to prove termination by iteration induction

To prove that $\mathsf{lfp}^{\dot\sqsubseteq} F$ terminates on $\mathcal{D}$, we can chose $\mathcal{P} \triangleq \{f \in \mathcal{D} \to \mathcal{D}_\bot \mid \forall x \in \mathcal{D} \,.\, f(x) \in \mathcal{D}\}$ but iteration induction cannot be used since (unless $\mathcal{D} = \varnothing$) $\dot\bot \notin \mathcal{P}$. So Scott's iteration induction principle is incomplete.

9 Generalized iteration induction principle

This incompleteness calls for a generalization of iteration induction where the characterization $\mathcal{Q}$ of the iterations differs from that of their limit $\mathcal{P}$.

A *maximal* $\sqsubseteq$-increasing chain in a poset $\langle S, \sqsubseteq \rangle$ is one that is not a proper subset of another $\sqsubseteq$-increasing chain in S.

Theorem 6 (Iteration induction) *Let $F \in \mathcal{L} \xrightarrow{uc} \mathcal{L}$ be an upper-continuous function on a cpo $\langle \mathcal{L}, \sqsubseteq, \bot, \sqcup \rangle$ and $\mathcal{P} \in \wp(\mathcal{L})$.*

$$\mathsf{lfp}^\sqsubseteq F \in \mathcal{P} \Leftrightarrow \exists \mathcal{Q} \in \wp(\mathcal{L}) \,.\; \begin{array}{ll} & \bot \in \mathcal{Q} \quad (6.a) \\ \wedge & \forall x \in \mathcal{Q} \,.\, F(x) \in \mathcal{Q} \quad (6.b) \\ \wedge & \textit{for any maximal } \sqsubseteq\textit{-increasing enumerable} \quad (6.c) \\ & \textit{chain } \{x_i \in \mathcal{Q} \mid i \in \mathbb{N}\} \,.\, \bigsqcup_{i \in \mathbb{N}} x_i \in \mathcal{P} \end{array} \quad \square$$

The proof below shows that the hypotheses (a), (b), and (c) are necessary only for the iterates of F. The soundness proof shows that $\mathcal{Q}$ is a valid property of the iterates of F from $\bot$ while $\mathcal{P}$ is a property of their least upper bound, that is of the fixpoint. Offering the possibility of choosing $\mathcal{Q} \neq \mathcal{P}$ is essential to solve the incompleteness problem of Scott induction (5) mentioned in the above Sect. **8**. But of course Th. 6 can be used with $\mathcal{Q} = \mathcal{P}$ so that it is generalization of Scott induction (5) and a proof that (5) is sound.

Proof (of Th. 6). Soundness ($\Leftarrow$): Let $F^{i+1}(\bot) = F(F^i(\bot))$ be the iterates of F from $F^0(\bot) = \bot$. $F^0(\bot) \in \mathcal{Q}$ by (6.a). By recurrence, $\forall i \in \mathbb{N} \,.\, F^i(\bot) \in \mathcal{Q}$ by (6.b). F is upper-continuous hence monotonically increasing so $\{F^i(\bot) \in \mathcal{Q} \mid i \in \mathbb{N}\}$ is a $\sqsubseteq$-increasing enumerable chain. It is infinite so maximal. By Th. 4 and (6.c), we conclude that $\mathsf{lfp}^{\sqsubseteq} F = \bigsqcup_{i\in\mathbb{N}} F^i(\bot) \in \mathcal{P}$.

Completeness ($\Rightarrow$): Let $F^{i+1}(\bot) = F(F^i(\bot))$ be the iterates of F from $F^0(\bot) = \bot$. Choosing $\mathcal{Q} = \{F^i(\bot) \mid i \in \mathbb{N}\}$, we have (6.a) and (6.b). By Th. 4, $\{F^i(\bot) \in \mathcal{Q} \mid i \in \mathbb{N}\}$ is a $\sqsubseteq$-increasing chain in $\mathcal{Q}$. It is enumerable and the only maximal one so $\{x_i \in \mathcal{Q} \mid i \in \mathbb{N}\} = \{F^i(\bot) \in \mathcal{Q} \mid i \in \mathbb{N}\}$. By Th. 4, $\mathsf{lfp}^{\sqsubseteq} F = \bigsqcup_{i\in\mathbb{N}} F^i(\bot)$. By hypothesis, $\mathsf{lfp}^{\sqsubseteq} F \in \mathcal{P}$, and so $\bigsqcup_{i\in\mathbb{N}} F^i(\bot) = \bigsqcup_{i\in\mathbb{N}} x_i \in \mathcal{P}$, proving (6.c). □

Example 5 (Hoare logic). Let $[\![\mathtt{W}]\!]$ be the denotational semantics of the iteration $\mathtt{W} = \mathtt{while\ (B)\ S}$ as defined in Ex. 1. Given $P, Q \in \wp(\mathcal{D})$, Hoare notation for partial correctness [10] is $\{\!|P|\!\}\ \mathtt{W}\ \{\!|Q|\!\}$ denoting $\forall x \in P \,.\, ([\![\mathtt{W}]\!]x \neq \bot) \Rightarrow ([\![\mathtt{W}]\!]x \in Q)$. Hoare partial correctness rule for the `while` iteration is

$$\frac{\{\!|I \cap B|\!\}\ \mathtt{S}\ \{\!|I|\!\}}{\{\!|I|\!\}\ \mathtt{W}\ \{\!|I \cap \neg B|\!\}} \tag{7}$$

[20, Sect. 6.6.6, p. 115] proves the soundness of the Hoare partial correctness rule for the `while` iteration based on its denotational semantics. The ad-hoc proof proceeds by induction on the semantics of the loop iterates and, assuming termination, passes to the limit. Formally, this consists in proving soundness by applying Th. 6, as follows.

– Take $\mathcal{Q} \triangleq \{f \in \mathcal{D} \to \mathcal{D}_\bot \mid \forall x \in I \,.\, f(x) \neq \bot \Rightarrow f(x) \in I\}$.

– $\bot \in \mathcal{Q}$ by def. $\mathcal{Q}$, proving (6.a).

– Assume that $f \in \mathcal{Q}$ and $F_{\mathtt{W}}(f)x = (\!|\, \neg B(x) \,?\, x : f(S(x)) \,|\!)$ as in Ex. 1. To prove (6.b), we must show that the premiss of Hoare rule (7) implies that $F_{\mathtt{W}}(f) \in \mathcal{Q}$.

If $x \in I$ and $\neg B(x)$ then obviously $x \in I$. Otherwise if $x \in I \cap B(x)$ and $F_{\mathtt{W}}(f)x \neq \bot$ then $\{\!|I \cap B|\!\}\ \mathtt{S}\ \{\!|I|\!\}$ implies $S(x) \in I$ so if $S(x) \neq \bot$ the $f(S(x)) \in I$ since $f \in \mathcal{Q}$ proving that $F_{\mathtt{W}}(f)x = f(S(x)) \in I$ that is $F_{\mathtt{W}}(f) \in \mathcal{Q}$.

– Let $\{f_i \in \mathcal{Q} \mid i \in \mathbb{N}\}$ be any maximal $\sqsubseteq$-increasing enumerable chain. Assume that $x \in I$ and $(\dot{\bigsqcup_{i\in\mathbb{N}}} f_i)x = \bigsqcup_{i\in\mathbb{N}} f_i(x) \neq \bot$. By def. lub $\sqsubseteq$, $\exists j \in \mathbb{N} \,.\, \bigsqcup_{i\in\mathbb{N}} f_i(x) = f_j(x) \neq \bot$. Since $f_j \in \mathcal{Q}$, $f_j(x) \in I$, proving $\bigsqcup_{i\in\mathbb{N}} f_i(x) \in I$ that is $\dot{\bigsqcup_{i\in\mathbb{N}}} f_i \in \mathcal{Q}$ which is (6.c).

– By Th. 6, we conclude that $[\![\mathtt{W}]\!] = \mathsf{lfp}^{\sqsubseteq} F_{\mathtt{W}} \in \mathcal{Q}$ so $\forall x \in I \,.\, [\![\mathtt{W}]\!](x) \neq \bot \Rightarrow [\![\mathtt{W}]\!](x) \in I$. Moreover, if $(\mathsf{lfp}^{\sqsubseteq} F_{\mathtt{W}})x \neq \bot$ then $\neg B(\mathsf{lfp}^{\sqsubseteq} F_{\mathtt{W}})$, as shown in Ex. 1, proving $\{\!|I|\!\}\ \mathtt{W}\ \{\!|I \wedge \neg B|\!\}$.

Obviously, this rule is incomplete since I may not be inductive (so, for completeness, [3,4] has to ensure that I is inductive and use the consequence rule). □

10 Equivalence of the generalized fixpoint and iteration induction principles

If $f \in \mathcal{L} \xrightarrow{uc} \mathcal{L}$ and $\langle \mathcal{L}, \sqsubseteq, \bot, \top, \sqcap, \sqcup \rangle$ is a complete lattice, then by soundness and completeness, a proof by the method of Th. 3 can be done by the method of Th. 6, and conversely. By Rem. 1, a proof by the method of Th. 2 can also be done by the method of Th. 6.

11 Proving total correctness by generalized iteration induction

By completeness, the termination of $\mathsf{lfp}^{\dot\sqsubseteq} F$ on a termination domain $T \in \wp(\mathcal{D})$ can always be proved by generalized iteration induction Th. 6, if $\mathsf{lfp}^{\dot\sqsubseteq} F \in \mathcal{P}_T$ does hold.

Example 6 (Total correctness II). Continuing Ex. 3, let us define $\mathcal{P}_{\mathbb{N}} \triangleq \{f \in \mathbb{N} \to \mathbb{N}_\bot \mid \forall n \in \mathbb{N} \,.\, f(n) \neq \bot\}$ and apply Th. 6 to prove that prove that $\mathsf{lfp}^{\subseteq} F_! \in \mathcal{P}_{\mathbb{N}}$ (which, together with Ex. 3, shows that $\mathsf{lfp}^{\subseteq} F_! = f_!$).

Let us define $\forall i \in \mathbb{N} \,.\, Q_i \triangleq \{f \in \mathbb{N} \to \mathbb{N}_\bot \mid \forall n \in [0, i[\,.\, f(n) \neq \bot\}$ and $Q \triangleq \bigcup_{i \in \mathbb{N}} Q_i$.

- We have $\varnothing \in \{\varnothing\} = Q_0 \subseteq Q$, proving (6.a).
- Assume that $f \in Q$ so $f \in Q_i$ for some $i \in \mathbb{N}$. We have

$$\begin{array}{ll} & F_!(f) \\ = & \lambda n \bullet (\!| n = 0 \mathbin{?} 1 : n \times f(n-1) |\!) & \wr \text{def. } F_! \text{ in Ex. 3} \wr \\ \Rightarrow & F_!(f)0 \neq \bot \wedge \forall n - 1 \in [0, i[\,.\, F_!(f)(n) \neq \bot & \wr f \in Q_i \wr \\ \Rightarrow & F_!(f) \in Q_{i+1} \subseteq Q & \wr \text{def. } Q_{i+1} \text{ and } Q \wr \end{array}$$

proving $F_!(r) \in Q$, that is (6.b).

- Let $f_0 \sqsubseteq f_1 \sqsubseteq \ldots \sqsubseteq f_i \sqsubseteq \ldots$ be a maximal increasing chain of elements of Q.

  Since the chain is assumed to be maximal, it must be strictly increasing, since, otherwise, it would be stationary at some rank i such that $f_{i-1} \sqsubsetneq f_i = f_{i+1} = \ldots$. Let i be the minimal such i. Then $f_i \in Q_j$ for some $j \in \mathbb{N}$. So there is a function $f' \in Q_{j+1}$ such that $f_i \sqsubsetneq f'$ so $\{f_0, f_1, \ldots, f_i\}$ is a proper subset of $\{f_0, f_1, \ldots, f_i, f'\}$, a contradiction with maximality.

  By def. Q, we have $f_0 \in Q_{j_0}$, $f_1 \in Q_{j_1}$, ..., $f_n \in Q_{j_n}$, $f_{n+1} \in Q_{j_{n+1}}$, ... Since the chain $f_0 \sqsubsetneq f_1 \sqsubsetneq \ldots \sqsubsetneq f_n \sqsubsetneq \ldots$ is strictly increasing, we have $j_0 < j_1 < \ldots < j_n < j_{n+1} < \ldots$ so $j_{n+1} > n + 1$. Since $f_{n+1} \in Q_{j_{n+1}}$, $f_{n+1}(n) \neq \bot$.

  To prove that $\dot{\bigsqcup}_{i \in \mathbb{N}} f_i \in \mathcal{P}_{\mathbb{N}}$, assume by contradiction, that $\exists n \in \mathbb{N} \,.\, (\bigcup_{i \in \mathbb{N}} f_i)n = \bot$ so, by def. $\dot{\sqcup}$, $\exists n \in \mathbb{N} \,.\, \forall i \in \mathbb{N} \,.\, f_i(n) = \bot$. In particular $f_{n+1}(n) = \bot$, a contradiction.

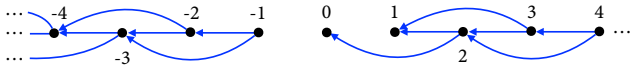
  We have proved (6.c) hence $\mathsf{lfp}^{\subseteq} F_! \in \mathcal{P}_{\mathbb{N}}$, that is $\forall n \in \mathbb{N} \,.\, (\mathsf{lfp}^{\subseteq} F_!)n \neq \bot$. □

A much simpler way of proving termination of $\mathsf{lfp}^{\subseteq} F_!$ for positive parameters is to observe that parameters strictly decreases on recursive call and remains positive which can be done only a finite number of times since $\langle \mathbb{N}, < \rangle$ is well-founded. Such termination proofs using a variant/convergence function are formalized in Th. 11. Th. 12 shows that this proof is equivalent to the above proof based on Th. 6.

12 Parameter dependency

The fact that the evaluation of $f(x) = F(f)x$ for parameter $x \in \mathcal{D}$ where $f = \mathsf{lfp}^{\sqsubseteq} F$ makes a recursive call to $f(y)$ with parameter $y \in \mathcal{D}$, written $x \overset{F}{\longmapsto} y$, is usually defined syntactically.

Example 7. Define $f(n) = F(f)n \triangleq (\!| n \in [0, 1] \mathbin{?} 0 : f(n-1) + f(n-2) |\!)$. A call of f for $n \notin [0, 1]$ will recursively call $f(n-1)$ and $f(n-2)$ in the expression $f(n-1) + f(n-2)$. So $\overset{F}{\longmapsto} = \{\langle n, n-1\rangle, \langle n, n-2\rangle \mid n \in \mathbb{Z} \setminus [0, 1]\}$:



□

Since we don't want to provide a specific syntax for defining F, we have to define the call relation $\overset{F}{\longmapsto}$ semantically. We let $f[y \leftarrow d] \in \mathcal{D} \to \mathcal{D}_\bot$ be the function f except for paramater y for which it has value $d \in \mathcal{D}_\bot$.

$$f[y \leftarrow d](y) \triangleq d$$
$$f[y \leftarrow d](z) \triangleq f(z) \quad \text{when} \quad z \neq y$$

The call relation is semantically defined as follows.

$$x \overset{F}{\longmapsto} y \triangleq \mathsf{let}\ f = \mathsf{lfp}^{\dot{\sqsubseteq}}\, F\ \mathsf{and}\ f'(z) = \llbracket\, f(z) = \bot \ ?\ 0 : f(z) \,\rrbracket\ \mathsf{in} \\ F(f'[y \leftarrow \bot])x = \bot \tag{8}$$

For simplicity, we assume F to be always well-defined so choosing $f'(z) = 0$ can never lead to a runtime error.

Example 8. Continuing Ex. 7, let $f(n) = F(f)n \triangleq \llbracket\, n \in [0,1] \ ?\ 0 : f(n-1) + f(n-2) \,\rrbracket$. The semantics is $f = \mathsf{lfp}^{\sqsubseteq} F = \boldsymbol{\lambda}\, n \bullet \llbracket\, n \geqslant 0 \ ?\ 0 : \bot \,\rrbracket$ and $f'(n) = 0$. We have

$$\begin{aligned}
& F(f'[n-1 \leftarrow \bot])n \\
&= \llbracket\, n \in [0,1] \ ?\ 0 : f'[n-1 \leftarrow \bot](n-1) + f'[n-1 \leftarrow \bot](n-2) \,\rrbracket && \text{(def. } F\text{)} \\
&= \llbracket\, n \in [0,1] \ ?\ 0 : \bot + 0 \,\rrbracket && \text{(def. } f'[n-1 \leftarrow \bot]\text{)} \\
&= \llbracket\, n \in [0,1] \ ?\ 0 : \bot \,\rrbracket && \text{(def. + assumed to be strict)}
\end{aligned}$$

Similarly $F(f'[n-2 \leftarrow \bot])n = \llbracket\, n \in [0,1] \ ?\ 0 : \bot \,\rrbracket$. In conclusion, $\overset{F}{\longmapsto} = \{\langle n, n-1\rangle, \langle n, n-1\rangle \mid n \in \mathbb{Z} \setminus [0,1]\}$. □

13 Recursive non-termination

Since we are interested in the termination of recursive functions $f(x) = F(f)x$, we exclude non-termination of the function due to causes other than recursive calls in F:

$$\forall f \in \mathcal{D} \to \mathcal{D}_\bot \,.\, \forall x \in \mathcal{D} \,.\, (F(f)x = \bot) \Rightarrow (\exists y \in \mathcal{D} \,.\, x \overset{F}{\longmapsto} y \wedge f(y) = \bot) \tag{9}$$

Example 9 (function body non-termination). Define $F(f)x =$ `if (x = 0) 1 else while (tt) ;` $f(0)$, we have $F(f)1 = \bot$ since the iteration is entered and never exited so (9) is not satisfied. This is because the non-termination is not due to the recursive calls but only to the loop body.

For $F(f)x = f(f(x))$, if $\forall x \in \mathcal{D} \,.\, f(x) \neq \bot$ is assumed to always terminate then $F(f)x = f(f(x)) \neq \bot$ does terminate, so satisfies (9). □

A recursive function definition satisfying (9) does not terminate for a given parameter if and only if it makes a recursive call that does not terminate.

Lemma 10 *Let $f = \mathsf{lfp}^{\sqsubseteq} F$ where F satisfies (9) $f(x) = \bot$ if and only if $\exists y \in \mathcal{D} \,.\, x \overset{F}{\longmapsto} y \wedge f(y) = \bot$.* □

Proof. – Let F satisfying (9) and $f = \mathsf{lfp}^{\sqsubseteq} F$. We have $f(x) = \bot$ if and only if $F(f)x = \bot$ which, by (9), implies $\exists y \in \mathcal{D} \,.\, x \overset{F}{\longmapsto} y \wedge f(y) = \bot$.

– Conversely, let $f'(z) = \llbracket\, f(z) = \bot \ ?\ 0 : f(z) \,\rrbracket$. Assume that $\exists y \in \mathcal{D} \,.\, x \overset{F}{\longmapsto} y \wedge f(y) = \bot$. By (8), $x \overset{F}{\longmapsto} y$ implies $F(f'[y \leftarrow \bot])x = \bot$. Since $\bot \sqsubseteq 0$ and $f(y) = \bot$, $f \mathrel{\dot{\sqsubseteq}} f'[y \leftarrow \bot]$ pointwise. Moreover, F is continuous hence monotonically increasing so $f(x) = F(f)x \sqsubseteq F(f'[y \leftarrow \bot]) = \bot$ so $f(x) = \bot$ since $\bot$ is the infimum. □

The hypothesis (9) is not restrictive. It simply means that, assuming that all recursive calls to f do terminate, the function body $F(f)$ must be proved to terminate. Depending on the considered programming language, this can be done *e.g.* by structural induction, using variant/convergence functions (as in Th. 11), etc. This may involve a preliminary partial correctness proof (*e.g.* using Th. 3) to restrict the values that can be taken by variables.

14 Proving termination by a variant/convergence function

Following Turing [24] and Floyd [8], most termination proofs are done using a variant/convergence function in a well-founded set which strictly decreases at each recursive call (or, equivalently, a well-founded relation). This is the case *e.g.* of the "size change principle" [9]. The variant/convergence function termination proof principle can be formulated as follows.

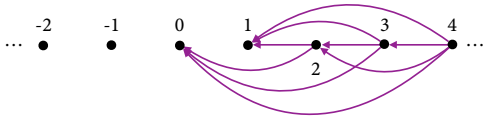
A relation $\langle D, \leqslant \rangle$ such that $\leqslant \in \wp(D \times D)$ is well-founded or Noetherian if and only if there is no infinite strictly $\gtrdot$-decreasing chain of elements of D.

Theorem 11 (variant/convergence function proof principle for termination) *Let* $F \in (\mathcal{D} \to \mathcal{D}_\bot) \xrightarrow{uc} (\mathcal{D} \to \mathcal{D}_\bot)$ *be an upper-continuous function on the cpo* $\langle \mathcal{D} \to \mathcal{D}_\bot, \dot{\sqsubseteq}, \dot{\bot}, \dot{\sqcup} \rangle$ *satisfying* (9)*,* $T \in \wp(\mathcal{D})$*, and* $\mathcal{P}_T \triangleq \{f \in \mathcal{D} \to \mathcal{D}_\bot \mid \forall x \in T \,.\, f(x) \neq \bot\}$*. Then*

$$\begin{array}{rll} \mathsf{lfp}^{\sqsubseteq} F \in \mathcal{P}_T \Leftrightarrow & \exists D \in \wp(\mathcal{D}) \,.\, T \subseteq D & (11.a) \\ \wedge & \exists \leqslant \in \wp(D \times D) \,.\, \langle D, \leqslant \rangle \textit{ is well-founded} & (11.b) \\ \wedge & \forall x \in D \,.\, \forall y \in \mathcal{D} \,.\, (x \overset{F}{\longmapsto} y) \Rightarrow (y \in D \wedge x \gtrdot y) & (11.c) \end{array}$$ □

Intuitively $\mathsf{lfp}^{\sqsubseteq} F$ must terminate since, by contradiction, an infinite call sequence would create an infinite descent along the called parameters. Completeness follows from the fact that $\gtrdot$ can be chosen as $\overset{F}{\longmapsto}$, which is always well-founded for terminating programs. This is proved formally in Cor. 13.

Example 10. Continuing Ex. 7, the well-founded relation $\lessdot$ can be chosen as follows



Termination follows from the fact that $\overset{F}{\longmapsto}$ restricted to the naturals in included in $\gtrdot$, which is well-founded. □

The variant/convergence function proof principle remains sound when this semantic dependency relation is over-approximated syntactically (but maybe not complete, as shown by $F(f)x \triangleq [\![\mathtt{tt} \;?\; x \wr f(x)]\!]$ where $x \overset{F}{\longmapsto} x$ syntactically, but not semantically because the recursive call $f(x)$ is not reachable).

15 Equivalence of the termination proof by generalized iteration induction and by variant/convergence function principle

Theorem 12 *Let $F \in \mathcal{L} \xrightarrow{uc} \mathcal{L}$ where $\mathcal{L} = \mathcal{D} \to \mathcal{D}_\perp$ which satisfies (9). There exists a termination proof by the generalized iteration induction of Th. 6 for F if and only if there exists one by the variant/convergence function principle of Th. 11.* □

The proof shows how to construct a proof by one method knowing a proof by the other method.

Proof. — Let us first show that the existence of a termination proof of $\mathsf{lfp}^{\sqsubseteq} F$ by Th. 6 implies the existence of a termination proof of $\mathsf{lfp}^{\sqsubseteq} F$ by Th. 11.

– If $\mathsf{lfp}^{\sqsubseteq} F \in \mathcal{P}_T$ has been proved by Th. 6, then, as shown by the completeness proof of this theorem, this can also be done by Th. 6 with $\forall i \in \mathbb{N} \,.\, Q^i \triangleq \{F^i\}$ where $F^0 = \lambda x \bullet \perp$ and $F^{i+1} = F(F^i)$ are the iterates of F from $\lambda x \bullet \perp$ and $Q \triangleq \bigcup_{i\in\mathbb{N}} Q^i$ so that (6.a) and (6.b) are satisfied. The only maximal $\sqsubseteq$-increasing enumerable chain $\{x_i \in Q \mid i \in \mathbb{N}\}$ is $\langle F^i, i \in \mathbb{N}\rangle$. By Th. 4, it is such that $\dot{\bigsqcup_{i\in\mathbb{N}}} F^i = \mathsf{lfp}^{\sqsubseteq} F$, By hypothesis $\mathsf{lfp}^{\sqsubseteq} F \in \mathcal{P}_T$ and so $\dot{\bigsqcup_{n\in\mathbb{N}}} F^i \in \mathcal{P}_T$, proving (6.c).

– Let us define $D^0 \triangleq \varnothing$, $D^i \triangleq \{x \mid F^{i-1}(x) = \perp \wedge F^i(x) \neq \perp\}$ for all $i > 0$, and $D \triangleq \bigcup_{i\in\mathbb{N}} D^i$. Let us prove that $T \subseteq D$.

By def. $\dot{\sqcup}$ and $\mathsf{lfp}^{\sqsubseteq} F = \dot{\bigsqcup_{i\in\mathbb{N}}} F^i$, for all $n \in \mathcal{D}$, we have $(\mathsf{lfp}^{\sqsubseteq} F)n \neq \perp \Leftrightarrow \exists i \in \mathbb{N} \,.\, F^i(n) \neq \perp \Leftrightarrow \exists i \in \mathbb{N} \,.\, n \in D^i \Leftrightarrow n \in D$. Since $\mathsf{lfp}^{\sqsubseteq} F \in \mathcal{P}_T$, for all $n \in T$, we have $(\mathsf{lfp}^{\sqsubseteq} F)n \neq \perp \Leftrightarrow n \in D$, proving $T \subseteq D$, that is (11.a).

– Let us define $x > y$ if and only if $\exists i \in \mathbb{N} \,.\, x \in D^{i+1} \wedge y \in D^i$. Let $\gtrdot$ be the irreflexive transitive closure of $>$. Let us prove that $\langle D, \lessdot\rangle$ is well-founded. By def. of $\lessdot$, for an infinite strictly decreasing chain for $\lessdot$ there exists one $x_0 > x_1 > x_2 \ldots$ for $<$, and so there would exists $x_0 \in D^{i_0}$, $x_1 \in D^{i_1}$, $x_2 \in D^{i_2}$, … with $i_0 > i_1 > i_2 > \ldots$ which is impossible since this chain on $\mathbb{N}$ cannot be infinite decreasing. This implies (11.b).

– Let x be any $x \in D$ and $y \in \mathcal{D}$ satisfies $x \overset{F}{\longmapsto} y$.

Since $x \in D$, there exists $i \in \mathbb{N}$ such that $x \in D^i$. Let i be the minimal such i. Since $x \overset{F}{\longmapsto} y$, we have $\mathsf{lfp}^{\dot{\sqsubseteq}} F(x) \neq F(\mathsf{lfp}^{\dot{\sqsubseteq}} F[y \leftarrow \perp])x$. Therefore $(\mathsf{lfp}^{\sqsubseteq} F)y \neq \perp$ since otherwise $F(\mathsf{lfp}^{\dot{\sqsubseteq}} F[y \leftarrow \perp]) = F(\mathsf{lfp}^{\dot{\sqsubseteq}} F) = \mathsf{lfp}^{\dot{\sqsubseteq}} F$, in contradiction with $F(\mathsf{lfp}^{\dot{\sqsubseteq}} F[y \leftarrow \perp])x \neq \mathsf{lfp}^{\dot{\sqsubseteq}} F(x)$. It follows that $\exists j \in \mathbb{N} \,.\, y \in D^j \subseteq D$.

- If $j < i$ then there exist $z_0 = y \in D^j$, $z_1 \in D^{j+1}$, …, $z_{i-j} = x \in D^i$ such that, by def. of D^k, $\forall k \in [0, i-j] \,.\, F^{k-1}(z_k) = \perp \wedge F^k(z_k) \neq \perp$. By def. $<$, we have $y = z_0 < z_1 < \ldots < z_{i-j} = x$ proving that $x \gtrdot y \in D$ *i.e.* (11.c).

- Else $j \geqslant i$. By def. $\dot{\sqsubseteq}$ $F_{i-1}(x) = \perp$, $F_i(x) = F_j(x) = (\mathsf{lfp}^{\dot{\sqsubseteq}} F)x \neq \perp$, $F_{i-1}(y) = F_i(y) = F_{j-1}(y) = \perp$, and $F_j(y) = (\mathsf{lfp}^{\dot{\sqsubseteq}} F)y \neq \perp$. Since $F_j(x) = F(F_{j-1}(x))$ and $F_{j-1}(y) = \perp$, we have $F_j(x) = F(F_{j-1}[y \leftarrow \perp](x))$ so $(\mathsf{lfp}^{\dot{\sqsubseteq}} F)(x) = F((\mathsf{lfp}^{\dot{\sqsubseteq}} F)[y \leftarrow \perp](x))$, a contradiction. This case is impossible and so (11.c) holds vacuously.

– Conversely, let us first show that the existence of a termination proof of $f = \mathsf{lfp}^{\sqsubseteq} F$ by Th. 11 implies the existence of a termination proof of $f = \mathsf{lfp}^{\sqsubseteq} F$ by Th. 6. So assume the existence of $D \in \wp(\mathcal{D})$ satisfying (11.a), (11.b), and (11.c).

– Define $Q_i = \{F^i(\perp)\}$ and $Q \triangleq \bigcup_{i\in\mathbb{N}} Q_i$ so (6.a) and (6.b) do hold. It remains to prove (6.c) that is $\dot{\bigsqcup_{i\in\mathbb{N}}} F^i(\perp) \in \mathcal{P}_T$. By reductio ad absurdum, assume that $\exists x_0 \in T \,.\, (\dot{\bigsqcup_{i\in\mathbb{N}}} F^i(\perp))x_0 = \perp$, that is, by (11.a) and Th. 4, $x_0 \in D$ and $(\mathsf{lfp}^{\dot{\sqsubseteq}} F)x_0 = \perp$. Assume $\exists x_j \in D \,.\, f(x_j) = \perp$ where $f = \mathsf{lfp}^{\dot{\sqsubseteq}} F$. Then, by hypothesis (9), Lem. 10 implies that $\exists x_{j+1} \,.\, x_j \overset{F}{\longmapsto} x_{j+1} \wedge f(x_{j+1}) = \perp$. In this way, we can built an

infinite sequence $x_0 \overset{F}{\longmapsto} x_1 \overset{F}{\longmapsto} x_2 \overset{F}{\longmapsto} \ldots$ such that $\forall j \in \mathbb{N} \,.\, (\mathsf{lfp}^{\sqsubseteq} F)x_j = \bot$. By recurrence and (11.c), this sequence is in D and $\gtrdot$-decreasing. This is in contradiction with the well-foundness (11.b) of $\langle D, \leqslant \rangle$. □

Corollary 13 *The variant/convergence function principle* (11) *is sound and complete for proving termination.* □

Proof. By Th. 12 and Th. 6. □

16 Extension to total correctness

The proof by [2] that Hoare logic does not exists for functional languages is based on the restriction of predicates to first-order logic with program variables only. But this is no longer the case without this restriction [7,19] and can be extended to total correctness.

Theorem 14 (The total correctness proof principle) *Let* $F \in \mathcal{D} \overset{uc}{\longrightarrow} \mathcal{D}_\bot$ *satisfying* (9) *be an upper-continuous function on the cpo* $\langle \mathcal{D}_\bot, \sqsubseteq, \bot, \sqcup \rangle$ *where* $\bot \notin \mathcal{D}$, $\mathcal{D}_\bot = \mathcal{D} \cup \{\bot\}$, $\forall x \in \mathcal{D} \,.\, \bot \sqsubseteq \bot \sqsubsetneq x \sqsubseteq x$, $P \in \wp(\mathcal{D})$, $Q \in \wp(\mathcal{D} \times \mathcal{D})$, *and* $\mathcal{P}_{P,Q} \triangleq \{f \in \mathcal{D} \to \mathcal{D}_\bot \mid \forall x \in P \,.\, \langle x, f(x) \rangle \in Q\}$. *Then*

$$
\begin{array}{lll}
\mathsf{lfp}^{\dot\sqsubseteq} F \in \mathcal{P}_{P,Q} \Leftrightarrow \exists D \in \wp(\mathcal{D}) \,.\, \exists I \in \wp(\mathcal{D} \times \mathcal{D}) \,. & & \\
\qquad & P \subseteq D & \text{(14.a)} \\
\qquad \wedge & \{\langle x, y \rangle \in I \mid x \in P\} \subseteq Q & \text{(14.b)} \\
\qquad \wedge & \exists \leqslant \in \wp(\mathcal{D} \times \mathcal{D}) \,.\, \langle D, \leqslant \rangle \textit{ is well-founded} & \text{(14.c)} \\
\qquad \wedge & \forall x, y \in \mathcal{D} \,.\, (x \in D \wedge x \overset{F}{\longmapsto} y) \Rightarrow (y \in D \wedge x \gtrdot y) & \text{(14.d)} \\
\qquad \wedge & \mathsf{let}\ \ \mathcal{P}_{D,I} \triangleq \{f \in \mathcal{D} \to \mathcal{D}_\bot \mid \forall x \in D \,.\, (f(x) \neq \bot \Rightarrow \langle x, f(x) \rangle \in I)\}\ \ \mathsf{in} & \text{(14.e)} \\
\qquad\qquad & \forall f \in \mathcal{P}_{D,I} \,.\, F(f) \in \mathcal{P}_{D,I} &
\end{array}
$$

□

Example 11 (Total correctness of the factorial). Define $F_!(f) \triangleq \lambda n \bullet [\![n = 0 \,?\, 1 : n \times f(n-1)]\!]$, $P = \mathbb{N}$, $Q = \{\langle n, n! \rangle \mid n \in \mathbb{N}\}$ So that $\mathsf{lfp}^{\dot\sqsubseteq} F \in \mathcal{P}_{P,Q}$ expresses that $F_!(f)n$ terminates for $n \in \mathbb{N}$ and returns the factorial $n!$ of n. Take $D = P$ and $I = Q$ so that (14.a), (14.b), (14.c) are trivially satisfied since $D = \mathbb{N}$ and $\langle \mathbb{N}, \leqslant \rangle$ is well-founded. If $n \in D$ and $n \overset{F}{\longmapsto} y$ then $n \neq 0$ and $y = n - 1$ so $n > n - 1 \in D$, proving (14.d). If $f \in \mathcal{P}_{D,I}$ and $n \in D = \mathbb{N}$ then $f(n) = n!$. So $F_!(f)n = n!$ since either $n = 0$ and $F_!(f)0 = 1 = 0!$ or $n > 0$ so $n - 1 \in \mathbb{N}$, $f(n-1) = (n-1)!$ so $F_!(f)n = n \times (n-1)! = n!$. Therefore $F_!(f) \in \mathcal{P}_{D,I}$, proving (14.e). By Th. 14, $\mathsf{lfp}^{\dot\sqsubseteq} F_! \in \mathcal{P}_{P,Q}$. □

Proof (of Th. 14). Soundness (⇐): Take $T = D$ in Th. 11. Then (14.a) implies (11.a), (14.c) implies (11.b), and (14.d) implies (11.c). By Th. 11, this implies $\forall x \in P \,.\, (\mathsf{lfp}^{\sqsubseteq} F)x \neq \bot$.

Take $\mathcal{L} = \mathcal{D} \to \mathcal{D}_\bot$, $\mathcal{P} = \mathcal{Q} = \mathcal{P}_{D,I}$. By def. $\mathcal{P}_{D,I}$, $\dot\bot \in \mathcal{Q}$ proving (6.a). By (14.d), $\forall f \in \mathcal{Q} \,.\, F(f) \in \mathcal{Q}$, proving (6.b). Let $\{f_i \in \mathcal{Q} \mid i \in \mathbb{N}\}$ be any maximal $\sqsubseteq$-increasing enumerable chain of elements of $\mathcal{Q}$. If $x \in \mathcal{D}$ and $(\dot\bigsqcup_{i \in \mathbb{N}} f_i)x \neq \bot$, then $(\dot\bigsqcup_{i \in \mathbb{N}} f_i)x = d \in \mathcal{D}$ so, by def. lub $\dot\bigsqcup$, there exists $j \in \mathbb{N} \,.\, (\dot\bigsqcup_{i \in \mathbb{N}} f_i)x = f_j(x) = d$. But $f_j \in \mathcal{Q} = \mathcal{P}_{D,I}$ so $\langle x, d \rangle = \langle x, (\dot\bigsqcup_{i \in \mathbb{N}} f_i)x \rangle \in I$. If follows that $(\dot\bigsqcup_{i \in \mathbb{N}} f_i) \in \mathcal{P}_{D,I} = \mathcal{P}$. By Th. 6, $\mathsf{lfp}^{\sqsubseteq} F \in \mathcal{P}_{D,I}$.

Since $\mathsf{lfp}^{\sqsubseteq} F \in \mathcal{P}_{D,I}$ and $\forall x \in P \,.\, (\mathsf{lfp}^{\sqsubseteq} F)x \neq \bot$, we conclude that $\mathsf{lfp}^{\sqsubseteq} F \in \mathcal{P}_{P,Q}$.

Completeness ($\Rightarrow$): Assume that $\mathsf{lfp}^{\sqsubseteq} F \in \mathcal{P}_{P,Q}$ so $\forall x \in P \,.\, (\mathsf{lfp}^{\sqsubseteq} F)x \neq \bot$. Applying Th. 11 with $T = P$, there exists $D \in \wp(\mathcal{D})$ satisfying (11.a), (11.b), and (11.c). Applying Th. 6 with $\mathcal{L} = \mathcal{D} \to \mathcal{D}_\bot$, there exists $\mathcal{Q} \in \wp(\mathcal{D} \to \mathcal{D}_\bot)$ satisfying (6.a), (6.b), (6.c). Moreover, the completeness proof of Th. 6 shows that $\mathcal{Q} = \{F^i(\bot) \mid i \in \mathbb{N}\}$.

Choose $I \triangleq \{\langle x,\, f(x)\rangle \in \mathcal{D} \times \mathcal{D} \mid f \in \mathcal{Q} \vee f = \bigsqcup \mathcal{Q}\}$, where, as shown in the completeness proof of Th. 6, $\bigsqcup \mathcal{Q}$ is well-defined and equal to $\mathsf{lfp}^{\sqsubseteq} F$.

- We have $P = T \subseteq D$ by (11.a), proving (14.a);
- If $\langle x, y\rangle \in I$ then $y \neq \bot$ and $y = f(x)$ where $f \in \mathcal{Q}$ or $f = \bigsqcup \mathcal{Q}$. In both cases, by $\mathcal{Q} = \{F^i(\bot) \mid i \in \mathbb{N}\}$ and $f(x) \neq \bot$, we have $y = \mathsf{lfp}^{\sqsubseteq} F)x$ so, by hypothesis $\mathsf{lfp}^{\sqsubseteq} F \in \mathcal{P}_{P,Q}$, if $x \in P$, then $\langle x,\, y\rangle \in Q$, proving (14.b);
- (11.b) is exactly (14.c);
- (11.c) is exactly (14.d);
- Assume that $f \in \mathcal{P}_{D,I} = \{f \in \mathcal{D} \to \mathcal{D}_\bot \mid \forall x \in D \,.\, (f(x) \neq \bot \Rightarrow \langle x,\, f(x)\rangle \in I)\}$. Then either $f \in \mathcal{Q}$ so, by (6.b), $F(f) \in \mathcal{Q}$ and therefore $F(f) \in \mathcal{P}_{D,I}$ or $f = \bigsqcup \mathcal{Q} = \mathsf{lfp}^{\sqsubseteq} F$ so $F(f) = f \in \mathcal{P}_{D,I}$, proving (14.e). □

17 Application to the `while` iteration

Manna and Pnueli [16] generalized Hoare partial correctness rule for total correctness $(\!|P|\!)\ \mathtt{w}\ (\!|Q|\!)$ denoting $\forall x \in P \,.\, [\![\mathtt{w}]\!]x \in Q$ which is traditionally decomposed in partial correctness $\{\!|P|\!\}\ \mathtt{w}\ \{\!|Q|\!\}$ and termination $\forall x \in P \,.\, [\![\mathtt{w}]\!]x \neq \bot$. They rely on the idea of relating the initial and final values of variables in Q, writing $P(x)$ for $x \in P \in \wp(\mathcal{D})$ and $Q(x, x')$ for $\langle x,\, x'\rangle \in Q \in \wp(\mathcal{D} \times \mathcal{D})$, so that the rule are written in the form $(\!|P(x)|\!)\ \mathtt{w}\ (\!|Q(x, x')|\!)$ where x is the value before execution and x' that upon termination.

$(\!|P(x)|\!)\ \mathtt{w}\ (\!|Q(x, x')|\!)$ is equivalent to $\mathsf{lfp}^{\dot{\sqsubseteq}} F_{\mathtt{w}} \in \mathcal{P}_{P,Q}$. So, by the soundness and completeness of Th. 14, this is equivalent to the existence of $D \in \wp(\mathcal{D})$ and $I \in \wp(\mathcal{D} \times \mathcal{D})$ satisfying the conditions.

$$
\begin{array}{ll}
 & P(x) \Rightarrow D(x) \qquad (15.\text{a})\\
\wedge & P(x) \wedge I(x, y) \Rightarrow Q(x, y) \qquad (15.\text{b})\\
\wedge & \exists \leqslant \in \wp(\mathcal{D} \times \mathcal{D}) \,.\, \langle D, \leqslant\rangle \text{ is well-founded} \qquad (15.\text{c})\\
\wedge & \forall x \in D \,.\, S(x) \in D \wedge x \gtrdot S(x) \qquad (15.\text{d})\\
\wedge & \forall x \in D, x'' \in \mathcal{D} \,.\, (B(x) \wedge I(S(x), x'') \Rightarrow I(x, x'') \qquad (15.\text{e})\\
\wedge & \forall x \in D \,.\, \neg B(x) \Rightarrow I(x, x) \qquad (15.\text{e}')
\end{array}
$$

since for (14.d), $x \overset{F}{\longmapsto} y$ if and only if $y = S(x)$, by def. $F_{\mathtt{w}}(f)x = [\![\, \neg B(x) \;?\; x : f(S(x)) \,]\!]$ and for (14.e/e′), given $\mathcal{P}_{D,I} \triangleq \{f \in \mathcal{D} \to \mathcal{D}_\bot \mid \forall x \in D \,.\, f(x) \neq \bot \Rightarrow \langle x,\, f(x)\rangle \in I\}$, we have

$$
\begin{array}{ll}
 & \forall f \in \mathcal{P}_{D,I} \,.\, F_{\mathtt{w}}(f) \in \mathcal{P}_{D,I}\\
\Leftrightarrow & \forall f \in \mathcal{D} \to \mathcal{D}_\bot \,.\, (\forall x \in D \,.\, (f(x) \neq \bot) \Rightarrow (\langle x,\, f(x)\rangle \in I)) \Rightarrow (\forall x \in D \,.\, [\![\, \neg B(x) \;?\; \langle x,\, x\rangle \in I : (f(S(x)) \neq \bot \Rightarrow (\langle x,\, f(S(x))\rangle \in I) \,]\!])\\
 & \qquad\qquad \text{〈def. } \mathcal{P}_{D,I} \text{ and } F_{\mathtt{w}}\text{〉}\\
\Leftrightarrow & \forall f \in \mathcal{D} \to \mathcal{D} \,.\, (\forall x \in D \,.\, (\langle x,\, f(x)\rangle \in I)) \Rightarrow (\forall x \in D \,.\, [\![\, \neg B(x) \;?\; \langle x,\, x\rangle \in I : (\langle x,\, f(S(x))\rangle \in I) \,]\!])\\
 & \text{〈since the } \bot \text{ case is excluded〉}\\
\Leftrightarrow & (\forall x \in D \,.\, \neg B(x) \Rightarrow \langle x,\, x\rangle \in I) \wedge (\forall x \in D \,.\, (B(x) \wedge \langle S(x),\, x''\rangle \in I) \Rightarrow \langle x,\, x'\rangle \in I)\\
 & \qquad\qquad \text{〈since } f \text{ is defined by } I \text{ and letting } x' = f(S(x))\text{〉} \quad \square
\end{array}
$$

Rewriting (15) in Manna-Pnueli style, we get the sound and complete rule (which incorporate the consequence rule):

$$P(x) \Rightarrow D(x), \quad P(x) \wedge I(x,y) \Rightarrow Q(x,y), \tag{16.a/b}$$
$$\exists \preccurlyeq \in \wp(\mathcal{D} \times \mathcal{D}) \,.\, \langle D, \preccurlyeq \rangle \text{ is well-founded}, \tag{16.c}$$
$$(\!| D(x) |\!)\ \mathtt{S}\ (\!| D(x') \wedge x \succ x' |\!), \tag{16.d}$$
$$(\!| D(x) \wedge B(x) |\!)\ \mathtt{S}\ (\!| I(x,x') \wedge \forall x'' \,.\, I(x',x'') \Rightarrow I(x,x'') |\!), \tag{16.e}$$
$$\forall x \,.\, D(x) \wedge \neg B(x) \Rightarrow I(x,x) \tag{16.e$'$}$$
$$\overline{\qquad\qquad}$$
$$(\!| P(x) |\!)\ \mathtt{W}\ (\!| Q(x,x') \wedge \neg B(x') |\!)$$

(The conjunction with the post-condition $\neg B(x')$ is explained in Ex. 5).

The original Manna and Pnueli rule [16, Sect. 8.3] is slightly different, as follows.

$$(\!| P(x) \wedge B(x) |\!)\ \mathtt{S}\ (\!| P(x') \wedge Q(x,x') \wedge x \succ x' |\!), \tag{17.i}$$
$$\forall x, x', x'' \,.\, Q(x,x') \wedge Q(x',x'') \Rightarrow Q(x,x''), \tag{17.ii}$$
$$\forall x \,.\, P(x) \wedge \neg B(x) \Rightarrow Q(x,x) \tag{17.iii}$$
$$\overline{\qquad\qquad}$$
$$(\!| P(x) |\!)\ \mathtt{W}\ (\!| Q(x,x') \wedge \neg B(x') |\!)$$

As in [10], the proof rules are postulated so no soundness or completeness proof is given. A soundness and completeness proof is provided in [1] based on Scott induction (using transfinite iterates in absence of continuity due to the consideration of unbounded nondeterminism).

Assume the hypotheses of Manna-Pnueli inference rule (17), and define (with informal notations)

- $P' = D \triangleq P(x)$;
- $Q' = I(x,y) \triangleq Q(x,y)$;

so that

- (15.a) holds trivially by reflexivity;
- (15.b) holds trivially since $P(x) \wedge I(x,y) \Rightarrow Q'(x,y)$. Moreover, the conjunction with the term $B(x')$ follows from the semantics of the while iteration, as shown in Ex. 1;
- (15.c) is a side condition in Manna-Pnueli rule (there should be a convergence function u into a well-founded set $\langle \mathcal{W}, \preceq \rangle$ with $x \preccurlyeq y$ if and only if $u(x) \preceq u(y)$);
- Since $(\!| P'(x) |\!)\ \mathtt{S}\ (\!| Q'(x,x') |\!)$ denotes $\forall x \,.\, P'(x) \Rightarrow (Q'(x,S(x)) \wedge S(x) \neq \bot)$ where $S = [\![\mathtt{S}]\!]$ is the denotational semantics of $\mathtt{S}$, (17.i) implies both
  - $\forall x \,.\, (P(x) \wedge B(x)) \Rightarrow (P(S(x)) \wedge x \succ S(x))$, which is (15.d);
  - and
    $\forall x \,.\, (P(x) \wedge B(x)) \Rightarrow Q(x,S(x))$
    which together with (17.ii)
    $\forall x, x'' \,.\, (Q(x,S(x)) \wedge Q(S(x),x'') \Rightarrow Q(x,x'')$
    yields
    $\forall x \,.\, (P(x) \wedge B(x) \wedge Q(S(x),x'')) \Rightarrow Q(x,x'')$, which is (15.e);
- (15.e$'$) is exactly (17.ii).

By Th. 14, Moreover, if $(\mathsf{lfp}^{\sqsubseteq} F_{\mathtt{W}})x \neq \bot$ then $\neg B(\mathsf{lfp}^{\sqsubseteq} F_{\mathtt{W}})$, as shown in Ex. 1, we conclude that Manna-Pnueli rule is sound.

Obviously Manna-Pnueli rule (17) is not complete (since $Q(x,x')$ might not be inductive), but it can be applied to the strongest invariant and the conclusion derived by the consequence rule.

18 Conclusion

Traditional Park/fixpoint induction can prove invariance/partial correctness but not termination (at least without introducing auxiliary variables such as bounded loop counters [11]). The traditional Scott/iteration induction cannot prove termination either. We generalized both of these induction principles to prove termination/total correctness. Park/fixpoint induction is useful to reason on post-fixpoints, above the least fixpoint. Scott/iteration induction is useful to reason on iterates, below the least fixpoint. For termination they are equivalent to the Turing/Floyd termination proof method using variant/convergence functions. This applies both to (first-order) functional and imperative programming. In particular, the Manna-Pnueli method for proving the total correctness of `while` loops is equivalent to Scott induction for the denotational semantics of these loops, hence answering the question `[TYPES] Variants and [Park or Scott] fixpoint Induction` by Peter O'Hearn dated `Fri, 14 Jun 2019 15:23:24 +0000`.

References

1. Apt, K.R., Plotkin, G.D.: Countable nondeterminism and random assignment. J. ACM **33**(4), 724–767 (1986)
2. Clarke, E.M.: Programming language constructs for which it is impossible to obtain good hoare axiom systems. J. ACM **26**(1), 129–147 (1979)
3. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM J. Comput. **7**(1), 70–90 (1978)
4. Cook, S.A.: Corrigendum: Soundness and completeness of an axiom system for program verification. SIAM J. Comput. **10**(3), 612 (1981)
5. Cousot, P., Cousot, R.: Induction principles for proving invariance properties of programs. In: Néel, D. (ed.) Tools & Notions for Program Construction: an Advanced Course. pp. 75–119. Cambridge University Press, Cambridge, UK (Aug 1982)
6. Cousot, P.: Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French). Thèse d'État ès sciences mathématiques, Université de Grenoble Alpes, Grenoble, France (March 21, 1978)
7. Damm, W., Josko, B.: A sound and relatively * complete hoare-logic for a language with higher type procedures. Acta Inf. **20**, 59–101 (1983)
8. Floyd, R.W.: Assigning meaning to programs. In: Schwartz, J. (ed.) Proc. Symp. in Applied Math., vol. 19, pp. 19–32. Amer. Math. Soc. (1967)
9. Heizmann, M., Jones, N.D., Podelski, A.: Size-change termination and transition invariants. In: SAS. Lecture Notes in Computer Science, vol. 6337, pp. 22–50. Springer (2010)
10. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969), http://doi.acm.org/10.1145/363235.363259
11. Katz, S., Manna, Z.: A closer look at termination. Acta Inf. **5**, 333–352 (1975)
12. Kleene, S.C.: Introduction to Meta-Mathematics. Elsevier North-Holland Pub. Co. (1952)
13. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL. pp. 81–92. ACM (2001)
14. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml system, release 4.08, Documentation and user's manual (Feb 2019), http://caml.inria.fr/pub/docs/manual-ocaml/, copyright © 2013 Institut National de Recherche en Informatique et en Automatique
15. Manna, Z., Ness, S., Vuillemin, J.: Inductive methods for proving properties of programs. Commun. ACM **16**(8), 491–502 (1973)
16. Manna, Z., Pnueli, A.: Axiomatic approach to total correctness of programs. Acta Inf. **3**, 243–263 (1974)

17. Manna, Z., Vuillemin, J.: Fix point approach to the theory of computation. Commun. ACM **15**(7), 528–536 (1972)
18. Park, D.M.R.: On the semantics of fair parallelism. In: Abstract Software Specifications. Lecture Notes in Computer Science, vol. 86, pp. 504–526. Springer (1979)
19. Régis-Gianas, Y., Pottier, F.: A hoare logic for call-by-value functional programs. In: MPC. Lecture Notes in Computer Science, vol. 5133, pp. 305–335. Springer (2008)
20. Schmidt, D.W.: Denotational Semantics: A Methodology for Language Development. William C. Brown Publishers, Dubuque, IA, USA (Jun 1988), http://people.cs.ksu.edu/~schmidt/text/DenSem-full-book.pdf
21. Scott, D.S.: Outline of a mathematical theory of computation. In: Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems. pp. 169–176. Princeton University (Mar 1970)
22. Scott, D.S.: The lattice of flow diagrams. In: Symposium on Semantics of Algorithmic Languages, Lecture Notes in Mathematics, vol. 188, pp. 311–366. Springer (1971)
23. Tarski, A.: A lattice theoretical fixpoint theorem and its applications. Pacific J. of Math. **5**, 285–310 (1955)
24. Turing, A.: Checking a large routine. In: Report of a Conference on High Speed Automatic Calculating Machines, University of Cambridge Mathematical Laboratory, Cambridge, England. pp. 67–69 (1949), http://www.turingarchive.org/browse.php/b/8

A General Framework for Static Cost Analysis of Parallel Logic Programs*

M. Klemen[1,2], P. López-García[1,3], J.P. Gallagher[1,4], J.F. Morales[1], and M.V. Hermenegildo[1,2]

[1] IMDEA Software Institute, Spain
[2] ETSI Informáticos, Universidad Politécnica de Madrid (UPM), Spain
[3] Spanish Council for Scientific Research (CSIC), Spain
[4] Roskilde University, Denmark
{maximiliano.klemen,pedro.lopez,john.gallagher,josef.morales,manuel.hermenegildo}@imdea.org

Abstract. The estimation and control of *resource usage* is now an important challenge in an increasing number of computing systems. In particular, requirements on timing and energy arise in a wide variety of applications such as internet of things, cloud computing, health, transportation, and robots. At the same time, parallel computing, with (heterogeneous) multicore platforms in particular, has become the dominant paradigm in computer architecture. Predicting resource usage on such platforms poses a difficult challenge. Most work on static resource analysis has focused on sequential programs, and relatively little progress has been made on the analysis of parallel programs, or more specifically on parallel logic programs. We propose a novel, general, and flexible framework for setting up cost equations/relations which can be instantiated for performing resource usage analysis of parallel logic programs for a wide range of resources, platforms and execution models. The analysis estimates both lower and upper bounds on the resource usage of a parallel program (without executing it) as functions on input data sizes. In addition, it also infers other meaningful information to better exploit and assess the potential and actual parallelism of a system. We develop a method for solving cost relations involving the *max* function that arise in the analysis of parallel programs. Finally, we instantiate our general framework for the analysis of logic programs with Independent And-Parallelism, report on an implementation within the CiaoPP system, and provide some experimental results. To our knowledge, this is the first approach to the cost analysis of *parallel logic programs*.

Keywords: Resource Usage Analysis, Parallelism, Static Analysis, Complexity Analysis, (Constraint) Logic Programming, Prolog.

1 Introduction

Estimating in advance the resource usage of computations is useful for a number of applications; examples include granularity control in parallel/distributed systems, automatic program optimization, verification of resource-related specifications and

* Research partially funded by Spanish MINECO TIN2015-67522-C3-1-R *TRACES* project, and the Madrid P2018/TCS-4339 *BLOQUES-CM* program. We are also grateful to the anonymous reviewers for their useful comments.

detection of performance bugs, as well as helping developers make resource-related design decisions. Besides *time* and *energy*, we assume a broad concept of resources as numerical properties of the execution of a program, including the number of *execution steps*, the number of *calls* to a procedure, the number of *network accesses*, number of *transactions* in a database, and other user-definable resources. The goal of automatic static analysis is to estimate such properties without running the program with concrete data, as a function of input data sizes and possibly other (environmental) parameters.

Due to the heat generation barrier in traditional sequential architectures, parallel computing, with (heterogeneous) multi-core processors in particular, has become the dominant paradigm in current computer architecture. Predicting resource usage on such platforms poses important challenges. Most work on static resource analysis has focused on sequential programs, and relatively little progress has been made on the analysis of parallel programs, or on parallel logic programs in particular. The significant body of work on static analysis of sequential logic programs has already been applied to the analysis of other programming paradigms, including imperative programs. This is achieved via a transformation into *Horn clauses* [23]. In this paper we concentrate on the analysis of parallel Horn clause programs, which could be the result of such a translation from a parallel imperative program or be themselves the source program. Our starting point is the well-developed technique of setting up recurrence relations representing resource usage functions parameterized by input data sizes [28, 25, 10, 8, 11, 24, 2, 26], which are then solved to obtain (exact or safely approximated) closed forms of such functions (i.e., functions that provide upper or lower bounds on resource usage). We build on this and propose a novel, general, and flexible framework for setting up cost equations/relations which can be instantiated for performing static resource usage analysis of parallel logic programs for a wide range of resources, platforms and execution models. Such an analysis estimates both lower and upper bounds on the resource usage of a parallel program as functions on input data sizes. We have instantiated the framework for dealing with Independent And-Parallelism (IAP) [16, 12], which refers to the parallel execution of conjuncts in a goal. However, the results can be applied to other languages and types of parallelism, by performing suitable transformations into Horn clauses.

The main contributions of this paper can be summarized as follows:

- We have extended a general static analysis framework for the analysis of sequential Horn clause programs [24, 26], to deal with parallel programs.
- Our extensions and generalizations support a wide range of resources, platforms and parallel/distributed execution models, and allow the inference of both lower and upper bounds on resource usage. This is the first approach, to our knowledge, to the cost analysis of *parallel logic programs* that can deal with features such as backtracking, multiple solutions (i.e., non-determinism), and failure.
- We have instantiated the developed framework to infer useful information for assessing and exploiting the potential and actual parallelism of a system.
- We have developed a method for finding closed-form functions of cost relations involving the *max* function that arise in the analysis of parallel programs.
- We have developed a prototype implementation that instantiates the framework for the analysis of logic programs with Independent And-Parallelism within the CiaoPP system [15, 24, 26], and provided some experimental results.

2 Overview of the Approach

Prior to explaining our approach, we provide some preliminary concepts. Independent And-Parallelism arises between two goals when their corresponding executions do not affect each other. For pure goals (i.e., without side effects) a sufficient condition for the correctness of IAP is the absence of variable sharing at run-time among such goals. IAP has traditionally been expressed using the `&/2` meta-predicate as the constructor to represent the parallel execution of goals. In this way, the conjunction of goals (i.e., literals) `p & q` in the body of a clause will trigger the execution of goals `p` and `q` in parallel, finishing when both executions finish.

Given a program $\mathcal{P}$ and a predicate $\mathtt{p} \in \mathcal{P}$ of arity k and a set Π of k-tuples of calling data to $\mathtt{p}$, we refer to the *(standard) cost* of a call $\mathtt{p}(\bar{\mathtt{e}})$ (i.e., a call to $\mathtt{p}$ with actual data $\bar{\mathtt{e}} \in \Pi$), as the resource usage (under a given cost metric) of the complete execution of $\mathtt{p}(\bar{\mathtt{e}})$. The *standard cost* is formalized as a function $\mathcal{C}_\mathtt{p} : \Pi \to \mathcal{R}_\infty$, where $\mathcal{R}_\infty$ is the set of real numbers augmented with the special symbol ∞ (which is used to represent non-termination). We extend the function $\mathcal{C}_\mathtt{p}$ to the powerset of Π, i.e., $\hat{\mathcal{C}}_\mathtt{p} : 2^\Pi \to 2^{\mathcal{R}_\infty}$, where $\hat{\mathcal{C}}_\mathtt{p}(E) = \{\mathcal{C}_\mathtt{p}(\bar{\mathtt{e}}) \mid \bar{\mathtt{e}} \in E\}$. Our goal is to abstract (safely approximate, as accurately as possible) $\hat{\mathcal{C}}_\mathtt{p}$ (note that $\mathcal{C}_\mathtt{p}(\bar{\mathtt{e}}) = \hat{\mathcal{C}}_\mathtt{p}(\{\bar{\mathtt{e}}\})$). Intuitively, this abstraction is the composition of two abstractions: a size abstraction and a cost abstraction. The goal of the analysis is to infer two functions $\hat{\mathcal{C}}^{\downarrow}_\mathtt{p}$ and $\hat{\mathcal{C}}^{\uparrow}_\mathtt{p} : \mathcal{N}^m_\top \to \mathcal{R}_\infty$ that give lower and upper bounds respectively on the cost function $\hat{\mathcal{C}}_\mathtt{p}$, where $\mathcal{N}^m_\top$ is the set of m-tuples whose elements are natural numbers or the special symbol $\top$, meaning that the size of a given term under a given size metric is *undefined*. Such bounds are given as a function of tuples of data sizes (representing the concrete tuples of data of the concrete function $\hat{\mathcal{C}}_\mathtt{p}$). Typical size metrics are the actual value of a number, the length of a list, the size (number of constant and function symbols) of a term, etc. [24, 26].

We now enumerate different metrics used to evaluate the performance of parallel versions of a logic program, compared against its corresponding sequential version. When possible, we define these metrics parameterized with respect to the resource (e.g., number of *resolution steps*, *execution time*, or *energy consumption*) in which the cost is expressed.

- **Sequential cost (*Work*)**: It is the standard cost of executing a program, assuming no parallelism.
- **Parallel cost (*Depth*)**: It is the cost of executing a program in parallel, considering an unbounded number of processors.
- **Maximum number of processes running in parallel ($Proc_{max}(P)$)**: The maximum number of processes that can run simultaneously in a program. This is useful, for example, to determine what is the minimum number of processors that are required to actually run all the processes in parallel.

The following example illustrates our approach.

Example 1. Consider the predicate `scalar/3` below, and a calling mode to it with the first argument bound to an integer n and the second one bound to a list of integers $[x_1, x_2, \cdots, x_k]$. Upon success, the third argument is bound to the list of products $[n \cdot x_1, n \cdot x_2, \cdots, n \cdot x_k]$. Each product is recursively computed by predicate

`mult/3`. The calling modes are automatically inferred by CiaoPP (see [15] and its references): the first two arguments of both predicates are input, and their last arguments are output.

```
1  scalar(_,[],[]).
2  scalar(N,[X|Xs],[Y|Ys]):-
3     mult(N,X,Y) & scalar(N,Xs,Ys).
4
5  mult(0,_,0).
6  mult(N,X,Y):-
7     N>1,
8     N1 is N - 1,
9     mult(N1,X,Y0),
10    Y is Y0 + X.
```

The call to the parallel `&/2` operator in the body of the second clause of `scalar/3` causes the calls to `mult/3` and `scalar/3` to be executed in parallel.

We want to infer the cost of such a call to `scalar/3`, in terms of the number of resolution steps, as a function of its input data sizes. We use the CiaoPP system to infer size relations for the different arguments in the clauses, as well as dealing with a rich set of size metrics (see [24, 26] for details). Assume that the size metrics used in this example are the *actual value* of `N` (denoted `int(N)`), for the first argument, and the *list-length* for the second and third arguments (denoted `length(X)` and `length(Y)`). Since size relations are obvious in this example, we focus only on the setting up of cost relations for the sake of brevity. Regarding the number of solutions, in this example all the predicates generate at most one solution. For simplicity we assume that all builtin predicates, such as `is/2` and the comparison operators have zero cost (in practice they have a "trust"assertion that specifies their cost as if it had been inferred by the system). As the program contains parallel calls, we are interested in inferring both total resolution steps, i.e., considering a sequential execution (represented by the `seq` identifier), and the number of parallel steps, considering a parallel execution, with infinite number of processors (represented by `par`). In the latter case, the definition of this resource establishes that the aggregator of the costs of the parallel calls that are arguments of the `&/2` meta-predicate is the `max/2` function. Thus, the number of parallel resolution steps for `p & q` is the maximum between the parallel steps performed by `p` and the ones performed by `q`. However, for computing the *total resolution steps*, the aggregation operator we use is the addition, both for parallel and sequential calls. For brevity, in this example we only infer upper bounds on resource usages.

We now set up the cost relations for `scalar/3` and `mult/3`. Note that the cost functions have two arguments, corresponding to the sizes of the input arguments[5]. In the equations, we underline the operation applied as cost aggregator for `&/2`.

For the sequential execution (`seq`), we obtain the following cost relations:

$$
\begin{array}{rll}
\mathtt{C}_{\mathtt{scalar}}(n,l) = 1 & & \text{if } l = 0 \\
\mathtt{C}_{\mathtt{scalar}}(n,l) = \mathtt{C}_{\mathtt{mult}}(n) \underline{+} \mathtt{C}_{\mathtt{scalar}}(n,l-1) + 1 & & \text{if } l > 0 \\
\mathtt{C}_{\mathtt{mult}}(n) = 1 & & \text{if } n = 0 \\
\mathtt{C}_{\mathtt{mult}}(n) = \mathtt{C}_{\mathtt{mult}}(n-1) + 1 & & \text{if } n > 0
\end{array}
$$

[5] For the sake of clarity, we abuse of notation in the examples when representing the cost functions that depend on data sizes.

After solving these equations and composing the closed-form solutions, we obtain the following closed-form functions:

$$\begin{array}{rl} \mathtt{C}_{\mathtt{scalar}}(n,l) = (n+2) \times l + 1 & \text{if } n \geq 0 \wedge l \geq 0 \\ \mathtt{C}_{\mathtt{mult}}(n) = n + 1 & \text{if } n \geq 0 \end{array}$$

For the parallel execution (`par`), we obtain the following cost relations:

$$\begin{array}{rl} \mathtt{C}_{\mathtt{scalar}}(n,l) = 1 & \text{if } l = 0 \\ \mathtt{C}_{\mathtt{scalar}}(n,l) = \overline{\underline{max}}(\mathtt{C}_{\mathtt{mult}}(n), \mathtt{C}_{\mathtt{scalar}}(n,l-1)) + 1 & \text{if } l > 0 \\ \mathtt{C}_{\mathtt{mult}}(n) = \overline{1} & \text{if } n = 0 \\ \mathtt{C}_{\mathtt{mult}}(n) = \mathtt{C}_{\mathtt{mult}}(n-1) + 1 & \text{if } n > 0 \end{array}$$

After solving these equations and composing the closed forms, we obtain the following closed-form functions:

$$\begin{array}{rl} \mathtt{C}_{\mathtt{scalar}}(n,l) = n + l + 1 & \text{if } n \geq 0 \wedge l \geq 0 \\ \mathtt{C}_{\mathtt{mult}}(n) = n + 1 & \text{if } n \geq 0 \end{array}$$

By comparing the complexity order (in terms of resolution steps) of the sequential execution of `scalar/3`, $O(n \cdot l)$, with the complexity order of its parallel execution (assuming an ideal parallel model with an unbounded number of processors) $O(n+l)$, we can get a hint about the maximum achievable parallelization of the program.

Another useful piece of information about `scalar/3` that we want to infer is the maximum number of processes that may run in parallel, considering all possible executions. For this purpose, we define a resource in our framework named `sthreads`. The operation that aggregates the cost of both arguments of the meta-predicate `&/2`, `count_process/3` for the `sthreads` resource, adds the maximum number of processes for each argument plus one additional process, corresponding to the one created by the call to `&/2`. The sequential cost aggregator is now the *maximum* operator, in order to keep track of the maximum number of processes created along the different instructions of the program executed sequentially. Note that if the instruction `p` executes at most Pr_p processes in parallel, and the instruction `q` executes at most Pr_q processes, then the program `p, q` will execute at most $max(Pr_p, Pr_q)$ processes in parallel, because all the parallel processes created by `p` will finish before the execution of `q`. Note also that for the sequential execution of both `p` and `q`, the cost in terms of the `sthreads` resource is always zero, because no additional process is created.
The analysis sets up the following recurrences for the `sthreads` resource and the predicates `scalar/3` and `mult/3` of our example:

$$\begin{array}{rl} \mathtt{C}_{\mathtt{scalar}}(n,l) = 0 & \text{if } l = 0 \\ \mathtt{C}_{\mathtt{scalar}}(n,l) = \mathtt{C}_{\mathtt{mult}}(n) + \mathtt{C}_{\mathtt{scalar}}(n,l-1) + 1 & \text{if } l > 0 \\ \mathtt{C}_{\mathtt{mult}}(n) = 0 & \text{if } n \geq 0 \end{array}$$

After solving these equations and composing the closed forms, we obtain the following closed-form functions:

$$\begin{array}{rl} \mathtt{C}_{\mathtt{scalar}}(n,l) = l & \text{if } n \geq 0 \wedge l \geq 0 \\ \mathtt{C}_{\mathtt{mult}}(n) = 0 & \text{if } n \geq 0 \end{array}$$

As we can see, this predicate will execute, in the worst case, as many processes as there are elements in the input list.

3 The Parametric Cost Relations Framework for Sequential Programs

The starting point of our work is the standard general framework described in [24] for setting up parametric relations representing the resource usage (and size relations) of programs and predicates.[6]

The framework is doubly parametric: first, the costs inferred are functions of input data sizes, and second, the framework itself is parametric with respect to the type of approximation made (upper or lower bounds), and to the resource analyzed. Each concrete resource r to be tracked is defined by two sets of (user-provided) functions, which can be constants, or general expressions of input data sizes:

1. *Head cost* $\varphi_{[ap,r]}(H)$*:* a function that returns the amount of resource r used by the unification of the calling literal (subgoal) $\mathtt{p}$ and the head H of a clause matching $\mathtt{p}$, plus any preparation for entering a clause (i.e., call and parameter passing cost).
2. *Predicate cost* $\Psi_{[ap,r]}(\mathtt{p}, \bar{\mathtt{x}})$: it is also possible to define the *full cost* for a particular predicate $\mathtt{p}$ for resource r and approximation *ap*, i.e., the function $\Psi_{[ap,r]}(\mathtt{p}) : \mathcal{N}_{\top}^{m} \rightarrow \mathcal{R}_{\infty}$ (with the sizes of $\mathtt{p}$'s input data as parameters, $\bar{\mathtt{x}}$) that returns the usage of resource r made by a call to this predicate. This is specially useful for built-in or external predicates, i.e., predicates for which the source code is not available and thus cannot be analyzed, or for providing a more accurate function than analysis can infer. In the implementation, this information is provided to the analyzer through *trust assertions*.

For simplicity we only show the equations related to our standard definition of cost. However, our framework has also been extended to allow the inference of a more general definition of cost, called accumulated cost, which is useful for performing static profiling, obtaining a more detailed information regarding how the cost is distributed among a set of user-defined *cost centers*. See [13, 22] for more details.

Consider a predicate $\mathtt{p}$ defined by clauses $C_1, \ldots, C_m$. Assume $\bar{\mathtt{x}}$ are the sizes of $\mathtt{p}$'s input parameters. Then, the resource usage (expressed in units of resource r with approximation *ap*) of a call to $\mathtt{p}$, for an input of size $\bar{\mathtt{x}}$, denoted as $C_{pred[ap,r]}(\mathtt{p}, \bar{\mathtt{x}})$, can be expressed as:

$$C_{pred[ap,r]}(\mathtt{p}, \bar{\mathtt{x}}) = \bigodot_{1 \leq i \leq m} (C_{cl[ap,r]}(C_i, \bar{\mathtt{x}})) \tag{1}$$

where $\bigodot = ClauseAggregator(ap, r)$ is a function that takes an approximation identifier *ap* and returns a function which applies over the cost of all the clauses, $C_{cl[ap,r]}(C_i, \bar{\mathtt{x}})$, for $1 \leq i \leq m$, in order to obtain the cost of a call to the predicate $\mathtt{p}$. For example, if *ap* is the identifier for approximation "upper bound" (ub), then a possible conservative definition for $ClauseAggregator(ub, r)$ is the $\sum$ function. In this case, and since the number of solutions generated by a predicate that will

[6] We give equivalent but simpler descriptions than in [24], which are allowed by assuming that programs are the result of a normalization process that makes all unifications explicit in the clause body, so that the arguments of the clause head and the body literals are all unique variables. We also change some notation for readability and illustrative purposes.

be demanded is generally not known in advance, a conservative upper bound on the computational cost of a predicate is obtained by assuming that all solutions are needed, and that all clauses are executed (thus the cost of the predicate is assumed to be the sum of the costs of all of its clauses). However, it is straightforward to take mutual exclusion into account to obtain a more precise estimate of the cost of a predicate, using the maximum of the costs of mutually exclusive groups of clauses, as done in [26].

Let us see now how to compute the resource usage of a clause. Consider a clause C of predicate p of the form H :- $L_1, \ldots, L_k$ where $L_j, 1 \leq j \leq k$, is a literal (either a predicate call, or an external or builtin predicate), and H is the clause head. Assume that $\psi_j(\bar{\mathtt{x}})$ is a tuple with the sizes of all the input arguments to literal L_j, given as functions of the sizes of the input arguments to the clause head. Note that these $\psi_j(\bar{\mathtt{x}})$ size relations have previously been computed during size analysis for all input arguments to literals in the bodies of all clauses. Then, the cost relation for clause C and a single call to p (obtaining all solutions), is:

$$C_{cl[ap,r]}(C, \bar{\mathtt{x}}) = \varphi_{[ap,r]}(head(C)) + \sum_{j=1}^{lim(ap,C)} sols_j(\bar{\mathtt{x}}) \times C_{lit[ap,r]}(L_j, \psi_j(\bar{\mathtt{x}})) \qquad (2)$$

where $lim(ap, C)$ gives the index of the last body literal that is called in the execution of clause C, $\psi_j(\bar{\mathtt{x}})$ are the sizes of the input parameters of literal L_j, and $sols_j$ represents the product of the number of solutions produced by the predecessor literals of L_j in the clause body:

$$sols_j(\bar{\mathtt{x}}) = \prod_{i=1}^{j-1} s_{pred}(L_i, \psi_i(\bar{\mathtt{x}})) \qquad (3)$$

where $s_{pred}(L_i, \psi_i(\bar{\mathtt{x}}))$ gives the number of solutions produced by L_i, with arguments of size $\psi_i(\bar{\mathtt{x}})$. The number of solutions and size relations are both inferred automatically by the framework (we refer the reader to [10, 8, 11, 26] for descriptions of this process).

Finally, $C_{lit[ap,r]}(L_j, \psi_j(\bar{\mathtt{x}}))$ is replaced by one of the following expressions, depending on L_j:

- If L_j is a call to a predicate q which is in the same strongly connected component as p (the predicate under analysis), then $C_{lit[ap,r]}(L_j, \psi_j(\bar{\mathtt{x}}))$ is replaced by the symbolic call $C_{pred[ap,r]}(\mathtt{q}, \psi_\mathtt{j}(\bar{\mathtt{x}}))$, giving rise to a recurrence relation that needs to be bounded with a closed-form expression by the solver afterwards.
- If L_j is a call to a predicate q which is in a different strongly connected component than p, then $C_{lit[ap,r]}(L_j, \psi_j(\bar{\mathtt{x}}))$ is replaced by the closed-form expression that bounds $C_{pred[ap,r]}(\mathtt{q}, \psi_\mathtt{j}(\bar{\mathtt{x}}))$. The analysis guarantees that this expression has been inferred beforehand, due to the fact that the analysis is performed for each strongly connected component, in a reverse topological order.
- If L_j is a call to a predicate q, whose cost is specified (with a trust assertion) as $\Psi_{[ap,r]}(q, \bar{\mathtt{x}})$, then $C_{lit[ap,r]}(L_j, \psi_j(\bar{\mathtt{x}}))$ is replaced by the expression $\Psi_{[ap,r]}(q, \psi_j(\bar{\mathtt{x}}))$.

4 Our Extended Resource Analysis Framework for Parallel Programs

In this section, we describe how we extend the resource analysis framework detailed above, in order to handle logic programs with Independent And-Parallelism, using the binary parallel `&/2` operator. First, we introduce a new general parameter that indicates the execution model the analysis has to consider. For our current prototype, we have defined two different execution models: standard *sequential* execution, represented by *seq*, and an abstract parallel execution model, represented by $par(n)$, where $n \in \mathbb{N} \cup \{\infty\}$. The abstract execution model $par(\infty)$ is similar to the *work* and *depth* model, presented in [6] and used extensively in previous work such as [18]. Basically, this model is based on considering an unbounded number of available processors to infer bounds on the depth of the computation tree. The *work* measure is the amount of work to be performed considering a sequential execution. These two measures together give an idea on the impact of the parallelization of a particular program. The abstract execution model $par(n)$, where $n \in \mathbb{N}$, assumes a finite number n of processors.

In order to obtain the cost of a predicate, equation (1) remains almost identical, with the only difference of adding the new parameter to indicate the execution model.

Now we address how to set the cost for clauses. In this case, equation (2) is extended with the execution model ex, and also the Σ default sequential cost aggregation is replaced by a parametric associative operator $\bigoplus$, that depends on the resource being defined, the approximation, and the execution model. For $ex \equiv par(\infty)$ or $ex \equiv seq$, the following equation is set up:

$$C_{cl[ap,r,ex]}(C, \bar{\mathtt{x}}) = \varphi_{[ap,r]}(head(C)) + \bigoplus_{j=1}^{lim(ap,ex,C)} (sols_j(\bar{\mathtt{x}}) \times C_{lit[ap,r,ex]}(L_j, \psi_j(\bar{\mathtt{x}}))) \quad (4)$$

Note that the cost aggregator operators must depend on the resource r (besides the other parameters). For example, if r is *execution time*, then the cost of executing two tasks in parallel must be aggregated by taking the maximun of the execution times of the two tasks. In contrast, if r is *energy consumption*, then the aggregation is the addition of the energy of the two tasks.

Finally, we extend how the cost of a literal L_i, expressed as $C_{lit[ap,r,ex]}(L_i, \psi_i(\bar{\mathtt{x}}))$, is set up. The previous definition is extended considering the new case where the literal is a call to the *meta-predicate* `&/2`. In this case, we add a new parallel aggregation associative operator, denoted by $\bigotimes$. Concretely, if $L_i = B_1 \& B_2$, where B_1 and B_2 are two sequences of goals, then:

$$C_{lit[ap,r,ex]}(B_1 \& B_2, \bar{\mathtt{x}}) = C_{body[ap,r,ex]}(B_1, \bar{\mathtt{x}}) \bigotimes C_{body[ap,r,ex]}(B_2, \bar{\mathtt{x}}) \quad (5)$$

$$C_{body[ap,r,ex]}(B, \bar{\mathtt{x}}) = \bigoplus_{j=1}^{lim(ap,ex,B)} (sols_j(\bar{\mathtt{x}}) \times C_{lit[ap,r,ex]}(L_j^B, \psi_j(\bar{\mathtt{x}}))) \quad (6)$$

where $B = L_1^B, \cdots, L_m^B$.

Consider now the execution model $ex \equiv par(n)$, where $n \in \mathbb{N}$ (i.e., assuming a finite number n of processors), and a recursive parallel predicate $\mathtt{p}$ that creates a parallel task $\mathtt{q}_i$ in each recursion i. Assume that we are interested in obtaining an upper bound on the cost of a call to $\mathtt{p}$, for an input of size $\bar{\mathtt{x}}$. We first infer the number k of parallel tasks created by $\mathtt{p}$ as a function of $\bar{\mathtt{x}}$. This can be easily done by using our cost analysis framework and providing the suitable assertions for inferring a resource named "*ptasks*." Intuitively, the "counter" associated to such resource must be incremented by the (symbolic) execution of the `&/2` parallel operator. More formally, $k = C_{pred[ub,ptasks]}(\mathtt{p}, \bar{\mathtt{x}})$. To this point, an upper bound m on the number of tasks executed by any of the n processors is given by $m = \lceil \frac{k}{n} \rceil$. Then, an upper bound on the cost (in terms of resolution steps, i.e., $r = steps$) of a call to $\mathtt{p}$, for an input of size $\bar{\mathtt{x}}$ can be given by:

$$C_{pred[ub,r,par(n)]}(\mathtt{p}, \bar{\mathtt{x}}) = C^u + Spaw^u \tag{7}$$

where C^u can be computed in two possible ways: $C^u = \sum_{i=1}^{m} C_i^u$; or $C^u = m\ C_1^u$, where C_i^u denotes an upper bound on the cost of parallel task $\mathtt{q}_i$, and $C_1^u, \ldots, C_k^u$ are ordered in descending order of cost. Each C_i^u can be considered as the sum of two components: $C_i^u = Sched_i^u + T_i^u$, where $Sched_i^u$ denotes the cost from the point in which the parallel subtask $\mathtt{q}_i$ is created until its execution is started by a processor (possibly the same processor that created the subtask), i.e. the cost of task preparation, scheduling, communication overheads, etc. T_i^u denotes the cost of the execution of $\mathtt{q}_i$ disregarding all the overheads mentioned before, i.e., $T_i^u = C_{pred[ub,r,seq]}(\mathtt{q}, \psi_q(\bar{\mathtt{x}}))$, where $\psi_q(\bar{\mathtt{x}})$ is a tuple with the sizes of all the input arguments to predicate $\mathtt{q}$ in the body of $\mathtt{p}$. $Spaw^u$ denotes an upper bound on the cost of creating the k parallel tasks $\mathtt{q}_i$. It will be dependent on the particular system in which $\mathtt{p}$ is going to be executed. It can be a constant, or a function of several parameters, (such as input data size, number of input arguments, or number of tasks) and can be experimentally determined.

In addition, we propose a method for finding closed-form functions for cost relations that arise in the analysis of parallel programs, where the *max* function usually plays a role both as parallel and sequential cost aggregation operation, i.e., as $\bigotimes$ and $\bigoplus$ respectively. In the following subsection, we detail these methods.

4.1 Solving Cost Recurrence Relations Involving *max* Operation

Automatically finding closed-form upper and lower bounds for recurrence relations is an uncomputable problem. For some special classes of recurrences, exact solutions are known, for example for linear recurrences with one variable. For some other classes, it is possible to apply transformations to fit a class of recurrences with known solutions, even if this transformation obtains an appropriate approximation rather than an equivalent expression.

Particularly for the case of analyzing independent and-parallel logic programs, nonlinear recurrences involving the *max* operator are quite common. For example, if we are analyzing elapsed time of a parallel logic program, a proper parallel aggregation operator is the maximum between the times elapsed for each literal running in parallel. To the best of our knowledge, no general solution exists for recurrences of this particular type. However, in this paper we identify some common cases of this type of recurrences, for which we obtain closed forms that are proven

to be correct. In this section, we present these different classes, together with the corresponding method to obtain a correct bound.

Consider the following function $f : \mathbb{N}^m \to \mathbb{N}$, defined as a general form of a first-order recurrence equation with a *max* operator:

$$f(\bar{\mathtt{x}}) = \begin{cases} max(C, f(\bar{\mathtt{x}}_{|i} - 1)) + D & x_i > \Theta \\ B & x_i \leq \Theta \end{cases} \tag{8}$$

where $\Theta \in \mathbb{N}$. C, D and B are arbitrary expressions possibly depending on $\bar{\mathtt{x}}$. Note that $\bar{\mathtt{x}} = x_1, x_2, \cdots, x_m$. We define $\bar{\mathtt{x}}_{|i} - 1 = x_1, \cdots, x_i - 1, \cdots, x_m$, for a given i, $1 \leq i \leq m$. If C and D do not depend on x_i, then C and D do not change through the different recursive instances of f. In this case, a closed-form upper bound is defined by the following theorem (whose proof is included in A):

Theorem 1. *Given $f : \mathbb{N}^m \to \mathbb{N}$ as defined in (8), where C and D are non-decreasing functions of $\bar{\mathtt{x}} \setminus x_i$. Then, $\forall \bar{\mathtt{x}}$:*

$$f(\bar{\mathtt{x}}) = f^{'}(\bar{\mathtt{x}}) = \begin{cases} max(C, B) + (x_i - \Theta) \cdot D & x_i > \Theta \\ B & x_i \leq \Theta \end{cases}$$

For the case where $C = g(\bar{\mathtt{x}})$ and $D = h(\bar{\mathtt{x}})$ are functions non-decreasing on x_i, then the upper bound is given by the following closed form:

Theorem 2. *Given $f : \mathbb{N}^m \to \mathbb{N}$ as defined in (8), where g and h are functions of $\bar{\mathtt{x}}$, non-decreasing on x_i. Then, $\forall \bar{\mathtt{x}}$:*

$$f(\bar{\mathtt{x}}) \leq f'(\bar{\mathtt{x}}) = \begin{cases} max(g(\bar{\mathtt{x}}), B) + (x_i - \Theta - 1) \times max(g(\bar{\mathtt{x}}), h(\bar{\mathtt{x}}_{|i} - 1)) + h(\bar{\mathtt{x}}_{|i}) & x_i > \Theta \\ B & x_i \leq \Theta \end{cases}$$

The proof of this Theorem is included in B.

For the remaining cases, where a $max(e_1, e_2)$ appears, we try to eliminate the *max* operator by proving either $e_1 \leq e_2$ or $e_2 \leq e_1$, for any input. In order to do that, we use the function comparison capabilities of CiaoPP, presented in [20, 21]. In cases where e_1 and/or e_2 contains non-closed recurrence functions, we use the Z3 SMT solver [7] to find, if possible, a proof either for $e_1 \leq e_2$ or $e_2 \leq e_1$, treating the non-closed functions as uninterpreted functions, assuming that they are positive and non-decreasing. As the algorithm used by SMT solvers in this case is not guaranteed to terminate, we set a timeout. In the worst case, when no proof is found, then we replace the *max* operator with an addition, loosing precision but still finding safe upper bounds.

5 Implementation and Experimental Results

We have implemented a prototype of our approach, leveraging the existing resource usage analysis framework of CiaoPP. The implementation basically consists of the parameterization of the operators used for sequential and parallel cost aggregation, i.e., for the aggregation of the costs corresponding to the arguments of `,/2` and `&/2`, respectively. This allows the user to define resources in a general way, taking into account the underlying execution model.

Table 1. Description of the benchmarks.

`map_add1/2`	Parallel increment by one of each element of a list.
`fib/2`	Parallel computation of the nth Fibonacci number.
`mmatrix/3`	Parallel matrix multiplication.
`blur/2`	Generic parallel image filter.
`add_mat/3`	Matrix addition.
`intersect/3`	Set intersection.
`union/3`	Set union.
`diff/3`	Set difference.
`dyade/3`	Dyadic product of two vectors.
`dyade_map/3`	Dyadic product applied on a set of vectors.
`append_all/3`	Appends a prefix to each list of a list of lists.

Table 2. Resource usage inferred for Independent And-Parallel Programs.

Bench	Res	Bound Inferred	BigO	T_A(ms)
`map_add1(x)`	SCost	$2 \cdot l_x + 1$	$\mathcal{O}(l_x)$	35.57
	PCost	$2 \cdot l_x + 1$	$\mathcal{O}(l_x)$	
	STheads	l_x	$\mathcal{O}(l_x)$	
`fib(x)`	SCost	$F(i_x) + L(i_x) - 1$	$\mathcal{O}(2^{i_x})$	52.66
	PCost	$i_x + 1$	$\mathcal{O}(i_x)$	
	STheads	$F(i_x) + L(i_x) - 1$	$\mathcal{O}(2^{i_x})$	
`mmatrix`(m_1, n_1, m_2, n_2)	SCost	$i_{n_2} \cdot i_{m_2} \cdot i_{m_1} + 2 \cdot i_{m_2} \cdot i_{m_1} + 2 \cdot i_{m_1} + 1$	$\mathcal{O}(i_{n_2} \cdot i_{m_2} \cdot i_{m_1})$	220.9
	PCost	$2 \cdot i_{m_1} + i_{n_1} + 1$	$\mathcal{O}(i_{n_1} + i_{m_1})$	
	STheads	$i_{m_2} \cdot i_{m_1} + i_{m_1}$	$\mathcal{O}(i_{m_2} \cdot i_{m_1})$	
`blur(m,n)`	SCost	$2 \cdot i_m \cdot i_n + 2 \cdot i_n + 1$	$\mathcal{O}(i_m \cdot i_n)$	123.321
	PCost	$2 \cdot i_m + 2 \cdot i_n + 1$	$\mathcal{O}(i_m + i_n)$	
	STheads	i_n	$\mathcal{O}(i_n)$	
`add_mat(m,n)`	SCost	$i_m \cdot i_n + 2 \cdot i_n + 1$	$\mathcal{O}(i_m \cdot i_n)$	62.72
	PCost	$i_m + 2 \cdot i_n + 1$	$\mathcal{O}(i_m + i_n)$	
	STheads	i_n	$\mathcal{O}(i_n)$	
`intersect`(a, b)	SCost	$l_a \cdot l_b + 3 \cdot l_a + 3$	$\mathcal{O}(l_x)$	191.16
	PCost	$l_b + 2 \cdot l_a + 3$	$\mathcal{O}(i_n)$	
	STheads	l_a	$\mathcal{O}(l_x)$	
`union`(a, b)	SCost	$l_a \cdot l_b + 3 \cdot l_a + 3$	$\mathcal{O}(l_a \cdot l_b)$	193.37
	PCost	$2 \cdot l_b + 2 \cdot l_a + 3$	$\mathcal{O}(l_a + l_b)$	
	STheads	l_a	$\mathcal{O}(l_a)$	
`diff`(a, b)	SCost	$l_a \cdot l_b + 3 \cdot l_a + 3$	$\mathcal{O}(l_a \cdot l_b)$	191.16
	PCost	$l_b + 2 \cdot l_a + 3$	$\mathcal{O}(l_a + l_b)$	
	STheads	l_a	$\mathcal{O}(l_a)$	
`dyade`(a, b)	SCost	$l_a \cdot l_b + 2 \cdot l_a + 1$	$\mathcal{O}(l_a \cdot l_b)$	71.08
	PCost	$l_b + l_a + 1$	$\mathcal{O}(l_a + l_b)$	
	STheads	l_a	$\mathcal{O}(l_a)$	
`dyade_map`(l, m)	SCost	$i_{max(m)} \cdot l_m \cdot l_l + 2 \cdot l_m \cdot l_l + 2 \cdot l_m + 1$	$\mathcal{O}(i_{max(m)} \cdot l_m \cdot l_l)$	248.39
	PCost	$i_{max(m)} + l_m + l_l + 1$	$\mathcal{O}(i_{max(m)} + l_m + l_l)$	
	STheads	$l_l \cdot l_m + l_l$	$\mathcal{O}(l_m \cdot l_l)$	
`append_all`(l, m)	SCost	$l_l \cdot l_m + 2 \cdot l_m + 1$	$\mathcal{O}(l_l \cdot l_m)$	108.4
	PCost	$l_l + l_m + 1$	$\mathcal{O}(l_l + l_m)$	
	STheads	l_m	$\mathcal{O}(l_m)$	

$F(n)$ represents the nth element of the Fibonacci sequence.
$L(n)$ represents the nth Lucas number.
l_n, i_n represent the size of n in terms of the metrics *length* and *int*, respectively.

We selected a set of benchmarks that exhibit different common parallel patterns, briefly described in Table 1, together with the definition of a set of resources that help understand the overall behavior of the parallelization. Table 2 shows some results of the experiments that we have performed with our prototype implementation. Column **Bench** shows the main predicates analyzed for each benchmark. Set operations (`intersect`, `union` and `diff`), as well as the programs `append_all`,

Table 3. Resource usage inferred for a bounded number of processors.

Bench	Bound Inferred	BigO	$\mathbf{T_A}$(ms)
`map_add1(x)`	$2 \cdot \lceil \frac{l_x}{p} \rceil + 1$	$\mathcal{O}(\lceil \frac{l_x}{p} \rceil)$	54.36
`blur(m,n)`	$2 \cdot \lceil \frac{i_n}{p} \rceil \cdot i_m + 2 \cdot \lceil \frac{i_n}{p} \rceil + 1$	$\mathcal{O}(\lceil \frac{i_n}{p} \rceil \cdot i_m)$	205.97
`add_mat(m,n)`	$\lceil \frac{i_n}{p} \rceil \cdot i_m + 2 \cdot \lceil \frac{i_n}{p} \rceil + 1$	$\mathcal{O}(\lceil \frac{i_n}{p} \rceil \cdot i_m)$	185.89
`intersect`(a, b)	$\lceil \frac{l_a}{p} \rceil \cdot l_b + 2 \cdot \lceil \frac{l_a}{p} \rceil + l_a + 2$	$\mathcal{O}(\lceil \frac{l_a}{p} \rceil \cdot l_b)$	330.47
`union`(a, b)	$\lceil \frac{l_a}{p} \rceil \cdot l_b + 2 \cdot \lceil \frac{l_a}{p} \rceil + l_a + l_b + 2$	$\mathcal{O}(\lceil \frac{l_a}{p} \rceil \cdot l_b)$	311.3
`diff`(a, b)	$\lceil \frac{l_a}{p} \rceil \cdot l_b + 2 \cdot \lceil \frac{l_a}{p} \rceil + l_a + 2$	$\mathcal{O}(\lceil \frac{l_a}{p} \rceil \cdot l_b)$	339.01
`dyade`(a, b)	$\lceil \frac{l_a}{p} \rceil \cdot l_b + 2 \cdot \lceil \frac{l_a}{p} \rceil + 1$	$\mathcal{O}(\lceil \frac{l_a}{p} \rceil \cdot l_b)$	120.93
`append_all`(l, m)	$\lceil \frac{l_m}{p} \rceil \cdot l_l + 2 \cdot \lceil \frac{l_m}{p} \rceil + 1$	$\mathcal{O}(\lceil \frac{l_m}{p} \rceil \cdot l_l)$	117.8

`p` is defined as the minimum between the number of processors and **SThreads**.

`dyade` and `add_mat`, are Prolog versions of the benchmarks analyzed in [18], which is the closest related work we are aware of.

Column **Res** indicates the name of each of the resources inferred for each benchmark: *sequential resolution steps* (**SCost**), *parallel resolution steps* assuming an unbounded number of processors (**PCost**), and *maximum number of processes executing in parallel* (**SThreads**). The latter gives an indication of the maximum parallelism that can potentially be exploited. Column **Bound Inferred** shows the upper bounds obtained for each of the resources indicated in Column **Res**. While in the experiments both upper and lower bounds were inferred, for the sake of brevity, we only show upper-bound functions. Column **BigO** shows the complexity order, in big O notation, corresponding to each resource. For all the benchmarks in Table 2 we obtain the exact complexity orders. We also obtain the same complexity order as in [18] for the Prolog versions of the benchmarks taken from that work. Finally, Column $\mathbf{T_A}$(**ms**) shows the analysis times in milliseconds. The results show that most of the benchmarks have different asymptotic behavior in the sequential and parallel execution models. In particular, for `fib(x)`, the analysis infers an exponential upper bound for sequential execution steps, and a linear upper bound for parallel execution steps. As mentioned before, this is an upper bound for an ideal case, assuming an unbounded number of processors. Nevertheless, such upper-bound information is useful for understanding how the cost behavior evolves in architectures with different levels of parallelism. In addition, this *dual* cost measure can be combined together with a bound on the number of processors in order to obtain a general asymptotic upper bound (see for example Brent's Theorem [14], which is also mentioned in [18]). The program `map_add1(l)` exhibits a different behavior: both sequential and parallel upper bounds are linear. This happens because we are considering *resolution steps*, i.e., we are counting each head unification produced from an initial call `map_add1(l)`. Even under the parallel execution model, we have a chain of head unifications whose length depends linearly on the length of the input list. It follows from the results of this particular case that this simple, non-associative parallelization will not be useful for improving the number of resolution steps performed in parallel.

Another useful information inferred in our experiments is the maximum number of processes that can be executed in parallel, represented by the resource named **SThreads**. We can see that for most of our examples the analysis obtains a linear upper bound for this resource, in terms of the size of some of the inputs. For example, the execution of `intersect(a,b)` (parallel set intersection) will create *at most* l_a processes, where l_a represents the length of the list a. For other examples,

the analysis shows a quadratic upper bound (as in `mmatrix`), or even exponential bounds (as in `fib`). The information about upper bounds on the maximum level of parallelism required by a program is useful for understanding its scalability in different parallel architectures, or for optimizing the number of processors that a particular call will use, depending on the size of the input data.

Finally, the results of our experiments considering a bounded number of processors are shown in Table 3.

6 Related Work

Our approach is an extension of an existing cost analysis framework for sequential logic programs [11, 13, 21], which extends the classical cost analysis techniques based on setting up and solving recurrence relations, pioneered by [28], with solutions for relations involving `max` and `min` functions. The framework handles characteristics such as backtracking, multiple solutions (i.e., non-determinism), failure, and inference of both upper and lower bounds including non-polynomial bounds. These features are inherited by our approach, and are absent from other approaches to parallel cost analysis in the literature.

The most closely-related work to our approach is [18], which describes an automatic analysis for deriving bounds on the worst-case evaluation cost of first order functional programs. The analysis derives bounds under an abstract *dual* cost model based on two measures: *work* and *depth*, which over-approximate the sequential and parallel evaluation cost of programs, respectively, considering an unlimited number of processors. Such an abstract cost model was introduced by [6] to formally analyze parallel programs. The work is based on type judgments annotated with a cost metric, which generate a set of inequalities which are then solved by linear programming techniques. Their analysis is only able to infer multivariate resource polynomial bounds, while non-polynomial bounds are left as future work. In [17] the authors propose an automatic analysis based on the *work* and *depth* model, for a simple imperative language with explicit parallel loops.

There are other approaches to cost analysis of parallel and distributed systems, based on different models of computation than the independent and-parallel model in our work. In [3] the authors present a static analysis which is able to infer upper bounds on the maximum number of *active* (i.e., not finished nor suspended) processes running in parallel, and the total number of processes created for imperative *async-finish* parallel programs. The approach described in [1] uses recurrence (cost) relations to derive upper bounds on the cost of concurrent object-oriented programs, with shared-memory communication and future variables. They address concurrent execution for loops with semi-controlled scheduling, i.e., with no arbitrary interleavings. In [4] the authors address the cost of parallel execution of object-oriented distributed programs. The approach is to identify the synchronization points in the program, use serial cost analysis of the blocks between these points, and then, exploiting the techniques mentioned, construct a graph structure to capture the possible parallel execution of the program. The path of maximal cost is then computed. The allocation of tasks to processors (called "locations") is part of the program in these works, and thus, although independent and-parallel programs could be modeled in this computation style, it is not directly comparable to our more abstract model of parallelism.

Solving, or safely bounding recurrence relations with `max` and `min` functions has been addressed mainly for recurrences derived from divide-and-conquer algorithms [5, 27, 19]. In [2] the authors present solutions for Cost Relation Systems by obtaining upper bounds for both the number of nodes and the cost added in each node, in the derived evaluation tree. These bounds are then combined in order to obtain a closed-form, upper-bound expression. This closed form possibly contains maximization operations to express upper bounds for a set of subexpressions. However, each cost relation is defined as a summatory of costs, while in our work we consider other operations for aggregating the costs, including `max` operators. The presence of these operators often generates recurrence relations where the recursive calls are under the scope of such a `max` operator, for which we present a method to obtain a closed-form bound. This class of recurrences are not handled by most of the current computer algebra systems, as the authors in [2] mention.

7 Conclusions

We have presented a novel, general, and flexible analysis framework that can be instantiated for estimating the resource usage of parallel logic programs, for a wide range of resources, platforms, and execution models. To the best of our knowledge, this is the first approach to the cost analysis of *parallel logic programs*. Such estimations include both lower and upper bounds, given as functions on input data sizes. In addition, our analysis also infers other information which is useful for improving the exploitation and assessing the potential and actual parallelism of a program. We have also developed a method for solving the cost relations that arise in this particular type of analysis, which involve the *max* function. Finally, we have developed a prototype implementation of our general framework, instantiated it for the analysis of logic programs with Independent And-Parallelism, and performed an experimental evaluation, obtaining very encouraging results w.r.t. accuracy and efficiency.

References

1. E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO programs. In *Proc. of APLAS'11*, volume 7078 of *LNCS*, pages 238–254. Springer, 2011.
2. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.
3. E. Albert, P. Arenas, S. Genaim, and D. Zanardini. Task-Level Analysis for a Language with Async-Finish parallelism. In *Proc. of LCTES'11*, pages 21–30. ACM Press, 2011.
4. E. Albert, J. Correas, E. Johnsen, K.I. Pu, and G. Román-Díez. Parallel cost analysis. *ACM Trans. Comput. Logic*, 19(4), November 2018.
5. L. Alonso, E.M. Reingold, and R. Schott. Multidimensional divide-and-conquer maximin recurrences. *SIAM J. Discret. Math.*, 8(3):428–447, August 1995.
6. Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *ACM Int'l. Conf. on Functional Programming*, pages 213–225, May 1996.
7. L. Mendonça de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
8. S. K. Debray and N. W. Lin. Cost analysis of logic programs. *ACM TOPLAS*, 15(5):826–875, November 1993.

9. S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.
10. S. K. Debray, N.-W. Lin, and M. V. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. PLDI'90*, pages 174–188. ACM, June 1990.
11. S. K. Debray, P. Lopez-Garcia, M. V. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *ILPS'97*, pages 291–305. MIT Press, 1997.
12. G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. V. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM TOPLAS*, 23(4):472–602, July 2001.
13. R. Haemmerlé, P. Lopez-Garcia, U. Liqat, M. Klemen, J. P. Gallagher, and M. V. Hermenegildo. A Transformational Approach to Parametric Accumulated-cost Static Profiling. In *FLOPS'16*, volume 9613 of *LNCS*, pages 163–180. Springer, 2016.
14. Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2 edition, 2016.
15. M. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez Garcia. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, 2005.
16. M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.
17. T. Hoefler and G. Kwasniewski. Automatic complexity analysis of explicitly parallel programs. In *26th ACM Symp. on Parallelism in Algorithms and Architectures*, SPAA '14, pages 226–235, 2014.
18. J. Hoffmann and Z. Shao. Automatic static cost analysis for parallel programs. In Jan Vitek, editor, *Programming Languages and Systems*, pages 132–157, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
19. H. Hwang and T.-H. Tsai. An asymptotic theory for recurrence relations based on minimization and maximization. *Theoretical Computer Science*, 290(3):1475 – 1501, 2003.
20. P. Lopez-Garcia, L. Darmawan, and F. Bueno. A Framework for Verification and Debugging of Resource Usage Properties. In *Technical Communications of ICLP*, volume 7 of *LIPIcs*, pages 104–113. Schloss Dagstuhl, July 2010.
21. P. Lopez-Garcia, L. Darmawan, M. Klemen, U. Liqat, F. Bueno, and M. V. Hermenegildo. Interval-based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption. *TPLP*, 18:167–223, March 2018.
22. P. Lopez-Garcia, M. Klemen, U. Liqat, and M. V. Hermenegildo. A General Framework for Static Profiling of Parametric Resource Usage. *TPLP*, 16(5-6):849–865, 2016.
23. M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *LOPSTR*, volume 4915 of *LNCS*, pages 154–168. Springer-Verlag, August 2007.
24. J. Navas, E. Mera, P. Lopez-Garcia, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *Proc. of ICLP'07*, volume 4670 of *LNCS*, pages 348–363. Springer, 2007.
25. M. Rosendahl. Automatic Complexity Analysis. In *Proc. of FPCA'89*, pages 144–156. ACM Press, 1989.
26. A. Serrano, P. Lopez-Garcia, and M. V. Hermenegildo. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *TPLP, ICLP'14 Special Issue*, 14(4-5):739–754, 2014.
27. B.-F. Wang. Tight bounds on the solutions of multidimensional divide-and-conquer maximin recurrences. *Theoretical Computer Science*, 242(1):377 – 401, 2000.
28. B. Wegbreit. Mechanical Program Analysis. *Comm. of the ACM*, 18(9):528–539, 1975.

Appendices

A Proof for Theorem 1

Theorem *Given $f : \mathbb{N}^m \to \mathbb{N}$ as defined in (8), where C and D are non-decreasing functions of $\bar{\mathtt{x}} \setminus x_i$. Then, $\forall \bar{\mathtt{x}}$:*

$$f(\bar{\mathtt{x}}) = f^{'}(\bar{\mathtt{x}}) = \begin{cases} max(C, B) + (x_i - \Theta + 1) \cdot D & x_i > \Theta \\ B & x_i \leq \Theta \end{cases}$$

Proof. The proof for the case $x_i \leq \Theta$ is trivial.
In the following, we prove the theorem for $x_i > \Theta$, or equivalently, for $x_i \geq \Theta + 1$. The proof is by induction on this subset.

Base Case. We have to prove that $f(x_1, \cdots, x_{i-1}, \Theta + 1, \cdots, x_m) = f'(x_1, \cdots, x_{i-1}, \Theta + 1, \cdots, x_m)$. Using the definition of f and f' we have that

$$\begin{aligned} f(x_1, \cdots, x_{i-1}, \Theta + 1, \cdots, x_m) &= max(C, f(x_1, \cdots, x_{i-1}, \Theta, \cdots, x_m)) + D \\ &= max(C, B) + D \\ f'(x_1, \cdots, x_{i-1}, \Theta + 1, \cdots, x_m) &= max(C, B) + (\Theta + 1 - \Theta) \cdot D \\ &= max(C, B) + D \end{aligned}$$

General Case. Assuming
$f(x_1, \cdots, x_{i-1}, x_i, \cdots, x_m) = f'(x_1, \cdots, x_{i-1}, x_i, \cdots, x_m)$, we need to prove that $f(x_1, \cdots, x_{i-1}, x_i + 1, \cdots, x_m) = f'(x_1, \cdots, x_{i-1}, x_i + 1, \cdots, x_m)$. By induction hypothesis we have that:

$$\begin{aligned} f(x_1, \cdots, x_{i-1}, x_i + 1, \cdots, x_m) &= max(C, f(x_1, \cdots, x_{i-1}, x_i, \cdots, x_m)) + D \\ &= max(C, max(C, B) + (x_i - \Theta) \cdot D) + D \\ &= max(C, B) + (x_i - \Theta) \cdot D + D \\ &= max(C, B) + (x_i - \Theta + 1) \cdot D \\ &= f'(x_1, \cdots, x_{i-1}, x_i + 1, \cdots, x_m) \end{aligned}$$

B Proof of Theorem 2

For all $a, b, c \in \mathbb{N} \cup \{0\}$, the following properties hold:

- Commutative: $max(a, b) = max(b, a)$
- Associative: $max(a, max(b, c)) = max(max(a, b), c)$
- Idempotent: $max(a, a) = a$

Lemma 1. $\forall a, b, c \in \mathbb{N} : max(a, b + c) \leq max(a, b) + max(a, c)$

Lemma 2. $\forall a, b, c, d \in \mathbb{N} : a \leq c \wedge b \leq d \implies max(a, b) \leq max(c, d)$

Theorem *Given $f : \mathbb{N}^m \to \mathbb{N}$ as defined in (8), where g and h are functions of $\bar{\mathtt{x}}$, non-decreasing on x_i. Then, $\forall \bar{\mathtt{x}}$:*

$$f(\bar{\mathtt{x}}) \leq f'(\bar{\mathtt{x}}) = \begin{cases} max(g(\bar{\mathtt{x}}), B) + (x_i - \Theta - 1) \times max(g(\bar{\mathtt{x}}), h(\bar{\mathtt{x}}_{|i} - 1)) + h(\bar{\mathtt{x}}_{|i}) & x_i > \Theta \\ B & x_i \leq \Theta \end{cases}$$

Proof. The proof for the case $x_i \leq \Theta$ is trivial.
In the following, we prove the theorem for $x_i > \Theta$, or equivalently, for $x_i \geq \Theta + 1$. The proof is by induction on this subset. For brevity, we only show the argument corresponding to the position of x_i in $\bar{\mathtt{x}}$. However, the proof is still valid considering all of the arguments.

Base Case. We have to prove that $f(\Theta + 1) \leq f'(\Theta + 1)$. Using the definition of f and f' we have that

$$\begin{aligned} f(\Theta + 1) &= max(g(\Theta + 1), f(\Theta)) + h(\Theta + 1) \\ &= max(g(\Theta + 1), B) + h(\Theta + 1) \\ f'(\Theta + 1) &= max(g(\Theta + 1), B) + ((\Theta + 1) - \Theta - 1) \times max(g(\Theta + 1), h(\Theta)) + h(\Theta + 1) \\ &= max(g(\Theta + 1), B) + h(\Theta + 1) \end{aligned}$$

General Case. Assuming $f(x) \leq f'(x)$, we need to prove that $f(x+1) \leq f'(x+1)$. By induction hypothesis and Lemma 2 we have that:

$$\begin{aligned} f(x + 1) &= max(g(x + 1), f(x)) + h(x + 1) \\ &\leq max(g(x + 1), max(g(x), B) + (x - \Theta - 1) \times max(g(x), h(x - 1)) + h(x)) + h(x + 1) \end{aligned}$$

By Lemma 1 we have that:

$$\begin{aligned} f(x + 1) \leq\ & max(g(x + 1), max(g(x), B)) \\ & + max(g(x + 1), (x - \Theta - 1) \times max(g(x), h(x - 1))) \\ & + max(g(x + 1), h(x)) \\ & + h(x + 1) \end{aligned} \tag{9}$$

Consider now the first term appearing in the sum of the right hand side of the inequality (9). Since max is associative, and it holds that $\forall x : g(x+1) \geq g(x)$ (which follows from the hypothesis of the theorem), we obtain:

$$\begin{aligned} max(g(x+1), max(g(x), B)) &= max(max(g(x+1), g(x)), B) \\ &= max(g(x+1), B) \end{aligned} \tag{10}$$

We consider now the second term in (9). By Lemma 1 we obtain:

$$\begin{aligned} &max(g(x+1), (x - \Theta - 1) \times max(g(x), h(x-1))) \\ &\leq (x - \Theta - 1) \times max(g(x+1), max(g(x), h(x-1))) \end{aligned}$$

As before, by associativity of max, this is equivalent to:

$$(x - \Theta - 1) \times max(g(x+1), h(x-1))$$

By Lemma 2, and $h(x-1) \leq h(x)$ (by hypothesis), we have that:

$$(x - \Theta - 1) \times max(g(x+1), h(x)) \tag{11}$$

Replacing the results of (10) and (11) in (9):

$$\begin{aligned} f(x+1) \leq\ & max(g(x+1), B) \\ & + (x - \Theta - 1) \times max(g(x+1), h(x)) \\ & + max(g(x+1), h(x)) + h(x+1) \\ & = max(g(x+1), B) \\ & + (x - \Theta) \times max(g(x+1), h(x)) + h(x+1) \\ & = f'(x+1) \end{aligned}$$

$\therefore f(x+1) \leq f'(x+1)$

Incremental Analysis of Logic Programs with Assertions and Open Predicates*

Isabel Garcia-Contreras[1,2], Jose F. Morales[1], and Manuel V. Hermenegildo[1,2]

[1] IMDEA Software Institute
[2] Universidad Politécnica de Madrid (UPM)

Abstract. *Generic* components represent a further abstraction over the concept of modules, which introduces dependencies on other (not necessarily available) components implementing specified interfaces. It has become a key concept in large and complex software applications. Despite its undeniable advantages, generic code is known to be *anti-modular*. Precise analysis (e.g., for detecting bugs or optimizing code) requires such code to be instantiated with concrete implementations, potentially leading to a prohibitively expensive combinatorial explosion. In this paper we claim that incremental (whole program) analysis can be very beneficial in this context, and alleviate the anti-modularity nature of generic code. We propose a simple Horn-clause encoding of generic programs, using *open* predicates and assertions, and we introduce a new *incremental* analysis algorithm that reacts incrementally not only to changes in program clauses, but also to changes in the assertions, upon which large parts of the analysis graph may depend. We also discuss the application of the proposed techniques in a number of practical use cases. In addition, as a realistic case study, we apply the proposed techniques in the analysis of the `LPdoc` documentation system. We argue that the proposed traits are a convenient and elegant abstraction for modular generic programming, and that our preliminary results support the conclusion that the new incrementality-related features added to the analysis bring promising analysis performance advantages.

Keywords: Incremental Static Analysis · Assertions · Logic Programming · Generic Code · Specifications · Abstract Interpretation

1 Introduction

When developing large, real-life programs it is important to ensure application reliability and coding convenience. An important component in order to achieve these goals is the availability in the language (and use in the development process) of some mechanism for expressing specifications, combined with a way of determining if the program meets the specifications or locate errors. This determination is usually achieved through some combination of compile-time analysis and verification with testing and run-time assertion checking.

Another relevant aspect when developing large programs is modularity. In modern coding it is rarely necessary to write everything from scratch. Modules and

* Research partially funded by MINECO TIN2015-67522-C3-1-R *TRACES* project, FPU grant 16/04811, and the Madrid P2018/TCS-4339 *BLOQUES-CM* program. We are also grateful to the anonymous reviewers for their useful comments.

interfaces allow dividing the program in manageable and interchangeable parts. *Interfaces*, including specifications and dependencies, are needed in order to connect with external code (including specifications of such code), to connect self-developed code that is common with other applications (also with specs), and as a placeholder for different implementations of a given functionality, in general referred to as *generic code*.

Despite its undeniable advantages, generic code is known to be in fact *anti-modular*, and the analysis of generic code poses challenges: code is not fully available and interface specifications may not be descriptive enough to verify the specifications for the whole application (e.g., proving termination if the interface specification does not enforce termination properties). Several approaches are possible in order to balance separate compilation with precise analysis and optimization. First, it is possible to analyze generic code by *trusting* its interface specifications, i.e., analyzing the client code and the interface implementations independently, flattening (widening) the analysis information inferred at the boundaries to that of the interface descriptions. This technique can reduce global analysis cost significantly at the expense of some loss of precision. Some of it may be regained by, e.g., enriching specifications manually for the application at hand. At the same time, when considering a closed set of interface implementations, it may also be desirable to analyze the whole application together with these implementations, allowing the transfer of information with "native analysis precision" (e.g., multi-variance) and specializations across the interfaces. This allows getting the most precise information, specializations, compiler optimizations, etc., but at a higher cost.

In this paper we claim that incremental (whole program) analysis can be very beneficial in this context. After providing the necessary notation and background (Section 2), we start by proposing a simple Horn-clause encoding of generic code, using *open* predicates and assertions, and introduce a novel extension for logic programming (*traits*) that is translated using open predicates (Section 3). This abstraction addresses typical use cases of generic code in a more elegant and analysis-friendly way than the traditional alternative in LP of using *multifile* predicates. We also introduce a new *incremental* analysis algorithm (Sections 4 and 5) that, in addition to supporting and taking advantage of assertions during analysis, i.e., as part of the fixpoint calculation, offers two interesting properties: it reacts incrementally not only to changes in program clauses, but also to *changes in the assertions*, upon which large parts of the analysis graph may depend, and it also *supports natively open predicates*.

Generic code offers many opportunities for this new analysis technique. For example: standalone analysis of trait-based code without particular implementations by using the (trust) assertions in the interfaces; refinement of standalone analysis for particular implementations; or reuse of analysis results when more implementations are made available. Note that fine-grained incremental analysis seems even more interesting when using generic code, where the scope of a program change may implicitly be scattered across many modules. We study a number of use cases (Section 5.2), including editing a client (of an interface), while keeping the interface unchanged (e.g., analyzing a program reusing the analysis of a –family of– libraries) and keeping the client code unchanged, but editing the interface implementation(s) (e.g., modifying one implementation of an interface). In addition, in

Section 6 we provide preliminary results from the application of a prototype implementation of the proposed techniques in a realistic case study: the analysis of the `LPdoc` documentation system and its multiple backends for generating documentation in the different formats. Finally, Section 7 discusses some related work and Section 8 presents our conclusions.

2 Background

Logic Programs. A *definite Logic Program*, or *program*, is a finite sequence of clauses. A *clause* is of the form H`:-`$B_1, \ldots, B_n$ where H, the *head*, is an atom, and $B_1, \ldots, B_n$ is the *body*, a possibly empty finite conjunction of atoms. Atoms are also called *literals*. An *atom* is of the form $p(V_1, \ldots, V_n)$. It is *normalized* if the $V_1, \ldots, V_n$ are all distinct variables. Normalized atoms are also called *predicate descriptor*s. Each maximal set of clauses in the program with the same descriptor as head (modulo variable renaming) defines a *predicate* (or *procedure*). We will use P and Q to denote predicates. Body literals can be predicate descriptors, which represent *calls* to the corresponding predicates, or *built-ins*. A *built-in* is a predefined relation for some background theory. Note that built-ins are not necessarily normalized. In the examples we may use non-normalized programs. We denote with $vars(A)$ the set of variables that appear in the atom A.

For presentation purposes, the heads of the clauses of each predicate in the program will be referred to with a unique subscript attached to their predicate name (the clause number), and the literals of their bodies with dual subscript (clause number, body position), e.g., P_k`:-`$P_{k,1}, \ldots P_{k,n_k}$. The clause may also be referred to as clause k of predicate P. For example, for the `append` predicate:

```
1 app(X,Y,Z):- X=[],  Y=Z.
2 app(X,Y,Z):- X=[U|V], Z=[U|W], app(V,Y,W).
```

$\texttt{app}_1$ will denote the head of the first clause of `app/3`, $\texttt{app}_{2,1}$ will denote the first literal of the second clause of `app`, i.e., the unification `X=[U|V]`.

Assertions. Assertions allow stating conditions on the state (current substitution) that hold or must hold at certain points of program execution. We use for concreteness a subset of the syntax of the `pred` assertions of [2, 12, 19], which allow describing sets of *preconditions* and *conditional postconditions* on the state for a given predicate. These assertions are instrumental for many purposes, e.g., expressing the results of analysis, providing specifications, and documenting [9, 12, 20]. A `pred` assertion is of the form:

`:- pred` *Head* `[:` *Pre*`] [=>` *Post*`].`

where *Head* is a predicate descriptor (i.e., a normalized atom) that denotes the predicate that the assertion applies to, and *Pre* and *Post* are conjunctions of *property literals*, i.e., literals corresponding to predicates meeting certain conditions which make them amenable to checking, such as being decidable for any input [19]. *Pre* expresses properties that hold when *Head* is called, namely, at least one *Pre* must hold for each call to *Head*. *Post* states properties that hold if *Head* is called in a state compatible with *Pre* and the call succeeds. Both *Pre* and *Post* can be empty conjunctions (meaning true), and in that case they can be omitted.

Example 1. The following assertions describe different behaviors of an implementation of a hashing function `dgst`: `(1)` states that, when called with argument `Word`

a string and `N` a variable, then, if it succeeds, `N` will be a number, `(2)` states that calls for which `Word` is a string and `N` is an integer are allowed, in other words, it can be used to check if `N` is the hash of `Word`.

```
:- pred dgst(Word,N) : (string(Word), var(N)) => num(N).  % (1)
:- pred dgst(Word,N) : (string(Word), int(N)).            % (2)
dgst(Word,N) :-
% implementation of the hashing function
```

Definition 1 (Meaning of a Set of Assertions for a Predicate). *Given a predicate represented by a normalized atom Head, and a corresponding set of assertions* $\{a_1 \ldots a_n\}$*, with* $a_i =$ "`:- pred` *Head* `:` Pre_i `=>` $Post_i$`.`" *the set of* assertion conditions *for Head is* $\{C_0, C_1, \ldots, C_n\}$*, with:*

$$C_i = \begin{cases} \texttt{calls}(Head, \bigvee_{j=1}^{n} Pre_j) & i = 0 \\ \texttt{success}(Head, Pre_i, Post_i) & i = 1..n \end{cases}$$

where $\texttt{calls}(Head, Pre)$[3] states conditions on all concrete calls to the predicate described by $Head$, and $\texttt{success}(Head, Pre_j, Post_j)$ describes conditions on the success substitutions produced by calls to $Head$ if Pre_j is satisfied.

3 An approach to modular generic programming: *traits*

In this section we present a simple approach to modular generic programming for logic programs without static typing. To that end we introduce the concept of *open* predicates. Then we show how they can be used to deal with generic code, by proposing a simple syntactic extension for logic programs for writing and using generic code (*traits*) and its translation to plain clauses.

Open vs. closed predicates. We consider a simple module system for logic programming where predicates are distributed in modules (each predicate symbol belongs to a particular module) and where module dependencies are explicit in the program [3]. An interesting property, specially for program analysis, is that we can distinguish between *open* and *closed* predicates. [4] Closed predicates within a module are those whose complete definition is available in the module. In contrast, open predicates (traditionally declared as `multifile` in many Prolog systems) are only partially defined within a given module, and different clauses can be scattered across different modules, and thus the complete definition is not known until all the application modules are linked (which is basically "anti-modular").

Open as "multifile." The following example shows an implementation of a generic password-checking algorithm in Prolog:

```
:- multifile dgst/3.

check_passwd(User) :-
    get_line(Plain),                  % Read plain text password
    passwd(User,Hasher,Digest,Salt), % Consult password database
    append(Plain,Salt,Salted),        % Append salt
    dgst(Hasher,Salted,Digest).       % Compute and check digest
```

[3] We denote the calling conditions with `calls` (plural) for historic reasons, and to avoid confusion with the higher order predicate in Prolog `call/2`.

[4] We only consider *static* predicates and modules. Dynamic predicates whose definition may change during execution, or modules that are dynamically changed (loaded/unloaded) at execution time can also be dealt with, using various techniques, and in particular the incremental analysis proposed, but for space reasons we limit the discussion to static predicates.

The code above is generic w.r.t. the selected hashing algorithm (`Hasher`). Note that there is no explicit dependency between `check_passwd/1` and the different hashing algorithms. The special *multifile* predicate `dgst/3` acts as an *interface* between implementations of hashing algorithms and `check_passwd/1`. While this type of encoding is widely used in practice, the use of multifile predicates is semantically obscure and error-prone. We propose a syntactic extension for defining interfaces, or *traits*, in logic programs, which captures the essential mechanisms necessary for writing generic code, but does not require the introduction of a static type system (beyond the *typing* that modules and their interfaces already represents).

Traits. A *trait* is defined as a collection of predicate specifications (as predicate assertions). For example:

```
:- trait hasher {
    :- pred dgst(Str, Digest) : string(Str) => int(Digest).
}.
```

defines a trait `hasher`, which specifies a predicate `dgst/2`, which must be called with an instantiated string, and obtains an integer in `Digest`.

As a minimalistic syntactic extension, we introduce a new head and literal notation $(X \texttt{ as } T).\texttt{p}(A_1, \ldots, A_n)$, which represents the predicate p for X implementing trait T. Basically, this is equivalent to $p(X, A_1, \ldots, A_n)$, where X is used to select the trait implementation. The `check_passwd/1` predicate using the trait above is:

```
check_passwd(User) :-
    get_line(Plain),
    passwd(User,Hasher,Digest,Salt),
    append(Plain,Salt,Salted),
    (Hasher as hasher).dgst(Salted,Digest).
```

The following translation rules convert code using traits to plain predicates. Note that we rely on the underlying module system to add module qualification to function and trait (predicate) symbols. Calls to trait predicates are done through the interface (open) predicate, which also carries the predicate assertions declared in the trait definition:

```
% open predicates and assertions for each p/n in trait
:- multifile 'T.p'/(n + 1).
:- pred 'T.p'(X, A_1, ..., A_n) : ... => ... .
% call to p/n for X implementing T
... :- ..., 'T.p'(X, A_1, ..., A_n), ... % (X as T).p(A_1, ..., A_n)
```

A trait *implementation* is a collection of predicates that implements a given trait, indexed by a specified functor associated with that implementation. E.g.:

```
:- impl(hasher, xor8/0).
(xor8 as hasher).dgst(Str, Digest) :- xor8_dgst(Xs, 0, Digest).

xor8_dgst([], D, D).
xor8_dgst([X|Xs], D0, D) :- D1 is D0 # X, xor8_dgst(Xs, D1, D).
```

declares that `xor8` (an atom in this case, although trait syntax allows arbitrary functors) implements a `hasher`, and provides an implementation for the `dgst/2` predicate (head `(xor8 as hasher).dgst(Str, Digest)`).

The translation rules to plain predicates are as follows:

```
% implementation closed predicate (head renamed)
'<f/k as T>.p'(f(...), A1, ..., An) :- ... % (f(...) as T).p(A1, ..., An)

% bridge from interface open predicate to implementation
'T.p'(X, A1, ..., An) :- X=f(...), '<f/k as T>.p'(X, A1, ..., An).
```

Adding new implementations is simple:

```
:- impl(hasher, sha256/0).
(sha256 as hasher).dgst(Str, Digest) :- ...
```

This approach still preserves some interesting modular features: trait names can be local to a module (and exported as other predicate/function symbols), and trait implementations (e.g., `sha256/0`) are just function symbols, which can also be made local to modules in the underlying module system.

4 Goal-dependent abstract interpretation

After introducing our generic code abstraction, we now describe our proposed incremental analysis to support generic code evolution (i.e., code with traits). We start by recalling in this section the base goal-dependent abstract interpretation algorithm, and in Section 5 we describe the proposed incremental version.

4.1 Preliminaries

Program Analysis with Abstract Interpretation. Our approach is based on *abstract interpretation* [5], a technique in which execution of the program is simulated (over-approximated) on an *abstract domain* (D_α) which is simpler than the actual, *concrete domain* (D). Although not strictly required, we assume that D_α has a lattice structure with meet ($\sqcap$), join ($\sqcup$), and less than ($\sqsubseteq$) operators. Abstract values and sets of concrete values are related via a pair of monotonic mappings $\langle\alpha, \gamma\rangle$: *abstraction* $\alpha : D \to D_\alpha$, and *concretization* $\gamma : D_\alpha \to D$, which form a Galois connection. A description (or abstract value) $d \in D_\alpha$ *approximates* a concrete value $c \in D$ if $\alpha(c) \sqsubseteq d$ where $\sqsubseteq$ is the partial ordering on D_α. Concrete operations on D values are (over-)approximated by corresponding abstract operations on D_α values.

Concrete Semantics. We use top-down, left-to-right SLD-resolution, which, given a *query* (*initial state*), returns the answers (*exit states*) computed for it by the program. A *query* is a pair $\langle G, \theta\rangle$ with G an atom and θ a substitution over the variables of G. Executing (answering) a query with respect to a logic program consists on determining whether the query is a logical consequence of the program and for which substitutions (answers). However, since we are interested in abstracting the calls and answers (states) that occur at different points in the program, we base our semantics on the well-known notion of generalized AND trees [1]. The concrete semantics of a program P for a given set of queries $\mathscr{Q}$, $[\![P]\!]_{\mathscr{Q}}$, is then the set of generalized AND trees that results from the execution of the queries in $\mathscr{Q}$ for P. Each node $\langle G, \theta^c, \theta^s\rangle$ in the generalized AND tree represents a call to a predicate G (an atom), with the substitution (state) for that call, θ^c, and the corresponding success substitution θ^s (answer). A *renaming* substitution, i.e., a substitution that replaces each variable in the term it is applied to with distinct, fresh variables. We use $\sigma(X)$ to denote the application of σ to X.

Graphs and paths. We denote by $G = (V, E)$ a finite *directed graph* (henceforward called simply a graph) where V is a set of nodes and $E \subseteq V \times V$ is an edge relation, denoted with $u \to v$. A *path* P is a sequence of edges $(e_1, \ldots, e_n)$ and each $e_i = (x_i, y_i)$ is such that $x_1 = u$, $y_n = v$, and for all $1 \leq i \leq n-1$ we have $y_i = x_{i+1}$, we also denote paths with $u \rightsquigarrow v \in G$. We use the notation $x \in P$ to denote that a node n appears in P, and $e \in P$ to denote that an edge e appears in P.

4.2 Goal-dependent abstract interpretation.

We perform goal-dependent abstract interpretation, whose result is an abstraction of the generalized AND tree semantics. This technique derives an analysis result from a program P, an abstract domain D_α and a set of initial abstract queries $\mathscr{Q} = \{\langle A_i, {\lambda^c}_i \rangle\}$, where A_i is a normalized atom, and ${\lambda^c}_i \in D_\alpha$. An *analysis result* encodes an abstraction of the nodes of the generalized AND trees derived from all the queries $\langle G, \theta \rangle$ s.t. $\langle G, \lambda \rangle \in \mathscr{Q} \wedge \theta \in \gamma(\lambda)$.

Analysis graphs. We will use graphs to overapproximate all possible executions of a program given an initial query. Each node in our graph is identified by a pair (P, λ) with P a predicate descriptor and $\lambda \in D_\alpha$, an element of the abstract domain, which represents that (a possibly infinite set of) calls to a predicate may occur. The analysis result defines a mapping function $ans : Pred \times D_\alpha \to D_\alpha$, denoted with $\langle P, \lambda^c \rangle \mapsto \lambda^s$ which over-approximates the answer to that abstract predicate call. It is interpreted as *calls to predicate P with calling pattern λ^c have the answer pattern λ^s* with $\lambda^c, \lambda^s \in D_\alpha$. For a given predicate P, the analysis graph can contain a number of nodes capturing different call situations. As usual, $\bot$ denotes the abstract description such that $\gamma(\bot) = \emptyset$. A call mapped to $\bot$ ($\langle A, \lambda^c \rangle \mapsto \bot$) indicates that calls to predicate A with description $\theta \in \gamma(\lambda^c)$ either fail or loop, i.e., they never succeed.

Each edge $\langle P, \lambda_1 \rangle_{c,l} \xrightarrow[\lambda^r]{\lambda^p} \langle Q, \lambda_2 \rangle$ in the graph represents a call dependency among two predicates. It represents that *calling predicate P with calling pattern λ_1 causes predicate Q (literal l of clause c) to be called with calling pattern λ_2*. It is annotated with the abstract element representing the context of the call (λ^p) and the return (λ^r) in the body of clause c and literal l of predicate P. Note that these values are introduced to ease the presentation of the algorithm, however they can be reconstructed with the identifiers of the nodes (i.e., predicate descriptor and abstract value) and the source code of the program. For simplicity, we may write $\bullet$ to omit the values that are not relevant for the operations that we are considering. Note also that if we have the edges that represent the calls to a literal l and the following one $l+1$, $\langle P, \lambda_1 \rangle_{c,l} \xrightarrow[\lambda]{\bullet} \langle Q, \lambda_2 \rangle$ the result at the return of the literal is the call substitution of the next literal: $\langle P, \lambda_1 \rangle_{c,l+1} \xrightarrow[\bullet]{\lambda} \langle Q', \lambda_2' \rangle$. Fig. 1 shows a possible analysis graph for a program that checks/computes the parity of a message.

The following operations defined over an analysis result g allow us to inspect and manipulate analysis results to partially reuse or invalidate.

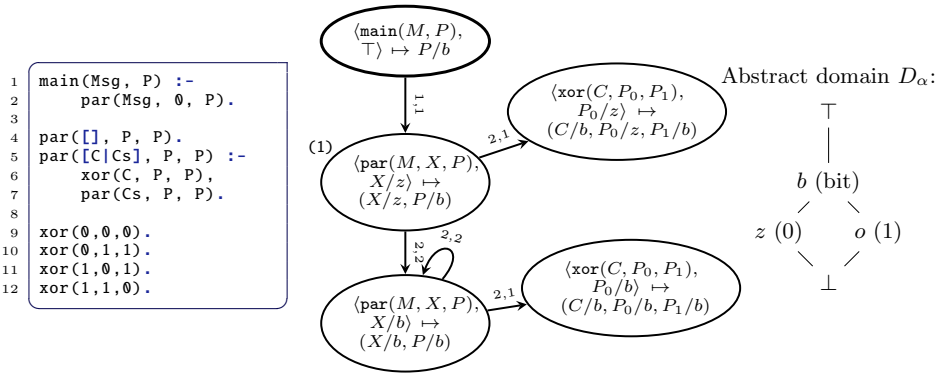


Fig. 1. A program that implements a parity function and a possible analysis result for domain D_α.

Graph consultation operations

$\langle P, \lambda^c \rangle \in g$: there is a node in the call graph of g with key $\langle P, \lambda^c \rangle$.

$\langle P, \lambda^c \rangle \mapsto \lambda^s \in g$: there is a node in g with key $\langle P, \lambda^c \rangle$ and the answer mapped to that call is λ^s.

$\langle P, \lambda^c \rangle_{c,l} \xrightarrow[\lambda^r]{\lambda^p} \langle Q, \lambda^{c\prime} \rangle \in g$: there are two nodes ($k = \langle P, \lambda^c \rangle$ and $k' = \langle Q, \lambda^{c\prime} \rangle$) in g and there is an annotated edge from k to k'.

Graph update operations

$\mathsf{add}(g, \{k_{c,l} \xrightarrow[\lambda^p]{\lambda^r} k'\})$: adds an edge from node k to k' (creating node k' if necessary) annotated with λ^p and λ^r for clause c and literal l.

$\mathsf{del}(g, \{k_{c,l} \xrightarrow[\bullet]{\bullet} k'\})$: removes the edge from node k to k' annotated for clause c and literal l.

The influence of assertions on the analysis result. As described earlier, assertions guarantee that incorrect or undesired behaviors of a program do not occur in the actual execution. We can take advantage of this to prune from the analysis result the states that will never be reached.

We first mention some properties of the analysis graph when no assertions are present in the program: (1) in any edge of the graph A, $\langle P, \lambda_1 \rangle_{i,j} \xrightarrow{\lambda^p} \langle Q, \lambda_2 \rangle$, the abstract substitution immediately before calling the literal (λ^p) is the same as the call to the predicate (λ_2), when projected to the variables of the literal (modulo renaming): $\lambda_2 = \sigma(\texttt{abs_project}(\lambda^p, \mathit{vars}(P_{i,j}))) \wedge Q = \sigma(P)$, and (2) the answer pattern λ^s to a call to a predicate P with λ^c ($\langle P, \lambda^c \rangle \mapsto \lambda^s \in \mathscr{A}$) is the abstract union of the answer of each of its clauses, i.e., the (abstract) state after the call (λ^r) to the last literal in the body ($P_{n,last}$):
$\lambda^s = \bigsqcup\{\texttt{abs_project}(\lambda^r, \mathit{vars}(P_n))$ s.t. $\langle P, \lambda^c \rangle_{n,last} \xrightarrow[\lambda^r]{} N \in \mathscr{A}\}$, with the variables restricted (projected) to the variables in the head of the clause P_n.

However, these properties do not hold when assertions are introduced in a program. (1) does not hold in general because if predicate Q has assertions that cause the analyzer to prune some over-approximated states then, it is not guaranteed to be exactly the same but $\lambda_2 \sqsubseteq \sigma(\texttt{abs_project}(\lambda^p, \mathit{vars}(P_{i,j}))) \wedge Q = \sigma(P)$. (2)

does not hold for the same reason (analysis result may be pruned), and the answer pattern λ^s could be smaller than the abstract join of its clauses.

5 Incremental analysis of programs with assertions

In this section we propose an analysis algorithm that responds incrementally to changes both in the clauses and the assertions of the program. We do so by taking advantage of previous analysis algorithms.

Baseline incremental analysis algorithm. The concepts of analyzing a program incrementally and analyzing a program using assertions are not new. We want to take advantage of the existing algorithms to design an analyzer that is sensible to changes in assertions also. We will use as a black box the straightforward combination of the algorithms to analyze incrementally a CHC program [13], and the analyzer that is guided by assertions [8]. This combined algorithm is detailed in Appendix A, and not included in the paper for space constraints. We will refer to it with the function $\mathscr{A}' = \textsc{IncAnalyze}(P, \Delta_{Cls}, \mathscr{Q}, \mathscr{A})$, meaning that the algorithm takes as input:

- A program $P = (Cls, As)$ as a pair of a set of clauses (Cls) and a set of assertions (As).
- A set of changes Δ_{Cls} in the form of added or deleted clauses
- A set $\mathscr{Q}$ of initial queries that will be the starting point of the analyzer.
- A previous result of the algorithm $\mathscr{A}$ which is a well formed analysis graph.

The algorithm produces a new $\mathscr{A}'$ that correctly abstract the behavior of the program reacting incrementally to changes in the clauses.

The algorithm is parametric on the abstract domain D_α, given by implementing (1) the domain-dependent operations $\sqsubseteq, \sqcap, \sqcup$, `abs_project`($\lambda$, Vs), which restricts the abstract substitution to the set of variables Vs, and `abs_extend`($P_{k,n}, \lambda^p, \lambda^s$), which propagates the information of the success abstract substitution over the variables of $P_{k,n}$, λ^s, to the substitution of the variables of the clause λ^p; and (2) *transfer functions* for program built-ins, that abstract the meaning of the basic operations of the language. These operations are assumed to be monotonic and to correctly over-approximate their correspondent concrete version.

In addition, we assume to have two functions `apply_call`(P, λ^c) and `apply_succ`(P, λ^c, λ^s) that are the transfer functions of the semantics of the assertion conditions (resp. calls and success conditions), i.e., they correctly abstract and apply the assertions to a given predicate and call description. Further details of these functions are described in Appendix A and in [8].

5.1 The incremental analyzer of programs with assertions

We extend IncAnalyze with an initial phase that will manipulate the analysis graph in a way such that we are able to call IncAnalyze to obtain results that are correct and precise, reacting incrementally to changes in assertions. This procedure is shown in Fig. 2. The phase prior to analyzing consists in inspecting all the program points affected by the changes in the assertions, collecting which call patterns need to be reanalyzed by the incremental analysis, i.e., it may be different

```
function IncAnalyze-w/AssrtChanges((Cls, As), Δ_Cls, Δ_As, 𝒬, 𝒜)
    R := ∅
    for each P ∈ Cls do
        if Δ_As[P] ≠ ∅ then
            R := R ∪ update_calls_pred(P)
            R := R ∪ update_success_pred(P)
    𝒜' := IncAnalyze((Cls, As), Δ_Cls, 𝒬 ∪ R, 𝒜)
    del (𝒜', {E | E ∈ 𝒜' ∧ Q ↛ E ∧ Q ∈ 𝒬})          ▷ Remove unreachable calls
    return 𝒜'
```

Fig. 2. High-level view of the proposed algorithm

from the set of initial queries $\mathscr{Q}$ originally requested by the user. In addition, after the analysis phase, the unreachable abstract calls that were safe to reuse may not be reachable anymore, so they need to be removed from the analysis result.

Detecting affected parts in the analysis results. The pseudocode to find potential changes in the analysis results when assertions are changed is detailed in Fig. 3 with procedures `update_calls_pred` and `update_successes_pred`. The goal is to identify which edges and nodes of the analysis graph are not precise or correct. Since, as mentioned in section 4.2, assertions may affect the inferred call or the inferred success of predicates we have split the procedure in two functions. However the overall idea is to obtain the current substitution (i.e., which encodes the semantics of the assertions in the previous version of the program), and the abstract substitution that would have been inferred if no assertions were present. Then we get the semantics of the new assertions (using the functions `apply_call` and `apply_success`), finally we call a general procedure to treat the potential changes, `treat_change` (see Fig. 4). Specifically, in the case of `call` conditions, for a given predicate we want to review all the program points from which it is called (by checking the incoming edges of the nodes of that predicate). So, for each node we project the substitution of the clause (λ^p) to the variables of the literal to obtain the call patterns if no assertions would be specified (line 4). We then detect if the call pattern produced by the new semantics of the assertions already existed in the analysis graph to reuse its result, and last, we call the procedure to treat the change. In the case of success conditions we obtain the substitution if no assertions were present by joining the return substitution at the last literal of each of the clauses of the predicate, previously projected to the variables of the head (line 16).

Amending the analysis results. Qualifying the changes in the abstract substitutions (i.e., they remain the same, they become more general, they become more concrete, or they become incompatible) becomes handy to take advantage of two key known facts about the concrete semantics of logic programs:

- Further constraining a substitution cannot cause more answers to appear.
- Generalizing a substitution cannot cause solutions to disappear.

Based on this knowledge, we define the procedure `treat_change` in Fig. 4. The goal is, given an edge that points to a literal whose success potentially changed,

1: **function** `update_calls_pred`(P)
2: $Q := \emptyset$
3: **for each** $\langle P', \lambda\rangle_{c,l} \xrightarrow{\lambda^p} \langle P, \lambda^c_{old}\rangle \in \mathscr{A}$ **do**
4: $\lambda^c := \sigma(\texttt{abs_project}(\lambda^p, vars(P'_{c,l}))$ s.t. $\sigma(P'_{c,l}) = P$ ▷ Original call
5: $\lambda^c_{new} := \texttt{apply_call}(P, \lambda^c)$
6: **if** $\exists \langle P', {\lambda^c}_{new}\rangle \mapsto \lambda^s \in \mathscr{A}$ **then** ▷ A node for that call already exist
7: $\lambda^{s\prime} := \lambda^s$
8: **else** $\lambda^{s\prime} := \bot$
9: $Q := Q \cup \texttt{treat_change}(\langle P', \lambda\rangle_{c,l} \xrightarrow[\lambda^r]{\lambda^p} \langle P, {\lambda^c}_{new}\rangle, \lambda^{s\prime})$
10: **return** Q
11: **function** `update_successes_pred`(P)
12: $Q := \emptyset$
13: **for each** $\langle P, \lambda^c\rangle \mapsto \lambda^s \in \mathscr{A}$ **do**
14: $\lambda := \bot$
15: **for each** $\langle P, \lambda^c\rangle_{c,last} \xrightarrow[\lambda^r]{} \langle Q, \lambda\rangle \in \mathscr{A}$ **do** ▷ Original success
16: $\lambda := \lambda \sqcup \texttt{apply_success}(P, \lambda^c, \texttt{abs_project}(\lambda^r, vars(P_c)))$
17: **for each** $E = N_{\bullet,\bullet} \xrightarrow[\bullet]{\bullet} \langle P, \lambda^c\rangle \in \mathscr{A}$ **do** ▷ Affected literals
18: $Q := Q \cup \texttt{treat_change}\ (E, \lambda)$
19: **return** Q

Fig. 3. Changes in assertions (split by assertion conditions)

update the analysis result, and decide which predicates and call patterns need to be recomputed. After an initial step to update the annotation of the edge (line 3), we study how the abstract substitution changed. If the new substitution ($\lambda^{r\prime}$) is more general than the previous one (λ^r), this means that the previous assertions where pruning more concrete states than the new one, and, thus, this call pattern needs to be reanalyzed. Else, if $\lambda^r \not\sqsubseteq \lambda^{r\prime}$, i.e., the new abstract substitution is more concrete or incompatible, some parts of the analysis graph may not be accurate. Therefore, we have to eliminate from the graph the literals that were affected by the change (i.e., the literals following the program point with a change) and all the dependent code from this call pattern. Also, the analysis have to be restarted from the original entry points that were affected by the deletion of these potentially imprecise nodes. The case that remains (line 13) is the case in which the old and the new substitutions are the same, and, thus, nothing needs to be reanalyzed (the $\emptyset$ is returned).

5.2 Use cases

We now show some examples in which recomputing is avoided by reasoning with the changes between versions of a program. We assume that we analyze with a shape domain in which the properties that appear in the assertions can be exactly represented.

Example 2 (Reusing a preanalyzed generic program). Consider a slightly modified version the program that checks a password as shown earlier, that only allows the user to write passwords with *lowercase* letters. Until we have a concrete implementation for the hasher we will not be able to analyze precisely this program.

function treat_change($\langle P, \lambda\rangle_{c,l} \xrightarrow[\lambda^r]{\lambda^p} \langle Q, \lambda^c\rangle, \lambda^s$)
  $\lambda^{r\prime} :=$ abs_extend(λ^p, λ^s) ▷ Obtain new info at literal return
  del($\mathscr{A}, \langle P, \lambda\rangle_{c,l} \xrightarrow[\bullet]{\bullet} \bullet$)
  add($\mathscr{A}, \langle P, \lambda\rangle_{c,l} \xrightarrow[\lambda^{r\prime}]{\lambda^p} \langle Q, \lambda^c\rangle$)
  if $\lambda^r \sqsubset \lambda^{r\prime}$ **then**
    return $\{\langle P, \lambda\rangle\}$ ▷ Restart the analysis for this predicate and call pattern
  else if $\lambda^r \not\sqsubseteq \lambda^{r\prime}$ **then** ▷ Analysis is potentially imprecise
    $Lits := \{E \mid E = \langle P, \lambda\rangle_{c,i} \longrightarrow N \in \mathscr{A} \wedge i > l\}$ ▷ Following literals
    $IN := \{E \mid E \rightsquigarrow L \in \mathscr{A} \wedge L \in Lits\}$ ▷ Potentially imprecise nodes
    $Q = IN \cap \mathscr{Q}$ ▷ Entry point of potentially imprecise nodes
    del($\mathscr{A}, IN$)
    return Q
  else return $\emptyset$

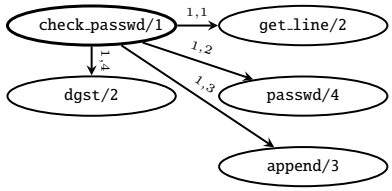
Fig. 4. Functions to determine how the analysis result needs to be recomputed.

However, we can preanalyze it by using the information of the assertion of the trait to obtain the following simplified analysis graph:

```
:- trait hasher { :- pred dgst(Str, Digest)
        : lowercase(Str) => int(Digest). }.

check_passwd(User) :-
    get_line(Plain),
    passwd(User,Hasher,Digest,Salt),
    append(Plain,Salt,Salted),
    (Hasher as hasher).dgst(Salted,Digest).

passwd(don,xor8,0x6d,"eNfwuBhtN9CUHxg==").
```

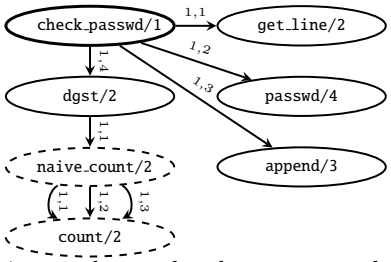


Concretely the node for `dgst/2` will represent the call $\langle$`dgst(S,D)`, $(S/$`lowercase`$, D/$`num`$)\rangle \mapsto (S/$`lowercase`$, D/$`int`$)$, in this case, `D` was inferred to be a *number* because of the success of `passwd/4`.

If we add a very naive implementation that consists on counting the number of some letters in the password, reanalyzing will cause adding to the graph some new nodes, shown with a dashed line:

```
:- impl(hasher, naive/0).
(naive as hasher).dgst(Str, Digest) :-
    naive_count(Xs, 0, Digest).

naive_count(L, D0, D) :-
    count(L,'a',Na), D1 is D0 + Na*1,
    count(L,'b',Nb), D2 is D1 + Nb*2,
    count(L,'c',Nc), D3 is D2 + Nc*3,
    %% implementation continues
```



We are able to detect that none of the previous nodes need to be recomputed due to tracking dependencies for each literal. The analysis was performed by going directly to the program point of `dgst/2` and inspecting the new clause (that was generated automatically by the translation) that calls `naive_count/2`. By analyzing `naive_count/2` we obtain nodes
$\langle$`naive_count(S,D)`, $(S/$`lowercase`$, D/$`num`$)\rangle \mapsto (S/$`lowercase`$, D/$`int`$)$, and

$\langle$count(L, C, N), $(S$/lowercase, C/char)$\rangle$ $\mapsto$ (S/lowercase, C/char, N/int). As no information needs to be propagated because the head does not contain any of the variables of the call to digest, we are done, and we avoid reanalyzing any caller to check_passwd/2, if existed.

Example 3 (Weakening assertion properties). Consider the program and analysis result of Example 2. We realize that allowing the user to write a password only with lowercase letters is not very secure. We can change the assertion of the trait to allow any string as a valid password.

```
1 :- trait hasher {
2       :- pred dgst(Str, Digest) : string(Str) => int(Digest). }.
```

When reanalyzing, node $\langle$dgst(S, D), (S/lowercase, D/num)$\rangle$ will disappear to become $\langle$dgst(S, D), (S/string, D/num)$\rangle$, and the same for naive_count/3. A new call pattern will appear for count/3 $\langle$count(L, C, N), (S/string, C/char)$\rangle$ $\mapsto$ (S/string, C/char, N/int), leading to the same result for dgst/2. I.e., we only had to partially analyze the library, instead of the whole program.

6 Experiments

We have implemented the proposed analysis algorithm within the CiaoPP system [9] and performed some preliminary experiments to test the use case described in Example 2. Our test case is the LPdoc documentation generator tool [10, 11], which takes a set of Prolog files with assertions and machine-readable comments and generates a reference manual from them. LPdoc consists of around 150 files, of mostly (Ciao) Prolog code,with assertions (most of which, when written, were only meant for documentation generation), as well as some auxiliary scripts in Lisp, JavaScript, bash, etc. The Prolog code analyzed is about 22K lines. This is a tool in everyday use that generates for example all the manuals and web sites for the Ciao system (http://ciao-lang.org, http://ciao-lang.org/documentation.html) and as well as for all the different *bundles* developed internal or externally, processing around 20K files and around 1M lines of Prolog and interfaces to another 1M lines of C and other miscellaneous code). The LPdoc code has also been adapted as the documentation generator for the XSB system [21].

LPdoc is specially relevant in our context because it includes a number of backends in order to generate the documentation in different formats such as texinfo, Unix man format, html, ascii, etc. The front end of the tool generates a documentation tree with all the content and formatting information and this is passed to one out of a number of these backends, which then does the actual, specialized generation in the corresponding typesetting language. We analyzed all the LPdoc code with a simple groundness domain (gr), and a domain tracking dependencies via propositional clauses [7] (def). The experiment consisted on preanalyzing the tool with no backends and then adding incrementally the backends one by one. In Table 6 we show how much time it took to analyze in each setting, i.e., for the different domains and with the incremental algorithm or analyzing from scratch. The experiments were run on a MacBook Pro with an Intel Core i5 2.7 GHz processor, 8GB of RAM, and an SSD disk. These preliminary results support our hypothesis that the proposed incremental analysis brings performance advantages when dealing with these use cases of generic code.

domain	no backend	texinfo	man	html
`gr`	3.3	4.4	5.3	6.0
`gr inc`	3.5	1.8	1.3	2.7
`def`	12.2	15.2	15.7	20.8
`def inc`	12.3	2.1	1.3	2.9

Table 1. Analysis time for `LPdoc` adding one backend at a time (time in seconds).

7 Related work

Languages like C++ require specializing all parametric polymorphic code (e.g., templates [22]) to monomorphic variants. While this is more restrictive than *run-time polymorphism* (variants must be statically known at compile time), it solves the analysis precision problem, but not without additional costs. First, it is known to be slow, as templates must be instantiated, reanalized, and recompiled for each compilation unit. Second, it produces many duplicates which must be removed later by the linker. Rust [15] takes a similar approach for *unboxed* types.

Runtime polymorphism or dynamic dispatch can be used in C++ (virtual methods), Rust (boxed traits), Go [6] (interfaces), or Haskell's[14] type classes. However, in this case compilers and analyzers do not usually consider the particular instances, except when a single one can be deduced (e.g., in C++ devirtualization [17]).

Mora et al. [16] perform modular symbolic execution to prove that some (versions of) libraries are equivalent with respect to the same client. Chatterjee et al. [4] analyze libraries in the presence of callbacks incrementally for data dependence analysis. I.e., they preanalyze the libraries and when a client uses it reuses the analysis and adds incrementally possible calls made by the client. We argue that when using our Horn clause encoding, both high analysis precision and compiler optimizations can be achieved more generally by combining the incremental static global analysis that we have proposed with abstract specialization [18].

8 Conclusions

While logic programming can intrinsically handle generic programming, we have illustrated a number of problems that appear when handling generic code with the standard solutions provided by current (C)LP module systems, namely, using multifile predicates. We argue that the proposed traits are a convenient and elegant abstraction for modular generic programming, and that our preliminary results support the conclusion that the novel incremental analysis proposed brings promising analysis performance advantages for this type of code. Our encoding is very close to the underlying mechanisms used in other languages for implementing dynamic dispatch or run-time polymorphism (like Go's interfaces, Rust's traits, or a limited form of Haskell's type clases), so we believe that our techniques and results can be generalized to other languages. Traits are also related to higher-order code (e.g., a "callable" trait with a single "call" method). We also claim that our work contributes to the specification and analysis of higher-order (LP) code.

References

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

2. Bueno, F., Cabeza, D., Hermenegildo, M.V., Puebla, G.: Global Analysis of Standard Prolog Programs. In: ESOP (1996)
3. Cabeza, D., Hermenegildo, M.V.: A New Module System for Prolog. In: Int'l. Conf. on Computational Logic. LNAI, vol. 1861, pp. 131–148. Springer (July 2000)
4. Chatterjee, K., Choudhary, B., Pavlogiannis, A.: Optimal Dyck reachability for data-dependence and alias analysis. PACMPL **2**(POPL), 30:1–30:30 (2018)
5. Cousot, P., Cousot, R.: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. of POPL'77. pp. 238–252. ACM Press (1977)
6. Donovan, A.A.A., Kernighan, B.W.: The Go Programming Language. Professional Computing, Addison-Wesley (October 2015)
7. Dumortier, V., Janssens, G., Simoens, W., García de la Banda, M.: Combining a Definiteness and a Freeness Abstraction for CLP Languages. In: Workshop on LP Synthesis and Transformation (1993)
8. Garcia-Contreras, I., Morales, J., Hermenegildo, M.V.: Multivariant Assertion-based Guidance in Abstract Interpretation. In: 28th Int'l. Symp. on Logic-based Program Synthesis and Transformation (LOPSTR'18). pp. 184–201. No. 11408 in LNCS, Springer (January 2019)
9. Hermenegildo, M., Puebla, G., Bueno, F., Lopez-Garcia, P.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Comp. Progr. **58**(1–2) (2005)
10. Hermenegildo, M.V.: A Documentation Generator for (C)LP Systems. In: Int'l. Conf. CL 2000. LNAI, vol. 1861, pp. 1345–1361. Springer-Verlag (July 2000)
11. Hermenegildo, M.V., Morales, J.: The LPdoc Documentation Generator. Ref. Manual (v3.0). Tech. rep. (July 2011), available at `http://ciao-lang.org`
12. Hermenegildo, M.V., Puebla, G., Bueno, F.: Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In: The Logic Programming Paradigm, pp. 161–192. Springer (1999)
13. Hermenegildo, M.V., Puebla, G., Marriott, K., Stuckey, P.: Incremental Analysis of Constraint Logic Programs. ACM TOPLAS **22**(2), 187–223 (March 2000)
14. Hudak, P., Peyton-Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzman, M.M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W., Peterson, J.: Report on the Programming Language Haskell. Haskell Special Issue, ACM Sigplan Notices **27**(5), 1–164 (1992)
15. Klabnik, S., Nichols, C.: The Rust Programming Language. No Starch Press, San Francisco, CA, USA (2018)
16. Mora, F., Li, Y., Rubin, J., Chechik, M.: Client-specific equivalence checking. In: 33rd ACM/IEEE Int'l. Conf. on Automated Softw. Eng., ASE. pp. 441–451 (2018)
17. Namolaru, M.: Devirtualization in GCC. In: Proceedings of the GCC Developers' Summit. pp. 125–133 (2006)
18. Puebla, G., Albert, E., Hermenegildo, M.V.: Abstract Interpretation with Specialized Definitions. In: SAS'06. pp. 107–126. No. 4134 in LNCS, Springer (2006)
19. Puebla, G., Bueno, F., Hermenegildo, M.V.: An Assertion Language for Constraint Logic Programs. In: Analysis and Visualization Tools for Constraint Programming, pp. 23–61. No. 1870 in LNCS, Springer-Verlag (2000)
20. Puebla, G., Bueno, F., Hermenegildo, M.V.: Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In: Proc. of LOPSTR'99. pp. 273–292. LNCS 1817, Springer-Verlag (March 2000)
21. Swift, T., Warren, D.: XSB: Extending Prolog with Tabled Logic Programming. TPLP (1-2), 157–187 (2012)
22. Vandevoorde, D., Josuttis, N.M.: C++ Templates. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)

A Full description of the base algorithm

In this section we describe the combination of the incremental analysis algorithm [13] with the algorithm that uses assertions of programs [8].

Additional graph operations. In addition to the operations introduced in the paper we will need some more operations to modify the analysis graph:

$\mathsf{upd}(g, \langle A, \lambda^c \rangle \leftarrow\!\shortmid\, \lambda^s)$: overwrites the value of $\langle A, \lambda^c \rangle$ in the mapping function and, if necessary, adding a node to g with key $\langle A, \lambda^c \rangle$.

$\mathsf{upd}(g, k_{c,l} \xrightarrow[\lambda^r]{\lambda^p} k')$: adds an edge node k to node k' with the corresponding annotation if it did not exist.

$\mathsf{upd}(g, \{e_i\})$: performs the upd operation for each of the elements of the set.

$\mathsf{ancestors}(g, k)$: obtains nodes from which there is a path to k $\{k' | k' \rightsquigarrow k \in g\}$

Algorithm usage. As mentioned in the paper, we refer to the analyzer with the function $\mathscr{A}' = \textsc{IncAnalyze}(P, \Delta_{Cls}, \mathscr{Q}, \mathscr{A})$, which is shown in Fig. 5. It takes as input:

- A program $P = (Cls, As)$ as a pair of a set of clauses (Cls) and a set of assertions (As).
- A set of changes Δ_{Cls} in the form of added or deleted clauses
- A set $\mathscr{Q}$ of initial queries that will be the starting point of the analyzer.
- A previous result of the algorithm $\mathscr{A}$ which is a well formed analysis graph.

And produces a new $\mathscr{A}'$ that correctly abstract the behavior of the program reacting incrementally to changes in the clauses.

Additional domain operation. As mentioned earlier, the algorithm is parametric on the abstract domain (D_α), apart from the operations introduced in the paper we define an additional operation: $\mathtt{abs_call}(\lambda, P, P_k)$ performs the abstract unification of predicate descriptor P with the head of the clause P_k, including in the new substitution abstract values for the variables in the body of clause P_k. This operation includes the necessary variable renamings.

Events. The algorithm is centered around processing tasks triggered by events. There are two kinds of events:

- $newcall(\langle A, \lambda^c \rangle)$ indicates that a new description for atom A has been encountered.
- $arc(D)$ means that recomputation needs to be performed starting at program point (literal) indicated by dependency D.

To add events to the queue we use the function $\mathsf{add_event}(E)$.

Operation of the algorithm. The algorithm starts by adding to the queue *newcall* events for each of the call patterns that need to be recomputed. The `process`($newcall(\langle P, \lambda^c \rangle)$) procedure initiates the processing of the clauses in the definition of predicate P. For each of them an *arc* event is added for the first literal. The initial_guess function returns a guess of the λ^s to $\langle P, \lambda^c \rangle$. If possible, it reuses the results in $\mathscr{A}$, otherwise returns $\bot$. Procedure `reanalyze_updated` propagates the information of new computed answers across the analysis graph by creating *arc* events with the program points from which the analysis has to be restarted. The `process`($arc(\langle P_k, \lambda^c \rangle_{l,c} \xrightarrow{\lambda^p} \langle P, \lambda^c \rangle)$) procedure performs the core of the module analysis. It performs a single step of the left-to-right traversal of a clause body. First of all the semantics of the assertions of predicate P are computed by `apply_call`. Then, if the literal $P_{k,i}$ is a *built-in*, it is added to the abstract description; otherwise, if it is an atom, an edge is added to $\mathscr{A}$ and the λ^s is looked up (a process that includes creating a newcall event for $\langle P, \lambda^c \rangle$ if the answer is not in the analysis graph). The obtained answer is combined with the description λ^p from the program point immediately before $P_{k,i}$ to obtain the description (return) for the program point after $P_{k,i}$. This is either used to generate an *arc* event to process the next literal (if there is one), or otherwise to update the answer of the rule in `insert_answer_info`. This function combines the new answer with the semantics of any applicable assertions (in `apply_succ`), and with the previous answers, and propagates the new answer if needed.

Procedure `add_clauses` add to the queue the tasks to analyze the new clause for each predicates. This information is used to update $\mathscr{A}$ and propagated to the rest of the graph. The computation and propagation of the added rules is done simply by adding *arc* events before starting the processing of the queue.

The `delete_clauses` function selects which information can be kept in order to obtain the most precise semantics of the module, by removing all information in the L which is potentially inaccurate, i.e., the information related to the calls that depend on the deleted rules (`remove_invalid_info`), which gathers all the callers to the set of obsolete $Calls$, and the $\langle P, \lambda^c \rangle$ generated from literals that follow in a clause body any $Calls$, because they were affected by the λ^s.

B Assertions

Assertions may not be exactly represented in the abstract domain used by the analyzer. We recall some definitions (adapted from [20]) which are instrumental to correctly approximate the properties of the assertions during the analysis.

Definition 2 (Set of Calls for which a Property Formula Trivially Succeeds (Trivial Success Set)). *Given a conjunction L of property literals and the definitions for each of these properties in P, we define the* trivial success set *of L in P as:*

$$TS(L,P) = \{\theta | Var(L) \text{ } s.t. \text{ } \exists \theta' \in answers(P, \{\langle L, \theta \rangle\}), \theta \models \theta'\}$$

where $\theta|Var(L)$ above denotes the projection of θ onto the variables of L, and $\models$ denotes that θ' is a more general constraint than θ (entailment). Intuitively, $TS(L,P)$ is the set of constraints θ for which the literal L succeeds without adding new constraints to θ (i.e., without constraining it further). For example, given the following program P:

```
1 list([]).
2 list([_|T]) :- list(T).
```

and $L = list(X)$, both $\theta_1 = \{X = [1,2]\}$ and $\theta_2 = \{X = [1,A]\}$ are in the trivial success set of L in P, since calling $(X = [1,2], list(X))$ returns $X = [1,2]$ and calling $(X = [1,A], list(X))$ returns $X = [1,A]$. However, $\theta_3 = \{X = [1|_]\}$ is not, since a call to $(X = [1|Y], list(X))$ will further constrain the term $[1|Y]$, returning $X = [1|Y], Y = []$. We define abstract counterparts for Def. 2:

Definition 3 (Abstract Trivial Success Subset of a Property Formula). *Under the same conditions of Def. 2, given an abstract domain D_α, $\lambda^-_{TS(L,P)} \in D_\alpha$ is an* abstract trivial success subset *of L in P iff $\gamma(\lambda^-_{TS(L,P)}) \subseteq TS(L,P)$.*

Definition 4 (Abstract Trivial Success Superset of a Property Formula). *Under the same conditions of Def. 3, an abstract constraint $\lambda^+_{TS(L,P)}$ is an* abstract trivial success superset *of L in P iff $\gamma(\lambda^+_{TS(L,P)}) \supseteq TS(L,P)$.*

I.e., $\lambda^-_{TS(L,P)}$ and $\lambda^+_{TS(L,P)}$ are, respectively, safe under- and over-approximations of $TS(L,P)$. These abstractions come useful when the properties expressed in the assertions cannot be represented exactly in the abstract domain.

```
ALGORITHM IncAnalyze
  input (global): (Cls, As), Δ_Cls, 𝒬
  global: 𝒜
 1: for all ⟨P, λ^c⟩ ∈ 𝒬 do
 2:   add_event(newcall(⟨P, λ^c⟩))
 3: if Δ_Cls = (Dels, Adds) ≠ (∅, ∅) then
 4:   delete_clauses(Dels)
 5:   add_clauses(Adds)
 6: analysis_loop()
 7: procedure analysis_loop()
 8:   while E :=next_event() do
 9:     process(E)
10: procedure add_clauses(Cls)
11:   for all P_k :- P_{k,1}, ..., P_{k,n_k} ∈ Cls do
12:     for all ⟨P, λ^c⟩ ↦ λ^s ∈ 𝒜 do
13:       λ^p :=abs_call(λ^c, P, P_k)
14:       λ^c_1 :=abs_project(λ^p, vars(P_{k,1}))
15:       add_event(arc(⟨P, λ^c⟩_{k,1} --λ^p--> ⟨pred(P_{k,1}), λ^c_1⟩))
16: procedure delete_clauses(Cls)
17:   Calls := {⟨P, λ^c⟩ | ⟨P, λ^c⟩ ∈ 𝒜, (P_k :- Body) ∈ Cls}
18:   Ns := ancestors(𝒜, Calls)
19:   del(𝒜, Ns)
20: function lookup_answer(⟨P, λ^c⟩)
21:   if ⟨P, λ^c⟩ ↦ λ^s ∈ 𝒜 then
22:     return λ^s
23:   else
24:     add_event(newcall(⟨P, λ^a⟩))
25:     return ⊥
26: procedure process(newcall(⟨P, λ^c⟩))
27:   for all P_k :- P_{k,1}, ..., P_{k,n_k} ∈ Cls do
28:     λ^p :=abs_call(λ^c, P, P_k)
29:     λ^c_1 :=abs_project(λ^p, vars(P_{k,1}))
30:     add_event(arc(⟨P, λ^c⟩_{k,1} --λ^p--> ⟨pred(P_{k,1}), λ^c_1⟩))
31:   λ^s :=initial_guess(⟨P, λ^c⟩)
32:   if λ^s ≠ ⊥ then
33:     reanalyze_updated(⟨P, λ^c⟩)
34:   upd(𝒜, ⟨P, λ^c⟩ ↤ λ^s)
35: procedure process(arc(⟨P, λ^c_0⟩_{k,i} --λ^p--> ⟨Q, λ^c_1⟩))
36:   λ^a = apply_call(Q, λ^c_1)
37:   if P_{k,i} is a built-in then
38:     λ^s_0 := f^α(P_{k,i}, λ^a)              ▷ Apply transfer function
39:   else λ^s_0 :=lookup_answer(⟨Q, λ^a⟩)
40:   λ^r :=abs_extend(λ^p, λ^s_0)
41:   upd(𝒜, ⟨P, λ^c_0⟩_{k,i} --λ^p / λ^r--> ⟨Q, λ^a⟩)
42:   if λ^r ≠ ⊥ and i ≠ n_k then
43:     λ^c_2 :=abs_project(λ^r, vars(P_{k,i+1}))
44:     add_event(arc(⟨H, λ^c_0⟩_{k,i+1} --λ^r--> ⟨B, λ^c_2⟩))
45:   else if λ^r ≠ ⊥ and i = n_k then
46:     λ^s :=abs_project(λ^r, vars(P_k))
47:     insert_answer_info(⟨P, λ^c_0⟩, λ^s)
48: procedure insert_answer_info(⟨P, λ^c⟩, λ^s)
49:   λ^a := apply_succ(P, λ^c, λ^s)
50:   if ⟨P, λ^c⟩ ↦ λ^s_0 ∈ 𝒜 then
51:     λ^s_1 :=abs_generalize(λ^a, λ^s_0)
52:   else λ^s_1 = ⊥
53:   if λ^s_0 ≠ λ^s_1 then
54:     upd(𝒜, ⟨P, λ^c⟩ ↤ λ^s_1)
55:     reanalyze_updated(⟨P, λ^c⟩)
56: procedure reanalyze_updated(⟨P, λ^c⟩)
57:   for all E := ⟨Q, λ^c_0⟩_{k,i} --λ^p--> ⟨P, λ^c⟩ ∈ 𝒜 do
58:     add_event(arc(E))
```

Fig. 5. The generic context-sensitive, incremental fixpoint algorithm using (not changing) assertion conditions.

global flag: `speed-up`

1: **function** `apply_call`(P, λ^c)
2: **if** $\exists \sigma, \lambda^t = \lambda^+_{TS(\sigma(Pre),P)}$ s.t. $\texttt{calls}(H, Pre) \in C, \sigma(H) = P$ **then**
3: **if** `speed-up` **return** λ^t **else return** $\lambda^c \sqcap \lambda^t$
4: **else return** λ^c
5: **function** `apply_succ`$(P, \lambda^c, \lambda^{s_0})$
6: $app = \{\lambda \mid \exists\ \sigma, \texttt{success}(H, Pre, Post) \in C, \sigma(H) = P,$
7: $\lambda = \lambda^+_{TS(\sigma(Post),P)}, \lambda^-_{TS(\sigma(Pre),P)} \sqsupseteq \lambda^c\}$
8: **if** $app \neq \emptyset$ **then**
9: $\lambda^t := \bigsqcap app$
10: **if** `speed-up` **return** λ^t **else return** $\lambda^t \sqcap \lambda^{s_0}$
11: **else return** λ^{s_0}

Fig. 6. Applying assertions.

Computing Abstract Distances in Logic Programs*

Ignacio Casso[1,2], José F. Morales[1], Pedro López-García[1,3], Roberto Giacobazzi[1,4], and Manuel V. Hermenegildo[1,2]

[1] IMDEA Software Institute
[2] T. University of Madrid (UPM)
[3] Spanish Council for Scientific Research (CSIC)
[4] University of Verona, Italy

Abstract. Abstract interpretation is a well-established technique for performing static analyses of logic programs. However, choosing the abstract domain, widening, fixpoint, etc. that provides the best precision-cost trade-off remains an open problem. This is in a good part because of the challenges involved in measuring and comparing the precision of different analyses. We propose a new approach for measuring such precision, based on defining distances in abstract domains and extending them to distances between whole analyses of a given program, thus allowing comparing precision across different analyses. We survey and extend existing proposals for distances and metrics in lattices or abstract domains, and we propose metrics for some common domains used in logic program analysis, as well as extensions of those metrics to the space of whole program analysis. We implement those metrics within the CiaoPP framework and apply them to measure the precision of different analyses on both benchmarks and a realistic program.

Keywords: Abstract interpretation, static analysis, logic programming, metrics, distances, complete lattices, program semantics.

1 Introduction

Many practical static analyzers for (Constraint) Logic Programming ((C)LP) are based on the theory of Abstract Interpretation [8]. The basic idea behind this technique is to interpret (i.e., execute) the program over a special abstract domain to obtain some abstract semantics of the program, which will over-approximate every possible execution in the standard (concrete) domain. This makes it possible to reason safely (but perhaps imprecisely) about the properties that hold for all such executions. As mentioned before, abstract interpretation has proved practical and effective for building static analysis tools, and

* Research partially funded by MINECO TIN2015-67522-C3-1-R *TRACES* project, and the Madrid P2018/TCS-4339 *BLOQUES-CM* program. We are also grateful to the anonymous reviewers for their useful comments.

in particular in the context of (C)LP [30,21,38,6,12,5,29,16,25]. Recently, these techniques have also been applied successfully to the analysis and verification of other programming paradigms by using (C)LP (Horn Clauses) as the intermediate representation for different compilation levels, ranging from source to bytecode or ISA [1,3,33,17,26,10,19,4,28,24].

When designing or choosing an abstract interpretation-based analysis, a crucial issue is the trade-off between cost and precision, and thus research in new abstract domains, widenings, fixpoints, etc., often requires studying this trade-off. However, while measuring analysis cost is typically relatively straightforward, having effective precision measures is much more involved. There have been a few proposals for this purpose, including, e.g., probabilistic abstract interpretation [13] and some measures in numeric domains [27,37] [5] , but they have limitations and in practice most studies come up with ad-hoc measures for measuring precision. Furthermore, there have been no proposals for such measures in (C)LP domains.

We propose a new approach for measuring the precision of abstract interpretation-based analyses in (C)LP, based on defining *distances in abstract domains* and extending them to *distances between whole analyses of a given program*, which allow comparison of precision across different analyses. Our contributions can be summarized as follows: We survey and extend existing proposals for distances in lattices and abstract domains (Sec. 3). We then build on this theory and ideas to propose distances for common domains used in (C)LP analysis (Sec. 3.2). We also propose a principled methodology for comparing quantitatively the precision of different abstract interpretation-based analyses of a whole program (Sec. 4). This methodology is parametric on the distance in the underlying abstract domain and only relies in a unified representation of those analysis results as AND-OR trees. Thus, it can be used to measure the precision of new fixpoints, widenings, etc. within a given abstract interpretation framework, not requiring knowledge of its implementation. Finally, we also provide experimental evidence about the appropriateness of the proposed distances (Sec. 5).

2 Background and Notation

Lattices: A *partial order* on a set X is a binary relation $\sqsubseteq$ that is reflexive, transitive, and antisymmetric. The *greatest lower bound* or *meet* of a and b, denoted by $a \sqcap b$, is the greatest element in X that is still lower than both of them ($a \sqcap b \sqsubseteq a,\ a \sqcap b \sqsubseteq b,\ (c \sqsubseteq a\ \wedge c \sqsubseteq b\ \implies\ c \sqsubseteq a \sqcap b)$). If it exists, it is unique. The *least upper bound* or *join* of a and b, denoted by $a \sqcup b$, is the smallest element in X that is still greater than both of them ($a \sqsubseteq a \sqcup b, b \sqsubseteq a \sqcup b$, $(a \sqsubseteq c \wedge b \sqsubseteq c \implies a \sqcup b \sqsubseteq c)$). If it exists, it is unique. A partially ordered set (poset) is a couple $(X, \sqsubseteq)$ such that the first element X is a set and the second one is a partial order relation on X. A *lattice* is a poset for which any two elements

[5] Some of these attempts (and others) are further explained in the related work section (Section 6).

have a meet and a join. A lattice L is complete if, extending in the natural way the definition of supremum and infimum to subsets of L, every subset S of L has both a supremum $sup(S)$ and an infimum $inf(S)$. The maximum element of a complete lattice, $\sup(L)$ is called *top* or $\top$, and the minimum, $\inf(L)$ is called *bottom* or $\bot$.

Galois Connections: Let $(L_1, \sqsubseteq_1)$ and $(L_2, \sqsubseteq_2)$ be two posets. Let $f : L_1 \longrightarrow L_2$ and $g : L_2 \longrightarrow L_1$ be two applications such that:

$$\forall x \in L_1, y \in L_2 : f(x) \sqsubseteq_2 y \iff x \sqsubseteq_1 g(y)$$

Then the quadruple $\langle L_1, f, L_2, g\rangle$ is a *Galois connection*, written $L_1 \underset{f}{\overset{g}{\leftrightarrows}} L_2$. If $f \circ g$ is the identity, then the quadruple is called a *Galois insertion*.

Abstract Interpretation and Abstract Domains: Abstract interpretation [8] is a well-known static analysis technique that allows computing sound over-approximations of the semantics of programs. The semantics of a program can be described in terms of the *concrete domain*, whose values in the case of (C)LP are typically sets of variable substitutions that may occur at runtime. The idea behind abstract interpretation is to interpret the program over a special abstract domain, whose values, called *abstract substitutions*, are finite representations of possibly infinite sets of actual substitutions in the concrete domain. We will denote the concrete domain as D, and the abstract domain as D_α. We will denote the functions that relate sets of concrete substitutions with abstract substitutions as the *abstraction* function $\alpha : D \longrightarrow D_\alpha$ and the *concretization* function $\gamma : D_\alpha \longrightarrow D$. The concrete domain is a complete lattice under the set inclusion order, and that order induces an ordering relation in the abstract domain herein represented by "$\sqsubseteq$." Under this relation the abstract domain is usually a complete lattice and $(D, \alpha, D_\alpha, \gamma)$ is a Galois insertion. The abstract domain is of finite height or alternatively it is equipped with a *widening operator*, which allows for skipping over infinite ascending chains during analysis to a greater fixpoint, achieving convergence in exchange for precision.

Metric: A metric on a set S is a function $d : S \times S \to \mathbb{R}$ satisfying:

- Non-negativity: $\forall x, y \in S,\ d(x, y) \geq 0.$
- Identity of indiscernibles: $\forall x, y \in S,\ d(x, y) = 0 \iff x = y.$
- Symmetry: $\forall x, y \in S,\ d(x, y) = d(y, x).$
- Triangle inequality: $\forall x, y, z \in S,\ d(x, z) \leq d(x, y) + d(y, z).$

A set S in which a metric is defined is called a metric space. A pseudometric is a metric where two elements which are different are allowed to have distance 0. We call the left implication of the identity of indiscernibles, weak identity of indiscernibles. A well-known method to extend a metric $d : S \times S \longrightarrow \mathbb{R}$ to a distance in 2^S is using the Hausdorff distance, defined as:

$$\mathrm{d}_H(A, B) = \max\left\{\sup_{a\in A}\inf_{b\in B}\mathrm{d}(a, b), \sup_{b\in B}\inf_{a\in A}\mathrm{d}(a, b)\right\}$$

3 Distances in Abstract Domains

As anticipated in the introduction, our distances between abstract interpretation-based analyses of a program will be parameterized by distance in the underlying abstract domain, which we assume to be a complete lattice. In this section we propose a few such distances for relevant logic programming abstract domains. But first we review and extend some of the concepts that arise when working with lattices or abstract domains as metric spaces.

3.1 Distances in lattices and abstract domains

When defining a distance in a partially ordered set, it is necessary to consider the compatibility between the metric and the structure of the lattice. This relationship will suggest new properties that a metric in a lattice should satisfy. For example, a distance in a lattice should be *order-preserving*, that is, $\forall a, b, c \in D \; with \; a \sqsubseteq b \sqsubseteq c, \; then \; d(a,b), d(b,c) \leq d(a,c)$. It is also reasonable to expect that it fulfills what we have called the diamond inequality, that is, $\forall a, b, c, d \in D \; with \; c \sqcap d \sqsubset a \sqcap b, \; a \sqcup b \sqsubset c \sqcup d, \; then \; d(a,b) \leq d(c,d)$. But more importantly, this relationship will suggest insights for constructing such metrics.

One such insight is precisely defining a partial metric $d_{\sqsubseteq}$ only between elements which are related in the lattice, which is arguably easier, and to extend it later to a distance between arbitrary elements x, y, as a function of $d_{\sqsubseteq}(x, x \sqcap y)$, $d_{\sqsubseteq}(y, x \sqcap y)$, $d_{\sqsubseteq}(x, x \sqcup y)$, $d_{\sqsubseteq}(x, x \sqcup y)$ and $d_{\sqsubseteq}(x \sqcap y, x \sqcup y)$. Jan Ramon et al. [36] show under which circumstances $d_{\sqsubseteq}(x, x \sqcup y) + d_{\sqsubseteq}(y, x \sqcup y)$ is a distance, that is, when $d_{\sqsubseteq}$ is order-preserving and fulfills $d_{\sqsubseteq}(x, x \sqcup y) + d_{\sqsubseteq}(y, x \sqcup y) \leq d_{\sqsubseteq}(x, x \sqcap y) + d_{\sqsubseteq}(y, x \sqcap y)$.

In particular, one could define a monotonic size $size : L \to \mathbb{R}$ in the lattice and define $d_{\sqsubseteq}(a, b)$ as $size(b) - size(a)$. Gratzer [18] shows that if the size fullfills $size(x) + size(y) = size(x \sqcap y) + size(x \sqcup y)$, then $d(x, y) = size(x \sqcup y) - size(x \sqcap y)$ is a metric. De Raedt [11] shows that $d(x, y) = size(x) + size(y) - 2 \cdot size(x \sqcup y)$ is a metric iff $size(x) + size(y) \leq size(x \sqcap y) + size(x \sqcup y)$, and an analogous result with $d(x, y) = size(x) + size(y) - 2 \cdot size(x \sqcup y)$ and $\geq$ instead of $\leq$. Note that the first distance is the equivalent of the *symmetric difference distance* in finite sets, with $\sqsubseteq$ instead of $\subseteq$ and $size$ instead of the cardinal of a set. Similar distances for finite sets, such as the Jaccard distance, can be translated to lattices in the same way. Another approach to defining $d_{\sqsubseteq}$ that follows from the idea of using the lattice structure, is counting the steps between two elements (i.e., the number of edges between both elements in the Hasse diagram of the lattice). This was used by Logozzo [27].

When defining a distance not just in any lattice, but in an actual abstract domain (*abstract distance* from now on), it is also necessary to consider the relation of the abstract domain with the concrete domain (i.e., the Galois connection), and how an abstract distance is interpreted under that relation. In that sense, we can observe that a distance $d_{D_\alpha} : D_\alpha \times D_\alpha \to \mathbb{R}^+$ in an abstract domain will induce a distance $d_D^\alpha : D \times D \to \mathbb{R}^+$ in the concrete one, as $d_D^\alpha(A, B) = d_{D_\alpha}(\alpha(A), \alpha(B))$, and the other way around: a distance $d_D : D \times D \to \mathbb{R}^+$ in

the concrete domain induces an abstract distance $d^{\gamma}_{D_\alpha} : D_\alpha \times D_\alpha \to \mathbb{R}^+$ in the abstract one, as $d^{\gamma}_{D_\alpha}(a, b) = d_D(\gamma(a), \gamma(b))$. Thus, an abstract distance can be interpreted as an abstraction of a distance in the concrete domain, or as a way to define a distance in it, and it is clear that it is when interpreted that way that an abstract distance makes most sense from a program semantics point of view.

It is straightforward to see (and we show in the appendix) that these induced distances inherit most metric and order-related properties. In particular, if a distance d_D in the concrete domain is a metric, its abstraction d_{D_α} is a pseudo-metric in the abstract domain, and a full metric if the Galois connection between D and D_α is a Galois insertion. This allows us to define distances d_α in the abstract domain from distances d the concrete domain, as $d_\alpha(a, b) = d(\gamma(a), \gamma(b))$. This approach might seem of little applicability, due to the fact that concretizations will most likely be infinite and we still need metrics in the concrete domain. But in the case of logic programs, such metrics for Herbrand terms already exist (e.g., [22,34,36]), and in fact we show later a distance for the *regular types* domain that can be interpreted as an extension of this kind, of the distance proposed by Nienhuys-Cheng [34] for sets of terms.

Finally, we note that a metric in the Cartesian product of lattices can be easily derived from existing distances in each lattice, for example as the 2-norm or any other norm of the vector of distances component to component. This is relevant because many abstract domains, such as those that are combinations of two different abstract domains, or non-relational domains which provide an abstract value from a lattice for each variable in the substitution, are of such form. However, although this is a well-known result, it is not clear whether the resulting distance will fulfill other lattice-related properties if the distances for each component do. It is straightforward to see that that is the case for the *order-preserving* property, but not for the *diamond inequality*.

3.2 Distances in Logic Programming Domains

We now propose some distances for two well-known abstract domains used in (C)LP, following the considerations presented in the previous section.

Sharing domain: The `sharing` domain [23,30] is a well-known domain for analyzing the sharing (aliasing) relationships between variables and grounding in logic programs. It is defined as $2^{2^{Pvar}}$, that is, an abstract substitution for a clause is defined to be *a set of sets of program variables* in that clause, where each set indicates that the terms to which those variables are instantiated at runtime might share a free variable. More formally, we define $Occ(\theta, U) = \{X | X \in dom(\theta), U \in vars(X\theta)\}$, the set of all program variables $X \in Pvar$ in the clause such that the variable $U \in Uvar$ appears in $X\theta$. We define the abstraction of a substitution θ as $\mathcal{A}_{sharing}(\theta) = \{Occ(\theta, U) \mid U \in Uvar\}$, and extend it to sets of substitutions. The order induced by this abstraction in $2^{2^{Pvar}}$ is the set inclusion, the join, the set union, and the meet, the set intersection. As an example, $\top = 2^{Pvar}$, a program variable that does not appear in any set is guaranteed to

be ground, or two variables that never appear in the same set are guaranteed to not share. The complete definition can be found in [23,30]).

Following the approach of the previous section, we define this monotone size in the domain: $size(a) = |a| + 1, size(\bot) = 0$. It is straightforward to check that $\forall a, b \in 2^{2^{Pvar}},\ size(a) + size(b) = size(a \sqcap b) + size(a \sqcup b)$. Therefore the following distance is a metric and order-preserving: $d_{share}(Sh_1, Sh_2) =$

$$size(Sh_1 \cup Sh_2) - size(Sh_1 \cap Sh_2) = |(Sh_1 \cup Sh_2)| - |size(Sh_1 \cap Sh_2)|$$

We would like our distance to be in a normalized range $[0, 1]$, and for that we divide it between $d(\bot, \top) = 2^n + 1$, where $n = |V|$ denotes the number of variables in the domain of the substitutions. This yields the following final distance, which is a metric by construction:

$$d_{share}(Sh_1, Sh_2) = (|(Sh_1 \cup Sh_2)| - |size(Sh_1 \cap Sh_2)|)/(2^n + 1)$$

Regular-type domain: Another well-known domain for logic programs is the *regular types* domain [9], which abstracts the shape or type of the terms to which variables are assigned on runtime. It associates each variable with a deterministic context free grammar that describes its shape, with the possible functors and atoms of the program as terminal symbols. A more formal definition can be found in [9]. We will write abstract substitutions as tuples $\langle T_1, \ldots, T_n \rangle$, where $T_i = (S_i, \mathcal{T}_i, \mathcal{F}_i, \mathcal{R}_i)$ is the grammar that describes the term associated to the i-th variable in the substitution. We propose to use as a basis the Hausdorff distance in the concrete domain, using the distance between terms proposed in [34], i.e.,

$$d_{term}(f(x_1, \ldots, x_n), g(y_1, \ldots, y_m)) = \begin{cases} 1 \quad if \quad f/n \neq g/m \\ else: \quad p \sum_{i=1}^{n} \frac{1}{n} d_{term}(x_i, y_i) \end{cases}$$

where p is a paremeter of the distance. As the derived abstract version, we propose the following distance between two types or grammars $S_1,\ S_2$, defined recursively and with a little abuse of notation:

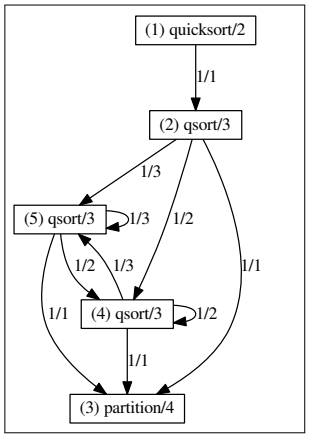
$$d'(S_1, S_2) = \begin{cases} 1\ if\ \exists\ (S_1 \to f(T_1, \ldots, T_n)) \in \mathcal{R}_1 \wedge \nexists (S_2 \to f(T'_1, \ldots, T'_n)) \in \mathcal{R}_2 \\ 1\ if\ \exists\ (S_2 \to f(T_1, \ldots, T_n)) \in \mathcal{R}_2 \wedge \nexists (S_1 \to f(T'_1, \ldots, T'_n)) \in \mathcal{R}_1 \\ else: max\{p \sum_{i=1}^{n} \frac{1}{n} d'(T_i, T'_i) \mid (S_1 \to f(T_1, \ldots, T_n)) \in \mathcal{R}_1 \wedge \\ \qquad\qquad (S_2 \to f(T'_1, \ldots, T'_n)) \in \mathcal{R}_2\} \end{cases}$$

We also extend this distance between types to distance between substitutions in the abstract domain as follows: $d(\langle T_1, \ldots, T_n \rangle, \langle T'_1, \ldots, T'_n \rangle) = \sqrt{d'(T_1, T'_1)^2 + \ldots + d'(T_n, T'_n)^2}$. Since d' is the abstraction of the Hausdorff distance with d_{term}, which it is proved to be a metric in [34], d' is a metric too. [6] Therefore d is also a metric, since it is its extension to the cartesian product.

4 Distances between analyses

We now attempt to extend a distance in an abstract domain to distances between results of different abstract interpretation-based analyses of the same program

[6] Actually that only guarantees that d' is a pseudo-metric. Proving that it is indeed a metric is more involved and not really relevant to our discussion.




```
:-
     module(quicksort,[quicksort/2],[assertions]).
:- use_module(partition,[partition/4]).

:- entry quicksort(Xs,Ys) : (ground(Xs),
     var(Ys)).

quicksort(Xs,Ys) :-
     qsort(Xs,Ys,[]).

qsort([],Ys,Ys).
qsort([X|Xs],Ys,TailYs) :-
     partition(Xs,X,L,R),
     qsort(R,R2,TailYs),
     qsort(L,Ys,[X|R2]).
```

(1) $\langle quicksort(Xs,Ys),\ \{Xs/g,Ys/ng\},\ \{Xs/g,Ys/g\}\rangle$
(2) $\langle qsort(Xs,Ys,[]),\ \{Xs/g,Ys/ng\},\ \{Xs/g,Ys/g\}\rangle$
(3) $\langle partition(Xs,X,L,R),\ \{Xs/g,X/g,L/ng,R/ng\},\ \{Xs/g,X/g,L/g,R/g\}\rangle$
(4) $\langle qsort(Xs,Ys,Zs),\ \{Xs/g,Ys/ng,Zs/g\},\ \{Xs/g,Ys/g,Zs/g\}\rangle$
(5) $\langle qsort(Xs,Ys,[Z|Zs]),\ \{Xs/g,Ys/ng,Z/g,Zs/g\},\ \{Xs/g,Ys/g,Z/g,Zs/g\}\rangle$

Fig. 1. Analysis of `quicksort/2` (using difference lists).

over that domain. In the following we will assume (following most "top-down" analyzers for (C)LP programs [30,5,16,25]) that the result of an analysis for a given entry (i.e., an initial predicate P, and an initial call pattern or abstract query λ_c), is an AND-OR tree, with root the OR-node $\langle P, \lambda_c, \lambda_s\rangle_\vee$, where λ_s is the abstract substitution computed by the analysis for that predicate given that initial call pattern. An AND-OR tree alternates AND-nodes, which correspond to clauses in the program, and OR-nodes, which correspond to literals in those clauses. An OR-node is a triplet $\langle L, \lambda_c, \lambda_s\rangle_\vee$, with L a call to a predicate P and λ_c, λ_s the abstract call and success substitutions for that goal. It has one AND-node $\langle C_j, \beta^j_{entry}, \beta^j_{exit}\rangle_\wedge$ as child for each clause C_j in the definition of P, where $\beta^j_{entry} = \lambda_c\ \forall j$ and $\lambda_s = \bigsqcup \beta^j_{exit}$. An AND-node is a triplet $\langle C, \beta_{entry}, \beta_{exit}\rangle_\wedge$, with C a clause $Head : -L_1, ..., L_n$ and with $\beta_{entry}, \beta_{exit}$ the abstract entry and exit substitutions for that clause. It has an OR-node $\langle L_i, \lambda^i_c, \lambda^i_s\rangle_\vee$ for each literal L_i in the clause, where $\beta_{entry} = \lambda^1_c,\ \lambda^i_s = \lambda^{i+1}_c,\ \lambda^n_s = \beta_{exit}$. This tree is the abstract counterpart of the resolution trees that represent concrete top-down executions, and represents a possibly infinite set of those resolution trees at once. The tree will most likely be infinite, but can be represented as a finite cyclic tree. We denote the children of a node T as $ch(T)$ and its triplet as $val(T)$.

Example 1. Let us consider as an example the simple quick-sort program (using difference lists) in Fig. 1, which uses an *entry* assertion to specify the initial abstract query of the analysis [35]. If we analyze it with a simple *groundness* domain (with just two values `g` and `ng`, plus $\top$ and $\bot$), the result can be represented with the graph shown in Fig. 1. That graph is a finite representation of an infinite abstract and-or tree. The nodes in the graph correspond to or-nodes $\langle L, \lambda^c, \lambda^s\rangle$ in the analysis tree, where the literals L, abstract call substitutions λ^c and abstract success substitutions λ^s are specified below the graph. The la-

bels in the edge indicate to which program point each node corresponds: if one node is connected to its predecessor by an arrow with label i/j, then that node corresponds to the j-th literal of the i-th clause of the predicate indicated by the predecessor. The and-nodes are left implicit. □

We propose three distances between AND-OR trees S_1, S_2 for the same entry, in increasing order of complexity, and parameterized by a distance d_α in the underlying abstract domain. We also discuss which metric properties are inherited by these distances from d_α. Note that a good distance for measuring precision should fulfill the identity of indiscernibles.

Top distance. The first consists in considering only the roots of the top trees, $\langle P, \lambda_c, \lambda_s^1 \rangle_\vee$ and $\langle P, \lambda_c, \lambda_s^2 \rangle_\vee$, and defining our new distance as $d(S_1, S_2) = d_\alpha(\lambda_s^1, \lambda_s^2)$. This distance ignores too much information (e.g., if the entry point is a predicate `main/0`, the distance would only distinguish analyses that detect failure from analysis which do not), so it is not appropriate for measuring analysis precision, but it is still interesting as a baseline. It is straightforward to see that it is a pseudometric if d_α is, but will not fulfill the identity of indiscernibles even if d_α does.

Flat distance. The second distance considers all the information inferred by the analysis for each program point, but forgetting about its context in the AND-OR tree. In fact, analysis information is often used this way, i.e., considering only the substitutions with which a program point can be called or succeeds, and not which traces lead to those calls (path insensitivity). We define a distance between program points

$$d_{PP}(S_1, S_2) = \frac{1}{2}(d_\alpha(\bigsqcup_{\lambda \in PP_c^1} \lambda, \bigsqcup_{\lambda \in PP_c^2} \lambda) + d_\alpha(\bigsqcup_{\lambda \in PP_s^1} \lambda, \bigsqcup_{\lambda \in PP_s^2} \lambda))$$

where $PP_c^i = \{\lambda_c \mid \langle PP, \lambda_c, \lambda_s \rangle_\vee \in S_i\}$, $PP_s^i = \{\lambda_s \mid \langle PP, \lambda_c, \lambda_s \rangle_\vee \in S_i\}$. If we denote P as the set of all program points in the program, that distance can later be extended to a distance between analyses as $d(S_1, S_2) = \frac{1}{|P|} \sum_{PP \in P} d_{PP}(S_1, S_2)$, or any other combination of the distances $d_{PP}(S_1, S_2)$ (e.g, weighted average, $|| \cdot ||_2$). This distance is more appropriate for measuring precision than the previous one, but it will still inherit all metric properties except the identity of indiscernibles. An example of this distance can be found in appendix B.1.

Tree distance. For the third distance, we propose the following recursive definition, which can easily be translated into an algorithm:

$$d(T_1, T_2) = \begin{cases} \mu \frac{1}{2}(d_\alpha(\lambda_c^1, \lambda_c^2) + d_\alpha(\lambda_s^1, \lambda_s^2)) + (1-\mu)\frac{1}{|C|}\sum_{(c_1,c_2)\in C} d(c_1, c_2) & if\ C \neq \emptyset \\ else \quad \frac{1}{2}(d_\alpha(\lambda_c^1, \lambda_c^2) + d_\alpha(\lambda_s^1, \lambda_s^2)) \end{cases}$$

where $T_1 = \langle P, \lambda_c^1, \lambda_s^1 \rangle$, $T_2 = \langle P, \lambda_c^2, \lambda_s^2 \rangle$, $\mu \in (0, 1]$ and $C = \{(c_1, c_2) \mid c_1 \in ch(T_1), c_2 \in ch(T_2), val(c_1) = \langle X, _, _ \rangle, val(c_2) = \langle Y, _, _ \rangle, X = Y\}$.

This definition is possible because the two AND-OR trees will necessarily have the same shape, and therefore we are always comparing a node with its

correspondent node in the other tree. Also, this distance is well defined, even if the trees, and therefore the recursions, are infinite, since the expression above always converges. Furthermore, the distance to which the expression converges can be easily computed in finite time. Since the AND-OR trees always have a finite representation as cyclic trees with n and m nodes respectively, there are at most $n * m$ different pairs of nodes to visit during the recursion. Assigning a variable to each pair that is actually visited, the recursive expression can be expressed as a linear system of equations. That system has a unique solution since the original expression had, but also because there is an equation for each variable and the associated matrix, which is therefore squared, has strictly dominant diagonal. An example can be found in the appendix B.2.

The idea of this distance is that we consider more relevant the distance between the upper nodes than the distance between the deeper ones, but we still consider all of them and do not miss any of the analysis information. As a result, this distance will directly inherit the identity of indiscernibles (apart from all other metric properties) from d_α.

5 Experimental Evaluation

To evaluate the usefulness of the program analysis distances, we set up a practical scenario in which we study quantitatively the cost and precision tradeoff for several abstract domains. In order to do it we need to overcome two technical problems described below.

Base domain. Recall that in the distances defined so far, we assume that we compare two analyses using the same abstract domain. We relax this requirement by translating each analysis to a common *base domain*, rich enough to reflect a particular program property of interest. An abstract substitution λ over a domain D_α is translated to a new domain $D_{\alpha'}$ as $\lambda' = \alpha'(\gamma(\lambda))$, and the AND-OR tree is translated by just translating any abstract substitution occurring in it. The results still over-approximates concrete executions, but this time all over the same abstract domain.

Program analysis intersection. Ideally we would compare each analysis with the actual semantics of a program for a given abstract query, represented also as an AND-OR tree. However, this semantics is undecidable in general, and we are seeking an automated process. Instead, we approximated it as the *intersection* of all the computed analyses. The intersection between two trees, which can be easily generalized to n trees, is defined as $inter(T_1, T_2) = T$, with

$$\begin{aligned} val(T_1) = \langle X, \lambda_c^1, \lambda_s^2 \rangle,\ val(T_2) = \langle X, \lambda_c^2, \lambda_s^2 \rangle,\ val(T) = \langle X, \lambda_c^1 \sqcap \lambda_c^2, \lambda_s^1 \sqcap \lambda_s^2 \rangle \\ ch(T) \quad = \{\ inter(c_1, c_2) \mid c_1 \in ch(T_1),\quad c_2 \in ch(T_2),\quad val(c_1) = \langle X, _, _ \rangle, \\ val(c_2) = \langle Y, _, _ \rangle,\quad X = Y \} \end{aligned}$$

That is, a new AND-OR tree with the same shape as those computed by the analyses, but where each abstract substitution is the greatest lower bound of the corresponding abstract substitutions in the other trees. The resulting tree is

the least general AND-OR tree we can obtain that still over-approximates every concrete execution.

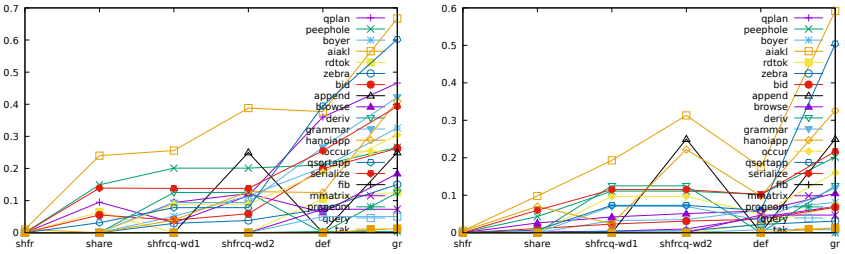


Fig. 2. (a) Precision using flat distance and (b) tree distance (micro-benchmarks)

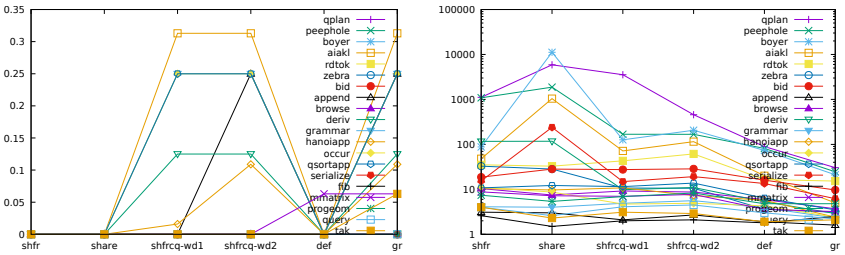


Fig. 3. (a) Precision using top distance and (b) Analysis time (micro-benchmarks)

Case study: variable sharing domains. We have applied the method above on a well known set of (micro-)benchmarks for CLP analysis, and a number of modules from a real application (the LPdoc documentation generator). The programs are analyzed using the CiaoPP framework [20] and the domains *shfr* [31], *share* [23,30], *def* [15,2], and *sharefree_clique* [32] with different widenings. All these domains express sharing between variables among other things, and we compare them with respect to the base *share* domain. All experiments are run on a Linux machine with Intel Core i5 CPU and 8GB of RAM.

Fig. 2 and Fig. 3 show the results for the micro-benchmarks. Fig. 4 and Fig. 5 show the same experiment on LPdoc modules. In both experiments we measure the precision using the flat distance, tree distance, and top distance. In general, the results align with our a priori knowledge: that *shfr* is strictly more precise than all other domains, but can sometimes be slower; while *gr* is less precise and generally faster. As expected, the flat and tree distances show that *share* is in all cases less precise than *shfr*, and not significantly cheaper (sometimes even more costly). The tree distance shows a more pronounced variation of precision when comparing *share* and widenings. While this can also be appreciated in the

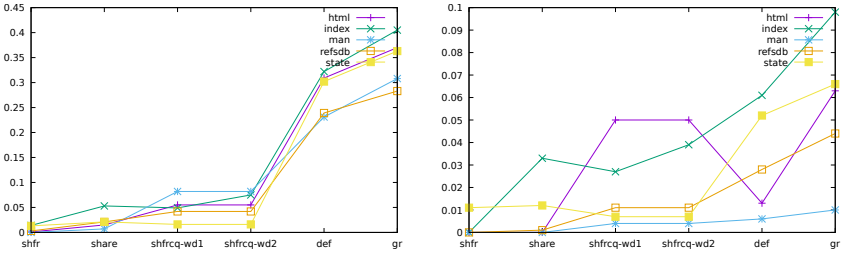


Fig. 4. (a) Precision using flat distance and (b) tree distance (LPdoc benchmark)

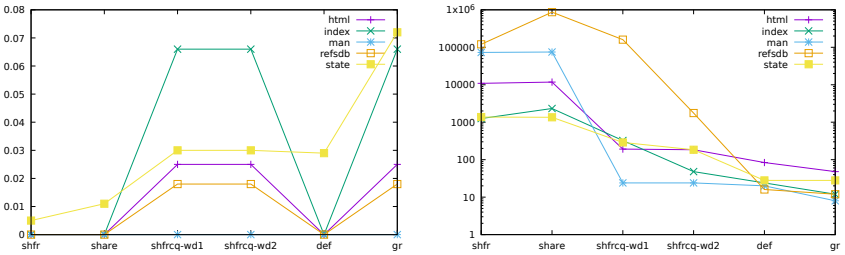


Fig. 5. (a) Precision using top distance and (b) Analysis time (LPdoc benchmarks)

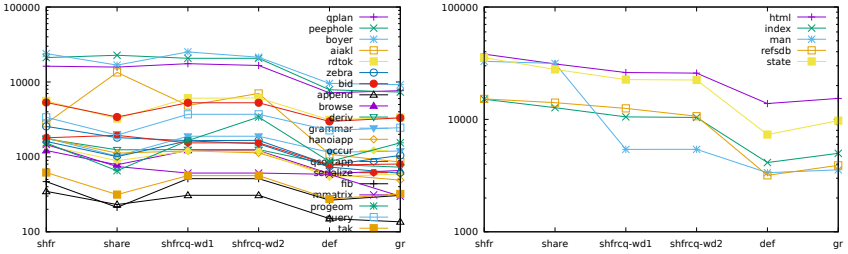


Fig. 6. (a) Analysis size (micro-benchmarks) and (b) Analysis size (LPdoc benchmark)

top distance, the top distance fails to show the difference between *share* and *shfr*. Thus, the tree distance seems to offer a good balance. For small programs where analysis requires less than 100ms in *shfr*, there seems to be no advantage in using less precise domains. Also as expected, for large programs widenings provide significant speedups with moderate precision lose. Small programs do not benefit in general from widenings. Finally, the *def* domain shows very good precision w.r.t. the top distance, representing that the domain is good enough to capture the behavior of predicates at the module interface for the selected benchmarks.

Fig. 6 reflects the size of the AND-OR tree and experimentally it is correlated with the analysis time. The size measures the cost of representing abstract substitutions as Prolog terms (roughly as the number of functor and constant symbols).

6 Related Work

Distances in lattices: Lattices and other structures that arise from order relations are common in many areas of computer science and mathematics, so it is not surprising that there have been already some attempts at proposing metrics in them. E.g., [18] has a dedicated chapter for metrics in lattices. *Distances among terms:* Hutch [22], Nienhuys-Cheng [34] and Jan Ramon [36] all propose distances in the space of terms and extend them to distances between sets of terms or clauses. Our proposed distance for *regular types* can be interpreted as the abstraction of the distance proposed by Nienhuys-Cheng. Furthermore, [36] develop some theory of metrics in partial orders, as also does De Raedt [11]. *Distances among abstract elements and operators:* Logozzo [27] proposes defining metrics in partially ordered sets and applying them to quantifying the relative loss of precision induced by numeric abstract domains. Our work is similar in that we also propose a notion of distance in abstract domains. However, they restrict their proposed distances to finite or numeric domains, while we focus instead on logic programming-oriented, possible infinite, domains. Also, our approach to quantifying the precision of abstract interpretations follows quite different ideas. They use their distances to define a notion of error induced by an abstract value, and then a notion of error induced by a finite abstract domain and its abstract operators, with respect to the concrete domain and concrete operators. Instead, we work in the context of given programs, and quantify the difference of precision between the results of different analyses for those programs, by extending our metrics in abstract domains to metrics in the space of abstract executions of a program and comparing those results. Sotin [37] defines measures in $\mathbb{R}^n$ that allow quantifying the difference in precision between two abstract values of a numeric domain, by comparing the size of their concretizations. This is applied to guessing the most appropriate domain to analyse a program, by under-approximating the potentially visited states via random testing and comparing the precision with which different domains would approximate those states. Di

Pierro [13] proposes a notion of probabilistic abstract interpretation, which allows measuring the precision of an abstract domain and its operators. In their proposed framework, abstract domains are vector spaces instead of partially ordered sets, and it is not clear whether every domain, and in particular those used in logic programming, can be reinterpreted within that framework. Cortesi [7] proposes a formal methodology to compare qualitatively the precision of two abstract domains with respect to some of the information they express, that is, to know if one is strictly more precise that the other according to only part of the properties they abstract. In our experiments, we compare the precision of different analyses with respect to some of the information they express. For some, we know that one is qualitatively more precise than the other in Cortesi's paper's sense, and that is reflected in our results.

7 Conclusions

We have proposed a new approach for measuring and comparing precision across different analyses, based on defining distances in abstract domains and extending them to distances between whole analyses. We have surveyed and extended previous proposals for distances and metrics in lattices or abstract domains, and proposed metrics for some common (C)LP domains. We have also proposed extensions of those metrics to the space of whole program analysis. We have implemented those metrics and applied them to measuring the precision of different sharing-related (C)LP analyses on both benchmarks and a realistic program. We believe that this application of distances is promising for debugging the precision of analyses and calibrating heuristics for combining different domains in portfolio approaches, without prior knowledge and treating domains as black boxes (except for the translation to the *base* domain). In the future we plan to apply the proposed concepts in other applications beyond measuring precision in analysis, such as studying how programming methodologies or optimizations affect the analyses, comparing obfuscated programs, giving approximate results in semantic code browsing [14], program synthesis, software metrics, etc.

References

1. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *Proc. of ESOP'07*, LNCS 4421, pages 157–172. Springer, 2007.
2. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In Springer-Verlag, editor, *Static Analysis Symposium, SAS'94*.
3. Gourinath Banda and John P. Gallagher. Analysis of Linear Hybrid Systems in CLP. In Michael Hanus, editor, *LOPSTR*, volume 5438 of *LNCS*, pages 55–70. Springer, 2009.
4. Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn Clause Solvers for Program Verification. In *Essays to Yuri Gurevich on his 75th Birthday*, pages 24–51, 2015.

5. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.
6. F. Bueno, M. García de la Banda, and M. V. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, November 1994.
7. A. Cortesi, G. File, and W. Winsborough. Comparison of Abstract Interpretations. In *Nineteenth International Colloquium on Automata, Languages, and Programming*, volume 623. Springer-Verlag LNCS, 1992.
8. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252. ACM Press, 1977.
9. P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In *Types in Logic Programming*, pages 157–187. MIT Press, 1992.
10. Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. VeriMAP: A Tool for Verifying Programs through Transformations. In *TACAS, ETAPS*, pages 568–574, 2014.
11. Luc De Raedt and Jan Ramon. Deriving distance metrics from generality relations. *Pattern Recogn. Lett.*, 30(3):187–191, February 2009.
12. S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, 1989.
13. Alessandra Di Pierro and Herbert Wiklicky. Measuring the precision of abstract interpretations. In *Logic Based Program Synthesis and Transformation*, pages 147–164, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
14. I. Garcia-Contreras, J. F. Morales, and M. V. Hermenegildo. Semantic Code Browsing. *TPLP (ICLP'16 Special Issue)*, 16(5-6):721–737, October 2016.
15. M. García de la Banda and M. V. Hermenegildo. A Practical Application of Sharing and Freeness Inference. In *1992 Workshop on Static Analysis WSA'92*, number 81–82 in BIGRE, pages 118–125, Bourdeaux, France, September 1992. IRISA-Beaulieu.
16. M. García de la Banda, M. V. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM TOPLAS*, 18(5), 1996.
17. S. Grebenshchikov, A. Gupta, N. P. Lopes, Co. Popeea, and A. Rybalchenko. HSF(C): A Software Verifier Based on Horn Clauses. In *TACAS*, pages 549–551, 2012.
18. George Grätzer. *General Lattice Theory, second edition*. 01 1998.
19. Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn Verification Framework. In *CAV*, pages 343–361, 2015.
20. M. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Comp. Progr.*, 58(1–2), 2005.
21. M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *JLP*, 13(4):349–367, August 1992.
22. Alan Hutchinson. Metrics on terms and clauses. In Maarten van Someren and Gerhard Widmer, editors, *Machine Learning: ECML-97*, pages 138–145, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.
23. D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *North American Conference on Logic Programming*, 1989.

24. B. Kafle, J. P. Gallagher, and J. F. Morales. RAHFT: A Tool for Verifying Horn Clauses Using Abstract Interpretation and Finite Tree Automata. In *CAV*, pages 261–268, 2016.
25. A. Kelly, A. Macdonald, K. Marriott, H. Sondergaard, P. Stuckey, and R. Yap. An optimizing compiler for CLP(R). In *Proc. of Constraint Programming (CP'95)*. LNCS, Springer, 1995.
26. U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. V. Hermenegildo, and K. Eder. Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In *LOPSTR*, volume 8901 of *LNCS*, pages 72–90. Springer, 2014.
27. Francesco Logozzo, Corneliu Popeea, and Vincent Laviron. Towards a quantitative estimation of abstract interpretations (extended abstract). In *Workshop on Quantitative Analysis of Software*, June 2009.
28. Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. From Datalog to FLIX: a Declarative Language for Fixed Points on Lattices. In *PLDI, ACM*, pages 194–208, 2016.
29. K. Marriott, H. Søndergaard, and N.D. Jones. Denotational Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, 1994.
30. K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *NACLP'89*, pages 166–189. MIT Press, October 1989.
31. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *ICLP'91*, pages 49–63. MIT Press, June 1991.
32. J. Navas, F. Bueno, and M. V. Hermenegildo. Efficient Top-Down Set-Sharing Analysis Using Cliques. In *8th Int'l. Symp. on Practical Aspects of Declarative Languages (PADL'06)*, number 2819 in LNCS, pages 183–198. Springer, January 2006.
33. J. Navas, M. Méndez-Lojo, and M. V. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *BYTECODE'09*, volume 253 of *ENTCS*. Elsevier, March 2009.
34. Shan-Hwei Nienhuys-Cheng. Distance between herbrand interpretations: A measure for approximations to a target concept. In Nada Lavrač and Sašo Džeroski, editors, *Inductive Logic Programming*, pages 213–226, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.
35. G. Puebla, F. Bueno, and M. V. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.
36. J. Ramon and M. Bruynooghe. A framework for defining distances between first-order logic objects. In *8th Intl. WS on Inductive Logic Programming*, pages 271–280, 1998.
37. Pascal Sotin. Quantifying the Precision of Numerical Abstract Domains. Research report, INRIA, February 2010.
38. P. Van Roy and A.M. Despain. High-Performance Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.

A Theory of Section 3

A.1 Properties inherited by abstraction or concretization of distances

Proposition 1. *Let us consider an abstract domain D_α, that abstracts the concrete domain D, with abstraction function $\alpha : D \to D_\alpha$ and concretization function $\gamma : D_\alpha \to D$. Both domains are complete lattices and α and γ form a Galois connection. Then:*

(1) If $d_\alpha : D_\alpha \times D_\alpha \to \mathbb{R}$ is a metric in the abstract domain, then $d : D \times D \to \mathbb{R}$, $d(A, B) = d_\alpha(\alpha(A), \alpha(B))$ is a pseudometric in the concrete domain. If d_α is order-preserving, so it is d.

(2) If $d : D \times D \to \mathbb{R}$ is a metric in the concrete domain, then $d_\alpha : D_\alpha \times D_\alpha \to \mathbb{R}$, $d_\alpha(a, b) = d(\gamma(a), \gamma(b))$ is a pseudometric in the abstract domain. If the Galois connection is a Galois insertion, then d is a full metric. If d is order-preserving, so it is d_α.

*Proof (**Proof**).*

– (1)

- d is a pseudometric:
  - Non-negativity: $d(A, B) = d_\alpha(\alpha(A), \alpha(B)) \geq 0$, since d_α is non-negative
  - Weak identity of indiscernibles : $d(A, A) = d_\alpha(\alpha(A), \alpha(A)) = 0$, since d_α fulfills the identity of indiscernibles
  - Symmetry: $d(A, B) = d_\alpha(\alpha(A), \alpha(B)) = d_\alpha(\alpha(B), \alpha(A)) = d(B, A)$, since d_α is symmetric
  - Triangle inequality: $d(A, C) = d_\alpha(\alpha(A), \alpha(C)) \leq d_\alpha(\alpha(A), \alpha(B)) + d_\alpha(\alpha(B), \alpha(C)) = d(A, B) + d(B, C)$, since d_α fulfills the triangle inequality

- d is *order-preserving*:
  If $A \subseteq B \subseteq C$, then $\alpha(A) \sqsubseteq \alpha(B) \sqsubseteq \alpha(C)$, since α is monotonic. But then $d(A, B) = d_\alpha(\alpha(A), \alpha(B)) \leq d_\alpha(\alpha(A), \alpha(C)) = d(A, C)$, since d_α is order-preserving.

– (2)

- d_α is a pseudometric: analogous. Besides, if the Galois connection is a Galois insertion, then γ is injective (otherwise, $\exists\ a \neq b \in D_\alpha\ s.t.\ \gamma(a) = \gamma(b) \implies \alpha(\gamma(a)) = \alpha(\gamma(b)) \implies a = b$, which is absurd). But then $d_\alpha(a, b) = 0 \implies d(\gamma(a), \gamma(b)) = 0 \implies \gamma(a) = \gamma(b) \implies a = b$, and therefore d_α is a full metric
- d_α is *order-preserving*: Analogous

B Examples for section 4

B.1 Example of *flat* distance

The analysis shown in Fig. 1 has only one triple $\langle L, \lambda^c, \lambda^s \rangle$ for each program point. Let us consider a different analysis for the same program, in which there is no information about the imported predicate `partition/4`, and therefore the analysis needs to assume the most general abstract substitution on success for calls to that predicate. Fig. 7 shows the result of the analysis in the same manner as Fig. 1 does. We observe that this time there are program points which have more that one triple in the analysis. Let us denote each program point as `P/A/N/M`, where that represents the `M`-th literal of the `N`-th clause of the predicate `P/A`. The correspondence between program points and analysis nodes is the following:

`quicksort/2/0` (entry)	`quicksort/2/1/1`	`qsort/3/1/1`	`qsort/3/1/2`	`qsort/3/1/3`
(1)	(2)	(3), (5)	(4), (6)	(7), (8)

The resulting single triples $\langle L, \lambda^c, \lambda^s \rangle$ for each program point will be the following:

`quicksort/2/0` (entry)	(1)	$\langle quicksort(Xs, Ys),\ \{Xs/g, Ys/ng\},\ \{Xs/g, Ys/any\}\rangle$
`quicksort/2/1/1`	(2)	$\langle qsort(Xs, Ys, []),\ \{Xs/g, Ys/ng\},\ \{Xs/g, Ys/any\}\rangle$
`qsort/3/1/1`	(3) '$\sqcup$' (5)	$\langle partition(Xs, X, L, R),\ \{Xs/any, X/any, L/ng, R/ng\}),\ \{Xs/any, X/any, L/any, R/any\})\rangle$
`qsort/3/1/2`	(4) '$\sqcup$' (6)	$\langle qsort(Xs, Ys, Zs),\ \{Xs/any, Ys/ng, Zs/any\},\ \{Xs/any, Ys/any, Zs/any\}\rangle$
`qsort/3/1/3`	(7) '$\sqcup$' (8)	$\langle qsort(Xs, Ys, [Z\|Zs]),\ \{Xs/any, Ys/ng, Z/any, Zs/any\},\ \{Xs/any, Ys/any, Z/g, Zs/any\}\rangle$

Let us compare the two analyses shown in Figs. 1 and 7. We already have their representation as one triple $\langle L, \lambda^c, \lambda^s \rangle$ for each program point. The distances for each program point, computed as the average of the distance between its abstract call substitution and the distance between its abstract success substitution, is the following:

`quicksort/2/0` (entry)	`quicksort/2/1/1`	`qsort/3/1/1`	`qsort/3/1/2`	`qsort/3/1/3`
0.354	0.354	0.427	0.454	0.467

The final distance between the analysis could be the average of all of them, 0.411. Alternatively, we could assign different weights to each program point taking into account the structure of the program, and use a weighted average as final distance. For example, we could assign the weights of the table below, which would yield the final distance 0.378.

`quicksort/2/0` (entry)	`quicksort/2/1/1`	`qsort/3/1/1`	`qsort/3/1/2`	`qsort/3/1/3`
$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$

B.2 Example of the *tree* distance

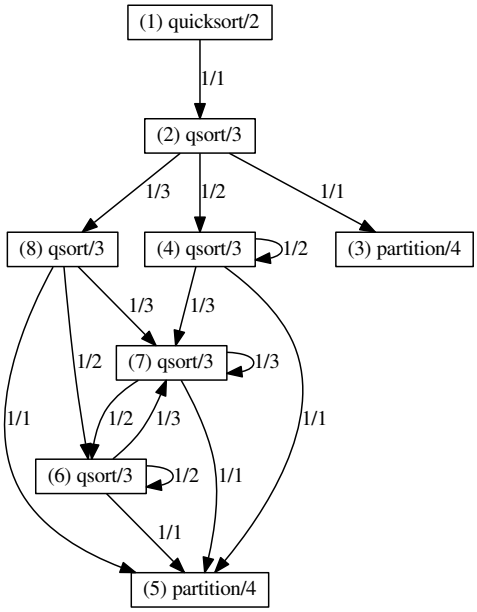
Let us compute the *tree* distance between the two analyses shown in Figs. 1 and 7. Fig. 8 shows the tree with distances between both analysis node to

node. The and-nodes are omitted for simplicity. Each or-node is a quintuple (P, Id_1, Id_2, D, W): P is the predicate corresponding to that program point, I_1 is the identifier of the node in analysis 1 corresponding to that or-node, I_2 is the analogous in analysis 7, D is the distance between the two nodes, and W is the corresponding weight to the distance in that node when we apply the definition of the *tree* distance. We use a factor $\mu = \frac{1}{5}$, and the average of the distance between the call substitutions and the distance between the success substitutions as distance between nodes, using an abstract distance in the underlying *groundness* domain.

If we follow the tree through the edges labelled 1,2,3..., we observe that we are visiting the same node over and over with decreasing weights $0.043, 0.011, 0.003 \ldots = w\frac{1}{5} + w\frac{4}{5}\frac{1}{3}\frac{1}{5} + w\frac{4}{5}\frac{1}{3}\frac{4}{5}\frac{1}{3}\frac{1}{5} + \ldots$, where $w = 1\frac{4}{5}\frac{1}{1}\frac{4}{5}\frac{1}{3}$. The sum of those weights converges ($\frac{1}{5}w\sum_{i=0}^{\infty}(\frac{4}{5}\frac{1}{3})^i = \frac{1}{5}w\frac{15}{11}$), but it is not trivial to compute in the general case and for all cases.

However, we can compute the final sum solving the following systems of equations, where the variable $X_{i,j}$ corresponds to the node (P, i, j, D, W):

$$\begin{cases} X_{1,1} = \frac{1}{5} * 0.177 + \frac{4}{5}X_{2,2} \\ X_{2,2} = \frac{1}{5} * 0.177 + \frac{4}{5}\frac{1}{3}X_{3,3} + \frac{4}{5}\frac{1}{3}X_{4,4} + \frac{4}{5}\frac{1}{3}X_{5,8} \\ X_{3,3} = 0.177 \\ X_{4,4} = \frac{1}{5} * 0.348 + \frac{4}{5}\frac{1}{3}X_{3,5} + \frac{4}{5}\frac{1}{3}X_{4,4} + \frac{4}{5}\frac{1}{3}X_{5,7} \\ X_{5,8} = \frac{1}{5} * 0.177 + \frac{4}{5}\frac{1}{3}X_{3,5} + \frac{4}{5}\frac{1}{3}X_{4,6} + \frac{4}{5}\frac{1}{3}X_{5,7} \\ X_{3,5} = 0.427 \\ X_{5,7} = \frac{1}{5} * 0.177 + \frac{4}{5}\frac{1}{3}X_{3,5} + \frac{4}{5}\frac{1}{3}X_{4,6} + \frac{4}{5}\frac{1}{3}X_{5,7} \\ X_{4,6} = \frac{1}{5} * 0.177 + \frac{4}{5}\frac{1}{3}X_{3,5} + \frac{4}{5}\frac{1}{3}X_{4,6} + \frac{4}{5}\frac{1}{3}X_{5,7} \end{cases}$$



(1) $\langle quicksort(Xs, Ys),\ \{Xs/g, Ys/ng\},\ \{Xs/g, Ys/any\}\rangle$
(2) $\langle qsort(Xs, Ys, []),\ \{Xs/g, Ys/ng\},\ \{Xs/g, Ys/any\}\rangle$
(3) $\langle partition(Xs, X, L, R),\ \{Xs/g, X/g, L/ng, R/ng\}),\ \{Xs/g, X/g, L/any, R/any\}\rangle$
(4) $\langle qsort(Xs, Ys, Zs),\ \{Xs/any, Ys/ng, Zs/g\}),\ \{Xs/any, Ys/any, Zs/g\}\rangle$
(5) $\langle partition(Xs, X, L, R),\ \{Xs/any, X/any, L/ng, R/ng\}),\ \{Xs/any, X/any, L/any, R/any\}\rangle$
(6) $\langle qsort(Xs, Ys, Zs),\ \{Xs/any, Ys/ng, Zs/any\},\ \{Xs/any, Ys/any, Zs/any\}\rangle$
(7) $\langle qsort(Xs, Ys, [Z|Zs]),\ \{Xs/any, Ys/ng, Z/any, Zs/any\}),\ \{Xs/any, Ys/any, Z/any, Zs/any\}\rangle$
(8) $\langle qsort(Xs, Ys, [Z|Zs]),\ \{Xs/any, Ys/ng, Z/g, Zs/any\},\ \{Xs/any, Ys/any, Z/g, Zs/any\}\rangle$

Fig. 7. Analysis of `quicksort/2`.

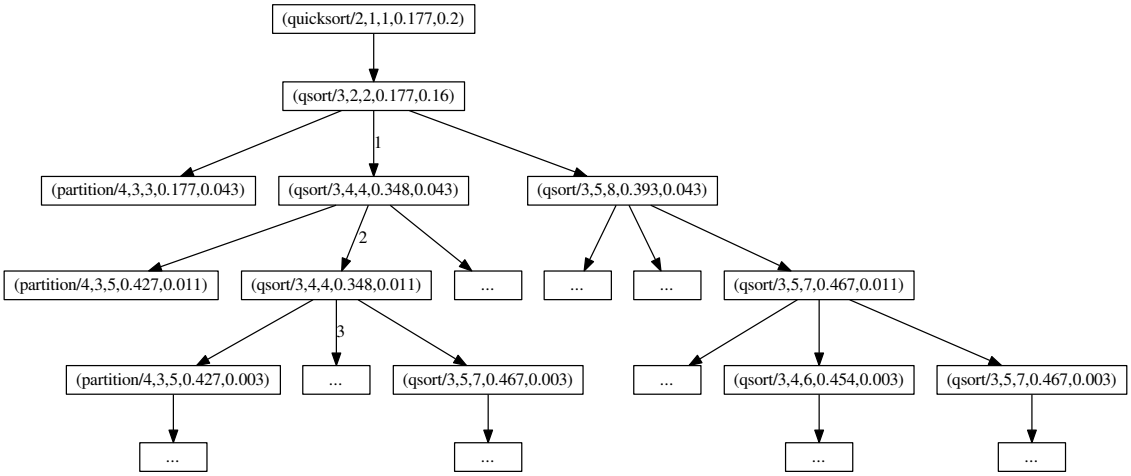


Fig. 8. 3rd approach: whole abstract execution tree

Synthesizing Imperative Code from Answer Set Programming Specifications*

Sarat Chandra Varanasi, Elmer Salazar, Neeraj Mittal, and Gopal Gupta

Department of Computer Science,
The University of Texas at Dallas,
Richardson, Texas, USA
{sxv153030,elmer.salazar,neerajm,gupta}@utdallas.edu

Abstract. We consider the problem of obtaining an implementation of an algorithm from its specification. We assume that these specifications are written in answer set programming (ASP). ASP is an ideal formalism for writing specifications due to its highly declarative and expressive nature. To obtain an implementation from its specification, we utilize the operational semantics of ASP implemented in the s(ASP) system. This operational semantics is used to transform the declarative specification written in ASP to obtain an equivalent efficient program that uses imperative control features. This work is inspired by our overarching goal of automatically deriving efficient concurrent algorithms from declarative specifications. This paper reports our first step towards achieving that goal where we restrict ourselves to simple sequential algorithms. We illustrate our ideas through several examples. Our work opens up a new approach to logic-based program synthesis not explored before.

Keywords: Program Synthesis, Program Transformation, Answer Set Programming, Partial Evaluation, Symbolic Execution

1 Introduction

Program synthesis concerns generating programs according to given specifications. This problem has been tackled in various ways. One of the earliest approaches was to use theorem proving to generate imperative programs from a given logical input-output specification [10]. Extensive work using logical approaches are surveyed [2]. More recently synthesis has been reduced to the task of verification [15]. In this paper, we provide a method to extract programs from executable specifications written in a particular class of datalog answer set programs, by means of program transformation. Answer Set Programming (ASP) is a declarative, logic-based programming paradigm for solving combinatorial search and knowledge representation problems [6]. In ASP, specification and computation are synonymous, which is highly desirable to rapidly prototype systems [4]. However the implementations of ASP are quite complex [5]. It is

* Work partially supported by US NSF grant IIS 1718945.

rather difficult to justify why a specification computes what it computes. On the other hand, imperative programs are relatively easier to trace and in general enjoy faster run-times than their declarative program counterparts. This is because imperative programs often represent computationally faster implementations of a specification without directly being concerned about the specification. This makes deriving imperative programs from ASP specifications highly desirable. Our main insight is to employ the recently-developed operational semantics for ASP realized in the s(ASP) system [11, 1]. s(ASP) is a query-driven answer set programming system which computes the (partial) stable models of a query of an answer set program without grounding the program first. Traditional implementations of ASP are based on grounding the program to its propositional equivalent and then using SAT solvers to compute the *answer sets*. Using the operational semantics of s(ASP), one can follow how the stable models are computed for a query in a step-by-step manner. This provides us the right playground to extract an imperative program while simplifying all the machinery that makes s(ASP) work.

Our key contribution is the following. We demonstrate how program transformation, based on s(ASP)-style operational semantics, can be used to realize efficient imperative programs from their declarative ASP specifications. We illustrate our approach through several sequential algorithms. Our major contribution is the novel use of the *forall* mechanism that is part of s(ASP)'s operational semantics. The *forall* directly translates into a *for-loop* in the derived imperative program. Eventually, we plan to employ partial evaluation obeying the s(ASP) operational semantics to further improve efficiency.

Answer set programming is essentially logic programming with negation as failure (NAF) under the stable model semantics [6]. Answer set programs consist of *even loops* that act as *generators of worlds* and *odd loops* (or *constraints*) that act as *destroyer of worlds*. ASP provides a paradigm where each rule can be written and understood in isolation from others. Due to presence of negation as failure, ASP permits specification of a concept by stating what it is not. ASP thus serves as a very high-level and expressive specification language.

Our main idea is as follows: Programs are written in terms of abstractions over certain allowed primitives and further abstractions are built on top of existing abstractions. This applies to specifications as well, particularly for the specifications we are interested in this paper. In logic programs, a specification contains either positive terms or negated terms. However, in a corresponding imperative program, there are only function-calls and constraints (if-checks). We thus eliminate negated terms in the imperative program making them implicit. In doing so, the meaning of negated terms is propagated downward from the abstractions to the primitives. In ASP, combinations of possibilities or situations that are not permitted are expressed as constraints that a consistent world (answer set) must obey; those that are permitted or desired are represented through simple rules, even loops, etc. Our insight behind obtaining an imperative algorithm from an ASP specification involves simply following the s(ASP) operational semantics to orchestrate the calls to predicates so that constraints

end up as if-then-else checks in the imperative execution. In addition, s(ASP) semantics creates universally quantified variables in the body of negated rules that translate directly into an imperative *for-loop*. These ideas are illustrated in more detail later.

Our ultimate goal is to synthesize efficient concurrent algorithms from their declarative specifications. Concurrent algorithms are notoriously difficult to design. They are also equally difficult to verify. ASP provides the right formalism to represent concepts of concurrency in an elegant, straightforward way. For instance, the ASP program that describes the behaviour of a concurrent linked list is no different than the ASP program for blocks world planning [6]. Thus, to be able to synthesize imperative code from a concurrent specification, will greatly aide designers of concurrent algorithms. Further, our work corroborates the idea of writing programs that are *correct by construction*.

Rest of the paper is organized as follows: We give an overview of s(ASP) and its features used for synthesis in Section 2. We then provide two examples, one in Section 3 for motivating the idea of synthesis and the second in Section 4 to elaborate in detail. Section 5 describes the assumptions on which our technique is based. Section 6 outlines the entire algorithm followed by concluding remarks in Section 7.

2 Background

The ASP Paradigm: We assume basic familiarity with ASP syntax [6] and its applications in building intelligent systems [3, 6]. The sets of satisfiable truth assignments to the propositional variables of an answer set program constitute its *answer sets* under the stable model semantics [6, 8]. In general, ASP programs have predicates with variables. Most of the ASP solvers [5] ground the predicated program into propositions and find stable models by SAT solving. The s(ASP) system takes a different path by finding stable models relevant to a given query such as `?- p(X).` s(ASP) finds the stable models for the issued query operationally without grounding the program. In short, stable models are solved in traditional ASP solvers, whereas the models are proved in s(ASP). For every answer set, there is a justification tree (proof tree) produced by the s(ASP) system describing all the sub-goals that lead up to the query. The s(ASP) system employs coinductive SLD Resolution [14] to execute even loops in a top down manner. Odd loops and constraints (NMR checks) serve as an extension of the top-level query in order to constrain the answers. For instance, a query `?- p(X)` issued by the user is extended with a constraint {`chk :- ...`} as `?- p(X), chk.` The goal-directed s(ASP) algorithm can be found elsewhere [11].

Prominent Features of s(ASP): We next summarize the salient features of the s(ASP) system that are relevant to our work here. The operational semantics of s(ASP) is a Prolog style execution of an issued query adhering to the stable model semantics. The query triggers a search where all clauses that are consistent are resolved and backtracking happens when inconsistent goals are encountered.

All the terms encountered in the consistent goals constitute the "partial" stable model associated with the query. At a basic level, the current resolvent is invalid when during expansion it leads to a stable model that has both a goal and its negation. This is, in fact, how stable models are constructed in the propositional variant of s(ASP), namely Galliwasp [13]. Note that the execution algorithm of s(ASP) relies on *corecursion* [14] to handle even loops [11], but this feature is not as important for the work reported here.

Handling negation with Dual rules: In s(ASP), negated goals are executed through dual rules. The dual rule of a predicate p systematically negates the literals in a rule body defining p. Dual rules of all predicates together with the rules in the original program represent its *completion* [9]. For instance, the dual rule of the predicate p with the following definition is shown below:

```
%definition of p      %dual of p/0
p :- not q.           not p :- np1, np2
p :- r, not s.        %negating of not q yields q
                      np1 :- q.
                      %negating r, not s yields
                          disjunction (not r) ∨ s
                      np2 :- not r.
                      np2 :- s.
```

Forall mechanism: We have just shown dual rules for propositions which are simple enough. However, writing dual rules for predicates is more involved. One complication is due to implicit quantifiers in predicate rules. For a rule such as:

```
p(X) :- q(Y), r(X).
```

X is universally quantified over the whole formula, whereas Y is existentially quantified in the body only. Therefore, negating $p(X)$ results in Y being universally quantified in the body as follows:

```
not p(X) :- np1(X) ;  np2(X).
np1(X) :- forall(Y, not q(Y)).
np2(X) :- not r(X).
```

Notice that the universal quantifier for Y in *not* $q(Y)$ is enclosed within a *forall*(...). The *forall* represents a proof procedure that runs through all values in the domain of Y and verifies if the goal enclosed within it is satisfied. In our example, the *forall* checks that *not* $q(Y)$ holds for all values in the domain of Y. No such mechanism existed in prior Prolog based systems. It is this *forall* mechanism coupled with dual rules that we make extensive use of in our synthesis procedure. Because they treat negated predicates *constructively* [11], we turn them into computations in our synthesis procedure.

3 Motivating Example

Consider the program that finds the maximum of n numbers, which is naturally specified in ASP as shown below:

```
max(X) :-  num(X), not smaller(X).
smaller(X) :- num(X), num(Y), X < Y.
```

$num(X)$ provides the domain of numbers over which the input values range. $smaller(X)$ defines when a number X is dominated by another number Y. $max(X)$ gives the definition for X to be the maximum. The main predicate from where the computation to decide whether a given number X is the maximum begins with $max(X)$. The negation of $smaller(X)$ can be translated to a *forall* as shown below:

```
not smaller(X) :- forall(Y, not (num(X), num(Y), X < Y)).
```

Assuming that $num(X)$ and $num(Y)$ are true, the negation only applies to $X < Y$. Therefore, replacing the definition of *not* $smaller(X)$ in $max(X)$ gives us the following:

```
max(X) :-  forall(Y, num(X), num(Y), not (X < Y)).
```

The *forall* definition of $max(X)$ makes apparent the operational flavor involved in finding the maximum of n numbers: enumerate all numbers Y and compare them with X. If X is not smaller than any Y, i.e., $X < Y$ is false for all Y, then X is maximum. The *forall* can be translated to a *for-loop* in imperative languages if the domains of the variables involved in the scope of *forall* are finite. From an Answer Set Programming point of view, $max(X)$ is present in the answer set if the *forall* succeeds (if X is the maximum), otherwise it is not present in the answer set. The abstract code synthesized through program tranformation, thus looks like:

```
def max(x):
    for y in num:
        if x < y:
            return False
    return True
```

The interpretation shown above is not sufficient to translate ASP programs written to solve graph-coloring or sorting an array. We illustrate our idea further in the following section taking the example of graph-coloring.

4 Synthesizing Code for Graph Coloring

We first provide an ASP program and then provide an operational interpretation which is still a logic program. Finally, we show how the various fragments of the "intermediate" logic program[1] can be transformed into an imperative program.

The Graph Coloring Problem: Graph coloring involves selecting a color `c` for every node `v` in an input graph such that no two adjacent (edge-connected) nodes `(x, y)` have the same color `c`. The ASP program is shown below:

[1] Both intermediate logic program and intermediate ASP program are used interchangeably

```
color(X, C) :- node(X), color(C), not another_color(X,C),
               not conflict(X,C).
another_color(X, C) :- node(X), color(C), color(C1), C != C1,
                       color(X, C1).
conflict(X, C) :- node(X), color(C), node(Y), X != Y, edge(X, Y),
                  color(X, C), color(Y, C).
```

The domain of nodes and colors come from the predicates `node/1` and `color/1`. The predicate `color(X, C)` provides the definition of what it means to color a node `X` with color `C`: Color `X` with `C` provided there is no other choice for `X` other than `C` and, the choice of color `C` does not conflict with the color `C'` of all nodes `Y` edge-connected to `X`. The two conditions stated now are represented by predicates `another_color/2` and `conflict/2`, respectively.

Operational interpretation for predicates in Graph Coloring: Considering `color(X, C)` to start off a computation for coloring node `X` (and subsequently the entire graph), we follow the definition of `color(X, C)` in a top-down, left-right manner while simplifying all intervening negations until we either reach an assertion about a domain variable (or) reach a recursive definition of `color/2`.

In the definition of `color(X, C)`, the dual clause synthesized by s(ASP) for `another_color(X, C)` will be used for the call `not another_color(X, C)`. This clause contains a *forall* with disjunction of negated terms as follows:

```
not another_color(X, C) :-
forall(C1,node(X),color(C),(not (C!= C1) or not color(X,C1))).
```

Operationally, `not another_color(X, C)` checks two constraints over all colors `C1`. For the first constraint we check if `C != C1` is false, an assertion on the domain variables `C` and `C1`. The second constraint checks if the negation of `not color(X, C1)` is true, which is the negation of the predicate that started the computation, but with a possibly different binding for `C1` than `C`. In order to compute the negation of `color(X, C1)` we generate the following dual rule:

```
not color(X, C) :- node(X), color(C), another_color(X, C).
not color(X, C) :- node(X), color(C), conflict(X, C).
```

Similarly, `not conflict(X, C)` is translated in terms of *forall* :

```
not_conflict(X, C) :-  forall(Y,node(X),color(C),node(Y),
                       (not (X != Y)) or not edge(X,Y) or
                        not color(Y,C)).
```

Notice that the negation is not applied to `node(X)`, `color(C)` and `node(Y)` as we are not concerned about bindings for `X`, `C`, `Y` that are neither nodes nor colors. Further, `not color(X, C)` is skipped in the disjunction. This is because expanding negation of `color(X, C)` in the expansion of `color(X, C)` leads to a direct contradiction.

Once we have the intermediate logic program generated by the s(ASP) compiler (s(ASP) automatically generates dual rules), we have to transform the program further in a top-down, left-right manner, using s(ASP) semantics, starting

from `color(X, C)` while generating imperative code for each of the predicates found in the body of `color(X, C)` (while excluding domain predicates). For our current research, the language of choice is Python. The choice could as well be any other imperative language as long as it supports recursion. Again, support for recursion is also not strictly necessary as the lack of it thereof would increase the size of the generated code to explicitly handle it.

Translating Graph Coloring to Imperative code: The intermediate ASP program for graph coloring is broken down into its constituent syntactic parts and the translation for each of them is discussed in turn:

1. ***Main Rule-Head maps to Function call (Abstraction):*** `color(X, C)` is our starting point which is in fact the rule-head defining `color(X, C)`. Rule heads represent abstractions in Logic Programming [9], and hence an obvious choice is to map them to a function prototype such as: `def color(x, c)`. But it is not clear what the "function" `color(x, c)` should compute. At this point we can treat the call to `color(x, c)` as a query in the s(ASP) system: `?- color(X, C)`. The query immediately finds bindings for node `X`, color `C` and also relevant bindings `color(X′, C′)` for other node, color pairs `X′,C′` that are consistent with `color(X, C)` in the ASP program.
   In other words, the s(ASP) system finds the first answer set (partial stable model) of the program. If s(ASP) cannot find an answer set, it returns false. Further, imperative languages can only operate on ground values (unlike logic program queries). Thus, the call `color(x, c)` in Python has ground values for parameters `x, c` at run-time. With these arguments, it makes sense that `def color(x, c)` should return all "terms" consistent with the choice `x, c` taken at run-time. Also, at run-time, the function should be able to track the consistent terms computed thus far. Therefore every function call `color(x, c)` is designated an additional parameter named *context*, which stores set of the consistent terms.
   Thus the complete function prototype is: `def color(x, c, context)` with the initial context being the empty set {}. In addition to returning consistent terms, the function should also report failure. Therefore, the function `color(x, c, {})` returns a pair $(success, set)$. *success* is a boolean variable taking the value $True$ upon success and $False$ upon failure. *set* is the set of consistent terms found.
2. ***Domains map to Dictionary objects (Domain mapping):*** Domains are represented by Dictionary objects. A dictionary object can be viewed as a set of key-value pairs and it is iterable over the keys. If the colors in the coloring problem are `color(red), color(green)` and `color(blue)` then the corresponding domain of colors is represented as:

   ```
   colors = {'red' : True, green' : True, 'blue' : True}
   ```

   Domains in rule-bodies, such as `color(X)` translate to `if color[x]: ...`
3. ***Predicates in the rule-body map to a sequence of statements (Sequencing):*** As stated previously, we follow the intermediate logic program in a top-down, left-right manner. For graph coloring, the *forall* for `not`

`another_color(X, C)` is followed by the *forall* for `not conflict(X, C)` in the body of the function `color(x, c, context)`. Note that there are recursive calls to `color` in both the *forall*s. Now that there are a sequence of statements following other statements, the *context* is passed and returned from any recursive calls to `color`. The *context* is always passed-by-value because we might have to backtrack if any recursive call fails for some reason. Although dictionary objects are always passed-by-reference, we emulate this by use of a `copy` function which always copies the value of the *context* before passing it to a recursive call.

4. ***Assertions on domain variables map to if-else conditions (Constraint checking):*** From the intermediate logic program, we can identify three assertions on the domain variables. The first one is the check `not (C != C1)` in `not another_color(X, C)` and the other two are `not (X != Y)` and `not edge(X, Y)` in `not conflict(X, C)`. All these constraints are trivially translated into simple if-conditions. For example, the check `not edge(X, Y)` is translated to `if not edge(x, y):...`. Failure to satisfy this assertion would explore any other assertion in the disjunction. If all assertions are not satisfied, then the program returns $(false, context)$ where *context* is the set of consistent terms explored upto failure.
5. ***Forall maps to a for-loop on the domain variable (Forall transformation):*** At first glance, it is easy to see the *forall* map exactly to the *for-loop* (this is in fact the case for `max(X)` as shown in Section 3). However, this is not always the case. There needs to be additional code added to handle recursive calls within the *forall* . For graph coloring, consider the following input graph:

```
node(a). node(b). node(c). edge(a,b). edge(b,c).edge(c,a).
color(red). color(green). color(blue).
```

It is clear that all the colors assigned to the nodes should be unique as the graph is a complete graph. If we translated the *forall* for `not conflict(X, C)` exactly to a *for-loop*, then the *for-loop* would look like:

```
1  def color(x, c, context):
2      ...
3  # forall corresponding to not conflict(X, C)
4  for y in node:
5    if(not (x != y):
6      # nothing else to check, continue
7    else:
8      if not edge(x, y):
9        # again nothing to check, continue
10     else:
11       ctx = not_color(y, c, copy(context))
12       if ctx['success']:
13         context = ctx
14       else:
15         return {'success': False, 'context': context}
16    ...
```

The above *for-loop*, starting at line 5 and ending at line 15, checks for nodes `y` that are consistent with the choice of `color(x, c)`. Say the choice is `x = a`, `c = green`, the iteration for `y = a` succeeds with the `not (x != y)` succeeding. The iteration for `y = b` fails the check at line 4 and the check at line 8 resulting in the execution of line 11. The dual `not_color(y, c, copy(context))` selects the first color `c′` which is not `c = green`. Say the first color that is not green is `c′ = red`. Then, `not_color(y, c, context)`[2] adds `color(b, red)` to `ctx`. Similary, for `y = c` the color selected would be red. The entire context is now {`color(a, green)`, `color(b, red)`, `color(c, red)`}. It is clear that the constraints are satisfied locally but are inconsistent globally.

To address this and thus to ensure global consistency, we check `y` with all pairs of choices `(x′,c′)` which are in the current context. If there is no violation, we proceed. Otherwise, there must be some choice `(x″,c″)` in the current context that is globally inconsistent and the recursive program reports failure while making another choice `c‴` for `x″` upon backtracking. The rectified code now looks like:

```
1  def color(x, c, context):
2          ...
3  # forall corresponding to not conflict(X, C)
4  for y in node:
5    for(color(x1,c1) in context):
6      if(not (x1 != y)):
7        # nothing else to check, continue
8      else:
9        if not edge(x1, y):
10         # again nothing to check, continue
11       else:
12         ctx = not_color(y, c, copy(context))
13         if ctx[‘success’]:
14           context = ctx
15         else:
16           return {‘success’: False, ‘context’: context}
17       ...
```

Notice that line 5 now ensures the variable y is checked against all terms in the current context. Since we are in the scope of `color(x, c)`, it is assumed that `color(x, c)` is already added to the context. This is done in the first statement of `color(x, c, context){...}` as `def color(x, c, context){ context[‘color(x, c)’] = true ... }`

5 Synthesis Assumptions

The program synthesis process we have illustrated is confined to a class of answer set programs. The source ASP program should be a safe program (datalog

[2] `not_color(y, c, copy(context))`

program) and cannot have head-less rules (constraints). Further, the program should consist of a unique *main predicate*. The program structure is as follows:

```
%main_predicate is defined in terms of abstractions
main_predicate(X1, X2, ..., Xn)
        :- dom_1(X1), dom_2(X2),...,dom_n(Xn),
           abstraction_1(...), abstraction_2(...), ...
               abstraction_m(...).
%every abstractions results in a constraint on domain
    variables
abstraction1(X1, X2, ..., Xp)
        :-  dom_α1(X1), dom_α2(X2), ..., dom_αp(Xp), constraint_α1.
%or, an abstraction can result in a constraint with a
    recursive call
abstraction1(X1, X2, ..., Xp)
        :- dom_α1(X1), dom_α2(X2), ..., dom_αp(Xp),
           constraint_α2, main_predicate(...).
 ...
abstraction_m(X1, X2, ..., Xq)
        :-  dom_μ1(X1), dom_μ2(X2), ..., dom_μq(Xq), constraint_μ1.
abstraction_m(X1, X2, ..., Xq)
        :- dom_μ1(X1), dom_μ2(X2), ..., dom_μq(Xq),
           constraint_μ2, main_predicate(...).
```

Abstractions can be either positive literals or NAF literals (dual rules). Constraints are traditional relational operators found in most imperative languages. Notice that the abstractions do not call other abstractions directly. This is required of the source ASP program. Because if they do call other abstractions directly, constraint propagation through negation becomes non-trivial and most of the s(ASP) machinery would have to be exposed in the imperative program. These restrictions are discussed more in Section 7.

6 Synthesis Task

Every rule-head in the ASP program stands for an abstraction and defines more abstractions that eventually reduce to constraints on domain variables. For the synthesis algorithm, it is useful to see the intermediate ASP program as a grammar with associated attributes for each of the non-terminal symbols. Each rule-head represents its own non-terminal symbol including the dual rules. Every rule-head (non-terminal symbol) has a set of attributes which are relevant for translation.

If the rule-head is `R`, then `R.arity` and `R.arglist` represent the number of arguments and the list of arguments with their associated domains. `R.callsMain` is a boolean attribute that signifies `R` directly calls the main predicate. This helps us in adding extra code required for global consistency checking within a *forall* as illustrated in the graph coloring example. `R.bodyVariables` represents the set of body variables that appear in the definition of the rule represented by the rule-head `R`. Body variables give rise to choice-points and `R.bodyVariables` enables

us to add backtracking code. Constraints on domain variables such as $X < Y$ are treated as terminal symbols in the grammar. We assume the rule-head `R` stands for its own name. This is useful when `R` represents a domain or an input fact (such as edge relation for graphs). Next, we give a syntax-directed translation of all syntactic fragments found in the intermediate logic program.

6.1 Syntax-Directed Translation

We provide a top-down translation of ASP code (grammar symbols at this juncture) into imperative code. The translation function T maps syntax of intermediate ASP to Python code. Letters $p, q, ..$ are used to denote predicates. Predicates get mapped to function-calls. Letters $X_i, Y_j, ...$ denote arguments of predicates. The arguments get translated to variables $x_i, y_j, ...$ with their respective associated domains $D_{x_i}, D_{y_j}, ...$ and so on. The notation $\bar{X}$ stands for an arbitrary list of arguments. It is also used in the Python translation as $\bar{x}$, representing an arbitrary list of arguments in a function-call. The *context*, which is the set of consistent terms of the main predicate, is taken to be visible in all code fragments and is universally passed as the last argument at every function-call. The call associated with the predicate $q(\bar{Y})$ would be $q(\bar{y}, context)$. Dual rules are prefixed with $not_$. For instance, if $p(\bar{x}, context)$ is a function-call associated with the predicate $p(\bar{X})$, then $not_p(\bar{x}, context)$ stands for the call associated with the dual rule $not\ p(\bar{X})$. We list important rules for translation and explain the ones that are not obvious. More general rules can be found elsewhere [16].

6.2 Synthesis Procedure

The procedure *Gen_Intermediate* generates the dual rules and foralls on body variables (if any). This procedure is quite straightforward and not further elucidated. Likewise, *Gen_Attribute_Grammar* associates the intermediate program with grammar symbols and annotates them with appropriate attributes. Again, it is easy to determine the body variables of a rule, to check if an abstraction calls the main predicate directly. The algorithm is given below:

Algorithm 1 Synthesizing Python from source ASP

```
Code ← ∅
P_int ← Gen_Intermediate(P_input)          ▷ generate dual rules and foralls
G ← Gen_Attribute_Grammar(P_int)
for every fact f ∈ P_input do
    Code ← Code ∪ T⟦f⟧                     ▷ add code for input facts and domains
end for
for every rule R ∈ G do
    Code ← Code ∪ T⟦R⟧                     ▷ where T is the translation function
end for
```

I. Domain mapping:

$T[\![domain(constant)]\!] =$ `domain[str(constant)] = True`

II. Constraint Checking:

$$T[\![X\ relop\ Y]\!] = \begin{cases} \texttt{if not x relop y:} \\ \texttt{ return \{'success':False,'context':ctx\}} \end{cases}$$

where *relop* is relational operator

III. Sequencing:

$$T[\![\ p_1(\bar{X}_1), p_2(\bar{X}_2)\ ]\!] = \begin{cases} \texttt{ctx_p1 = } p_1(x_1\texttt{,..,copy(ctx))} \\ \texttt{if ctx['success']:} \\ \texttt{ ctx = ctx_p1} \\ \texttt{ } T[\![p_2(\bar{X}_2)]\!] \\ \texttt{else:} \\ \texttt{ return\{'success':False,'context':ctx_p1\}} \end{cases}$$

IV. Double negations cancel out:

$$T[\![\ not\ not_p(\bar{X})\ ]\!] = T[\![\ p(\bar{X})\ ]\!]$$

V. Forall transformation (global consistency):

$$T[\![\ forall(Y, p(X_1, .., Y, .., X_n)\ ]\!] = \begin{cases} \texttt{for y in } D_y\texttt{:} \\ \texttt{ for main_pred(}x_{\alpha_1}\texttt{,..,}x_{\alpha_m}\texttt{) in ctx:} \\ \texttt{ ctx_p = p(}x_{\beta_1}\texttt{,..y,..,}x_{\beta_m}\texttt{,copy(ctx))} \\ \texttt{ if ctx_p['success']:} \\ \texttt{ ctx = ctx_p} \\ \texttt{ else:} \\ \texttt{ return \{'success':False'context':ctx\}} \end{cases}$$

where `p.callsMain` is *true*, X_{β_1},..,X_{β_m} are some variables in $\texttt{p(}X_1\texttt{,..,}Y\texttt{,..,}X_n\texttt{)}$ which unify with $\texttt{main_pred(}X_{\alpha_1}\texttt{,..,}X_{\alpha_m}\texttt{)}$. Also, the choice $p(x_1, .., x_n, ctx)$ is checked against all terms in the context explored thus far.

VI. Disjunction translates to nested if-else condition:

$$T[\![\ p_1(\bar{X}_1) \vee p_2(\bar{X}_2)\ ]\!] = \begin{cases} \texttt{context = } p_1\texttt{(}x_1\texttt{, ..., copy(context))} \\ \texttt{if not context['success']:} \\ \quad T[\![p_2(\bar{X}_2)]\!] \\ \texttt{else:} \\ \quad \texttt{pass} \end{cases}$$

VII. Abstraction (positive rule-heads):

$$T[\![p(\bar{X}) :- q_1(\bar{X}_1), .., q_m(\bar{X}_m)]\!] = \begin{cases} \texttt{def p(}\bar{x}\texttt{,ctx):} \\ \texttt{ if ctx[p(}\bar{x}\texttt{)]:} \\ \texttt{ return \{'success': True, 'context': ctx\}} \\ \texttt{ ctx[p(}\bar{x}\texttt{)] = true} \\ \texttt{ } T[\![q_1(\bar{X}_1), q_2(\bar{X}_2), ..., q_m(\bar{X_m})]\!] \\ \texttt{ return \{'success': False, 'context': ctx\}} \\ \texttt{ \}} \end{cases}$$

The if-check in the beginning signals the function to report success if it already part of the context. Otherwise, it is added to the context contingent on the success of the rule body.

VIII. Abstraction (dual rule-heads):

$$T[\![not_p(\bar{X}) : -q_1(\bar{X}_1), .., q_m(\bar{X}_m)]\!] = \begin{cases} \texttt{def not_p}(\bar{x}\texttt{,ctx):} \\ \quad \texttt{if ctx[p}(\bar{x})\texttt{]:} \\ \quad\quad \texttt{return \{`success':False,`context':ctx\}} \\ \quad T[\![q_1(\bar{X_1}), .., q_m(\bar{X_m})]\!] \\ \quad \texttt{return \{`success': False,`context':ctx\}} \end{cases}$$

If the positive term corresponding to the dual is already part of the context, then we report failure. Else, it would be a contradiction to have both the positive term and its NAF term in the same answer set (context).

6.3 Efficiency of the Synthesized Code

Since the synthesized code has the s(ASP) logic encoded within and due to the fact that it is imperative in nature, the execution time is expected to be faster. This is, in fact, the case. We ran the synthesized graph coloring code on complete graphs of size 3, 4, 5 respectively. A complete graph of size n is labelled K_n. The running times are shown in the table below. The programs were run on an Intel i7 Processor at 2.90 GHz with 8 GB RAM. Average of 10 runs is shown:

K_n	Graph Colorable				Graph Not Colorable			
	Python Basic	Python Optimized	s(ASP)	Speedup Optimized	Python Basic	Python Optimized	s(ASP)	Speedup Optimized
K_3	0.109s	0.043s	1.14s	26.51	0.109s	0.039s	0.359s	9.21
K_4	0.135s	0.05s	4.80s	96	0.129s	0.046s	3.330s	72.39
K_5	0.334s	0.071s	34.63s	487.64	0.444s	0.074s	62.295s	841.82
K_{10}	$\approx$ 3h	772.05s	>15h	Large	$\approx$ 3h	79.78s	> 15h	Large

The speedup as the graph sizes increase is significant. Although asymptotically both the synthesized code and the s(ASP) execution suffer from the same exponential time complexity (in the size of the input graph), the improvement in the constant factor cannot be ignored. This echoes the idea that program transformation by partial evaluation results in program speedup [7]. Although we have very minimal partial evaluation, we eliminate the NMR checks that s(ASP) appends to every query. The NMR checks are implicitly handled by reducing them to a simple check of asserting whether both a proposition and its negation are present in the same context (partial stable model). For larger graph sizes such as K_{10}, the imperative program found a solution in 3 hours in whereas s(ASP) did not produce even after 15 hours. This again confirms the effectiveness of our program transformation. Nonetheless, it should also be noted that s(ASP) is still experimental and can handle any answer set program. Another point that should be highlighted is that the imperative program just produces one stable model as opposed to producing all stable models. This is well suited for ASP programs that have at most one answer set. The top-down translation

can admit optimization in the imperative program by having the program maintain a set of inconsistent models in its computation. It can also be improved by user-provided heuristics. We leave this for future work. An optimization of the graph-coloring and n-queens can be found elsewhere [16].

6.4 Proof of Correctness The first proof obligation is to ensure that the synthesized program is adherent to the stable model semantics. For this, we need to ensure that the steps taken by the imperative algorithm correspond to the steps taken by s(ASP) given a query. The imperative program represents on-the-fly grounding. This is stated in the following theorem as follows:

Theorem 1 Goal-directed on-the-fly grounding is equivalent to goal-directed grounded execution.

Proof Outline The proof can be found in the paper on s(ASP) [11] □

Next we state two lemmas. The first lemma justifies having just the *main predicate* in the *context*, the second lemma proves the equivalence of the *forall* and *for-loop*. Let $\mathscr{P}$ represent the class of answer set programs from Section 5.

Lemma 1 For every program $\Pi \in \mathscr{P}$ that is satisfiable, every partial stable model consists of at least one grounded term of the *main predicate*. □

Lemma 2 The translated *for-loop* is equivalent to the *forall* . □

Theorem 2 For every query Q of *main predicate* of every program $\Pi \in \mathscr{P}$, the synthesized imperative program returns a *context* corresponding to some partial stable model of Q in Π. The program returns an empty context if Q is unsatisfiable in Π.

Proof Outline Follows directly from Theorem 1, Lemmas 1 and 2. □

Note: The proofs of Lemmas 1, 2 can be found elsewhere [16].

7 Conclusion

In this paper, we described a way to synthesize imperative code from (a class of) ASP program specification under the goal-directed operational semantics of s(ASP). The work reported here is the first step towards obtaining efficient implementation from high level specifications. Our eventual goal is to specify algorithms (e.g., inserting, deleting, and checking for membership of an element) for concurrent data structures (lists, queues, trees, heaps, etc.) as answer set programs and automatically be able to obtain the efficient, imperative versions of those algorithms. Achieving such a goal requires making extensive use of partial evaluation and semantics-preserving re-arrangement of goals in the current resolvent. It is part of our future work. Essentially, we should be able to take constraints and move them up in the resolvent to the earliest point where this constraint can be executed. Note that this is essential for efficiency since constraints could be specified globally as in this different version of the graph-coloring answer set program below:

```
color(X, C) :- node(X), color(C), not another_color(X, C).
another_color(X, C) :- node(X), color(C), node(Y), X != Y, color(Y,C).
:- color(X, C), color(Y, C), edge(X, Y).
```

Handling global constraints is closely related to dynamic consistency checking (DCC) in goal-directed answer set programs [12, 13]. DCC ensures only the necessary constraints, that are relevant to a rule and the possible bindings variables could take, are checked along with the rule-body. All other constraints not relevant to the rule can be ignored. This would also enable synthesis of programs specifying data structures and planning problems.

References

[1] Joaquin Arias et al. "Constraint answer set programming without grounding". In: *Theory and Practice of Logic Programming* 18.3-4 (2018), pp. 337–354.

[2] David Basin et al. "Synthesis of programs in computational logic". In: *Program Development in Computational Logic*. Springer, 2004, pp. 30–65.

[3] Zhuo Chen et al. "A Physician Advisory System for Chronic Heart Failure management". In: *TPLP* 16.5-6 (2016), pp. 604–618.

[4] Thomas Eiter. "Data integration and answer set programming". In: *LPNMR*. Springer. 2005, pp. 13–25.

[5] Martin Gebser et al. "Clasp: A conflict-driven answer set solver". In: *LPNMR*. Springer. 2007, pp. 260–265.

[6] M. Gelfond and Y. Kahl. *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach*. Cambridge Univ. Press, 2014.

[7] N. Jones. "Constant time factors do matter". In: *STOC'93:6-2-611*.

[8] Nicola Leone et al. "The DLV system". In: *European Workshop on Logics in Artificial Intelligence*. Springer. 2002, pp. 537–540.

[9] John W Lloyd. *Foundations of logic programming*. Springer, 1987.

[10] Z. Manna and R. J. Waldinger. "Towards automatic program synthesis". In: *Symp. on Semantics of Algo. Lang.* Springer. 1971, pp. 270–310.

[11] K. Marple, E. Salazar, and G. Gupta. "Computing stable models of normal logic programs without grounding". In: *arXiv:1709.00501* (2017).

[12] Kyle Marple and Gopal Gupta. "Dynamic consistency checking in goal-directed answer set programming". In: *TPLP* 14.4-5 (2014), pp. 415–427.

[13] Kyle Marple et al. "Goal-directed execution of answer set programs". In: *Proc. 14th PPDP Symposium*. ACM. 2012, pp. 35–44.

[14] Luke Simon et al. "Coinductive logic programming". In: *International Conference on Logic Programming*. Springer. 2006, pp. 330–345.

[15] S. Srivastava, S. Gulwani, and J. S. Foster. "From program verification to program synthesis". In: *ACM Sigplan Notices*. Vol. 45(1):313-326. 2010.

[16] Sarat Chandra Varanasi et al. "Synthesizing Imperative Code from Answer Set Programming Specifications (Extended)". In: UT Dallas CS Technical Report, 2019. URL: `www.utdallas.edu/~gupta/aspsynthesis.pdf`.

Logic-Based Synthesis of Fair Voting Rules Using Composable Modules

Karsten Diekhoff, Michael Kirsten, and Jonas Krämer

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
kirsten@kit.edu, {karsten.diekhoff,jonas.kraemer}@student.kit.edu

Abstract. Voting rules aggregate multiple individual preferences in order to make collective decisions. Commonly, these mechanisms are expected to respect a multitude of different fairness and reliability properties, e.g., to ensure that each voter's ballot accounts for the same proportion of the elected alternatives, or that a voter cannot change the election outcome in her favor by insincerely filling out her ballot. However, no voting rule is fair in all respects, and trade-off attempts between such properties often bring out inconsistencies, which makes the construction of arguably practical and fair voting rules non-trivial and error-prone.
In this paper, we present a formal systematic approach for the flexible synthesis of voting rules from composable core modules to respect such properties by construction. Formal composition rules guarantee resulting properties from properties of the individual components, which are of generic nature to be reused for various voting rules. We provide a prototypical logic-based implementation with proofs for a selected set of structures and composition rules within the theorem prover Isabelle/HOL. The approach can be readily extended in order to support many voting rules from the literature by extending the set of basic modules and composition rules. We exemplarily synthesize the well-known voting rule *sequential majority comparison* (SMC) from simple generic modules, and automatically produce a proof that SMC satisfies the fairness property monotonicity. Monotonicity is a well-known social-choice property that is easily violated by voting rules in practice.

Keywords: Social choice · Higher-order logic · Program synthesis · Modular design.

1 Introduction

In an election, voters cast ballots to express their individual preferences about eligible alternatives. From these individual preferences, a collective decision, i.e., a set of winning alternatives, is determined using a *voting rule*. Throughout the literature and existing elections, many different voting rules have been proposed and are in use, and exhibit each different behaviors and properties. Depending on the specific applications and regulations, voting rules are devised for a variety of different design goals towards carefully selected behaviors and properties. Imagine, for instance, one situation where a village wants to elect a local council,

or another one where a group of friends wants to choose a destination to go on vacation based on each of the friend's preferences. In the former case, the village might prefer to be represented by a larger council rather than having only few representatives who are elected by a majority, but are strongly disliked by everybody else. For the latter, it is clearly undesirable to choose multiple destinations, but rather settle for one so that the group can spend the vacation together.

Indeed, there is no general rule which caters for every requirement, and any voting rule shows paradoxical behavior for some voting situation [1]. The *axiomatic method* permits the analysis of desired behavior by formal properties. It advocates the use of rules that provide rigorous guarantees (which we call *properties*), and compares them based on guarantees that they do, or do not, satisfy. Voting rules can be characterized by formal properties to understand and predict their behavior, or check that they comply with the law. These properties capture very different requirements, e.g., principles that each vote is counted equally, that the winning alternatives are proportionally composed with respect to the composition of the voters, or that all winning alternatives are preferred over all other alternatives by a sufficient amount of voters. Devising voting rules towards such properties is generally cumbersome as their trade-off is inherently difficult and error-prone. Attempting to prove properties for specific voting rules often exhibits design errors, but is cumbersome as well [4]. As of yet, there exists no general formal process to systematically devise voting rules towards formal properties without being either error-prone or extremely cumbersome.

Contribution. In this paper, we present a formal systematic approach for the flexible synthesis of voting rules from compact composable modules with formal properties guaranteed by construction. Indeed, when taking an abstract view, many voting rules share similar structures such as, e.g., aggregating the individual votes by calculating the sum or some other aggregator function. Based on this observation, our approach enables flexible and intuitive synthesis of interesting voting rules from a small number of compositional structures. These structures exhibit precise and general interfaces such that their scope may easily be extended with further modules. We devise a general component type as well as special types, e.g., for aggregation functions, and compositional structures for revising choices from prior modules, as well as sequential, parallel and loop composition. The resulting formal properties, e.g., common social choice properties such as monotonicity or Condorcet consistency, are guaranteed from composing modules with given individual properties by rigorous composition rules. We demonstrate the logic-based application with proofs for a selected set of composition structures and rules, and composable modules within the theorem prover Isabelle/HOL [13]. Thereby, the approach is amenable both for external scrutiny as compositions are rigorously and compactly defined, and for an integration in larger automatic voting rule design or verification frameworks. As case study, we define composition rules for common social choice properties, and demonstrate a formal correct-by-construction synthesis of the well-known voting rule *sequential*

majority comparison (SMC). The synthesis produces a proof that SMC fulfills the monotonicity property using a set of basic modules.

Outline. The rest of this paper is structured as follows: Section 2 introduces formal concepts and definitions from social choice theory for our synthesis approach. We present core modules and structures in Section 3 and demonstrate the logic-based synthesis approach within Isabelle/HOL in Section 4. In Section 5, we apply our approach to the case study of synthesizing the monotone voting rule sequential majority comparison, and give an overview of related work to our approach in Section 6. We finally conclude and discuss future work in Section 7.

2 Concepts and Definitions from Social Choice Theory

We consider a fixed finite set $\mathcal{A}$ of eligible *alternatives* and a finite (possibly ordered) set $\mathscr{K}$ of *voters* (with cardinality k). In an election, each voter i casts a *ballot* $\succsim_i \in \mathcal{L}(\mathcal{A})$, which is a linear order[1], ranking the alternatives $\mathcal{A}$ according to i's preference. We collect all votes in a *profile*, i.e., a sequence $\succsim = (\succsim_1, \ldots, \succsim_k)$ of k ballots. Given $\mathcal{L}(\mathcal{A})$ the set of linear orders on $\mathcal{A}$, $\mathcal{L}(\mathcal{A})^k$ defines the set of all profiles on $\mathcal{A}$ of length k, i.e., for all voters, which lets us define $\mathcal{L}(\mathcal{A})^+$ as the set of all finite, nonempty profiles on $\mathcal{A}$. Hence, we have $\mathcal{L}(\mathcal{A})^+ = \bigcup_{k \in \mathbb{N}^+} \mathcal{L}(\mathcal{A})^k$, which is the input domain for a voting rule. Voting rules elect a nonempty subset $\mathcal{C}(\mathcal{A})$ of the alternatives as (possibly tied) winners, given $\mathcal{C}(X)$ denotes the set of all nonempty subsets of a set X.

Definition 1 (Voting Rule). *Given a finite set of alternatives $\mathcal{A}$, a* voting rule *f maps each possible profile $\succsim \in \mathcal{L}(\mathcal{A})^+$ to a nonempty set of winning alternatives $w \in \mathcal{C}(\mathcal{A})$:*

$$f : \mathcal{L}(\mathcal{A})^+ \to \mathcal{C}(\mathcal{A}).$$

In practice and in literature, a multitude of voting rules are in use. A common example is the function that returns all alternatives that are ranked at first position by a plurality of the voters, hence called *plurality voting*. Another common kind of voting rules assigns values for every ballot to each alternative according to her position occupied on the ballot, and elects the alternatives with the maximal score, i.e., sum of all such values for her. Such rules are called scoring rules, e.g., the Borda rule, where the value of an alternative on a ballot is equal to the number of alternatives ranked below her on that ballot.

Social Choice Properties. Within social choice theory, the axiomatic method has established a number of general fairness and reliability properties called (*axiomatic*) *social choice properties*. They formally capture intuitively desirable or in other ways useful properties to compare, evaluate, or characterize voting rules. Such properties are applicable in a general way as they are defined on

[1] A linear order is a transitive, complete, and antisymmetric relation.

abstract voting rules only with respect to profiles and returned sets of winning alternatives. For the sake of simplicity, the examples illustrated in this paper only address properties of universal nature, i.e., they require that all mappings of a given voting rule belong to some set of admissible ways, as formally described by the property of interest, for associating sets of winners to profiles. Besides properties which *functionally* limit the possible sets of winning alternatives for any one given profile, properties may also *relationally* limit combinations (of finite arity) of mappings, e.g., certain (hypothetical) changes of a profile may only lead to certain changes of the winning alternatives. Relational properties capture a voter's considerations such as how certain ways of (not) filling out her ballot may or may not affect the chances of winning for some alternatives.

Within this paper, we use *Condorcet consistency* and *monotonicity* as running examples. The functional property *Condorcet consistency* requires that if there is an alternative w that is the *Condorcet winner*, the rule elects w as unique winner. A Condorcet winner is an alternative that wins every pairwise majority comparison against all other alternatives, i.e., for any other alternative, there is a majority of voters who rank the Condorcet winner higher than that alternative.

Definition 2 (Condorcet Winner). *For a set of alternatives $\mathcal{A}$ and a profile $\succsim \in \mathcal{L}(\mathcal{A})^+$, an alternative $w \in \mathcal{A}$ is a* Condorcet winner *iff the following holds:*

$$\forall a \in \mathcal{A} \setminus \{w\} : |\{i \in \mathscr{K} : a \succsim_i w\}| < |\{i \in \mathscr{K} : w \succsim_i a\}|.$$

Note that, if a Condorcet winner exists, it is unique by the above definition.

Definition 3 (Condorcet Consistency). *For a set of alternatives $\mathcal{A}$, a voting rule f is* Condorcet consistent *iff for every profile $\succsim \in \mathcal{L}(\mathcal{A})^+$ and (if existing) the respective Condorcet winner $w \in \mathcal{A}$, the following holds:*

$$w \textit{ is } \text{Condorcet winner } \textit{for} \succsim \quad \Rightarrow \quad f(\succsim) = \{w\}.$$

For profiles where no Condorcet winner exists, the property imposes no requirements on the election outcome. The relational property *monotonicity* expresses that if a voter were to change her vote in favor of some alternative, the outcome could never change to the disdvantage of that alternative. Monotone voting rules are resistant to some forms of strategic manipulation where a voter might make their preferred alternative the (unique) winner by misrepresenting her actual preferences and assigning a higher rank to another alternative on her ballot. A voting rule is monotone iff for any profiles $\succsim$ and $\succsim'$ which are identical except for one alternative a that is ranked higher in $\succsim'$ (while preserving all remaining pairwise-relative rankings), the election of a in $\succsim$ always implies her election in $\succsim'$. We define this "ranking higher" as *lifting* an alternative.

Definition 4 (Lifting). *For a set of alternatives $\mathcal{A}$ and two profiles $\succsim, \succsim' \in \mathcal{L}(A)^k$, $\succsim'$ is obtained from $\succsim$ by* lifting *an alternative $a \in \mathcal{A}$ iff there exists a ballot $i \in [1, k]$ such that $\succsim_i \neq \succsim'_i$ and for each such i the following holds:*

i. There exists some alternative $x \in \mathcal{A}$ such that $x \succsim_i a$ and $a \succsim'_i x$, and

ii. we have $y \succsim_i z \Leftrightarrow y \succsim'_i z$ *for all other alternatives* $y, z \in \mathcal{A} \setminus \{a\}$.

We may thus define the monotonicity property as follows.

Definition 5 (Monotonicity). *For a set of alternatives* $\mathcal{A}$ *and an alternative* $a \in \mathcal{A}$*, a voting rule* f *is* monotone *iff for all profiles* $\succsim, \succsim' \in \mathcal{L}(\mathcal{A})^+$ *where* $\succsim'$ *is obtained from lifting* a *in* $\succsim$*, the following implication holds:*

$$a \in f(\succsim) \Rightarrow a \in f(\succsim').$$

3 Composable Modules and Compositional Structures

Component-Based Synthesis The approach in this paper consists of two structural and two semantic concepts, namely (i) *component types* that specify structural interfaces wherein components can be implemented, and (ii) *compositional structures* that specify structural contracts which combine components to create new components that are again composable. Moreover, semantic aspects for synthesizing concrete voting rules are addressed by (iii) *composition rules* that define semantic rules which compositions can contractually depend on in the sense that, if components fulfill a rule's requirements, the composition guarantees the semantics specified by the rule, as well as (iv) *composable modules* that define concrete semantics of either directly implemented or further decomposable modules from which other modules can be composed using the above-named rules. In the following, we elaborate our component types and compositional structures for composing voting rules as introduced in [9].

3.1 Electoral Modules

The structural foundation of our approach are *electoral modules*, a generalization of voting rules from Definition 1. Electoral modules are the principal component type within our component-based synthesis, and is more general than a voting rule in the sense that it does not need to make final decisions in the form of a partitioning the alternatives into only winning and losing alternatives. Instead, electoral modules partition alternatives into *elected*, *rejected* and *deferred* alternatives. Besides the additional set of deferred alternatives to let another module decide on them, electoral modules may receive any (possibly empty) subset $A \subseteq \mathcal{A}$, and may leave the set of winning (elected) alternatives empty.

Hence, an electoral module maps a (sub-)set of alternatives and a profile (that considers only the mentioned (sub-)set of alternatives) to three sets of alternatives that partition the received (sub-)set of alternatives. We say that a sequence of sets $s_1, \ldots, s_n$ partitions a set S if and only if S equals the union $\bigcup_{i \in [1,n]} s_i$ over all sets s_i for $i \in [1, n]$ and all their pairwise intersections are empty, i.e., $\forall i \neq j \in [1, n] : s_i \cap s_j = \emptyset$. Moreover, we demand that the profile passed to a module only addresses alternatives from the (sub-)set that the module receives. We define the domain $\mathcal{D}^{\mathcal{A}}_{\text{mod}}$ as $\mathcal{D}^{\mathcal{A}}_{\text{mod}} := \{(A, \succsim) \mid A \subseteq \mathcal{A}, \succsim \in \mathcal{L}(A)^+\}$ which contains all subsets of $\mathcal{A}$ paired with matching profiles, for the following definition:

Definition 6 (Electoral Module). *For the eligible alternatives $\mathcal{A}$ and a (sub-)set $A \subseteq \mathcal{A}$ an* electoral module *m is a function*

$$m : \mathcal{D}^{\mathcal{A}}_{\text{mod}} \to \mathcal{P}(\mathcal{A})^3$$

which maps a set of alternatives with a matching profile to the set-triple (e, r, d) of sets of elected*,* rejected*, and* deferred *alternatives such that*

$$(A, \succsim) \in \mathcal{D}^{\mathcal{A}}_{\text{mod}} \Rightarrow (m(A, \succsim) = (e, r, d) \text{ partitions } A).$$

For the sake of readability, we write $\mathcal{M}_{\mathcal{A}}$ for the set of electoral modules. We introduce the shorthands $m_e(A, \succsim)$, $m_r(A, \succsim)$, and $m_d(A, \succsim)$ respectively for the elected, rejected and deferred alternatives of an electoral module m for $(A, \succsim)$.

Moreover, we can easily translate voting rules to electoral modules by returning a triple of empty sets in case the module receives an empty set. Otherwise, we return an empty deferred set, an elected set with exactly the winning alternatives, and a rejected set with the complement of the winning alternatives (we remove the alternatives which are not contained in the received set of alternatives). Note that this also entails that social choice properties can be easily translated for electoral modules.

3.2 Sequential Composition

Sequential composition is a compositional structure for composing two electoral modules m, n into a new electoral module $(m \triangleright n)$ such that the second module n only decides on alternatives which m defers and cannot reduce the set of alternatives already elected or rejected by m. In this composition, n receives only m's deferred alternatives $m_d(A, \succsim)$ and a profile $\succsim_{|(m_d(A,\succsim))}$ which only addresses alternatives contained in $m_d(A, \succsim)$.

Definition 7 (Sequential Composition). *For any set of alternatives $\mathcal{A}$ and a (sub-)set $A \subseteq \mathcal{A}$, electoral modules $m, n \in \mathcal{M}_{\mathcal{A}}$ and input $(A, \succsim) \in \mathcal{D}^{\mathcal{A}}_{\text{mod}}$, we define the sequential composition function $(\triangleright) : \mathcal{M}^2_{\mathcal{A}} \to \mathcal{M}_{\mathcal{A}}$ as*

$$\begin{aligned}(m \triangleright n)(A, \succsim) := \quad & (m_e(A, \succsim) \cup n_e(m_d(A, \succsim), \succsim_{|(m_d(A,\succsim))}),\\ & m_r(A, \succsim) \cup n_r(m_d(A, \succsim), \succsim_{|(m_d(A,\succsim))}),\\ & n_d(m_d(A, \succsim), \succsim_{|(m_d(A,\succsim))}))\end{aligned}$$

3.3 Revision Composition

For situations where we want to revise the alternatives already elected by a prior module, e.g., for making them available in a sequential composition to apply a tie-breaking module for instance, the revision composition comes in handy. For an electoral module m, the revision composition of m reinterprets m's elected alternatives by removing them and attaching them to the previous deferred

alternatives, while leaving the rejected alternatives unchanged. Whereas the revision composition can also be achieved by parallel composition, this dedicated structure turned out to be beneficial in our implementation due to its frequent uses.

Definition 8 (Revision Composition). *For any set of alternatives $\mathcal{A}$ and a (sub-)set $A \subseteq \mathcal{A}$, electoral module $m \in \mathcal{M}_{\mathcal{A}}$, and input $(A, \succsim) \in \mathcal{D}^{\mathcal{A}}_{\text{mod}}$, we define the revision composition $(\downarrow) : \mathcal{M}_{\mathcal{A}} \to \mathcal{M}_{\mathcal{A}}$ as*

$$(m\downarrow)(A, \succsim) := (\emptyset,\ m_r(A, \succsim),\ m_e(A, \succsim) \cup m_d(A, \succsim)).$$

3.4 Parallel Composition

The parallel composition lets two electoral modules make two independent decisions for the given set of alternatives. Their two decisions are then aggregated by an *aggregator*, which is another component type that combines two set-triples of elected, rejected and deferred alternatives (as well as the set of alternatives) into a single such triple (we define the input domain $\mathcal{D}^{\mathcal{A}}_{\text{agg}}$ accordingly).

Definition 9 (Aggregator). *For a set of alternatives $\mathcal{A}$, a (sub-)set $A \subseteq \mathcal{A}$ and input $(A, p_1, p_2) \in \mathcal{D}^{\mathcal{A}}_{\text{agg}}$, an aggregator is a function*

$$agg : \mathcal{D}^{\mathcal{A}}_{\text{agg}} \to \mathcal{P}(\mathcal{A})^3 \text{ such that } agg(A, p_1, p_2) \text{ partitions } A.$$

A useful instance of such an aggregator is the max-aggregator $agg_{\max}$:

Definition 10 (Max-Aggregator). *Given two set-triples (e_1, r_1, d_1), (e_2, r_2, d_2) of elected (e), rejected (r) and deferred (d) alternatives, $agg_{\max}$ picks for each alternative a and the sets containing a the superior one of the two sets (assuming the order* e > d > r*).*

$$agg_{\max}((e_1, r_1, d_1),\ (e_2, r_2, d_2)) = \\ (e_1 \cup e_2, (r_1 \cup r_2) \setminus (e_1 \cup e_2 \cup d_1 \cup d_2), (d_1 \cup d_2) \setminus (e_1 \cup e_2))$$

Based on the notion of aggregators, we can now define the parallel composition as a function mapping two electoral modules m, n and an aggregator $agg \in \mathcal{G}_{\mathcal{A}}$ (the set of all aggregators) to a new electoral module $(m\ ||_{agg}\ n)$:

Definition 11 (Parallel Composition). *For a set of alternatives A and a (sub-)set $A \subseteq \mathcal{A}$, electoral modules m, n, and an aggregator agg we define the parallel composition $(||) : (\mathcal{M}_{\mathcal{A}} \times \mathcal{G}_{\mathcal{A}} \times \mathcal{M}_{\mathcal{A}}) \to \mathcal{M}_{\mathcal{A}}$ as*

$$(m\ ||_{agg}\ n)(A, \succsim) := agg(A,\ m(A, \succsim),\ n(A, \succsim)).$$

3.5 Loop Composition

Based on sequential composition (Section 3.2) of electoral modules, we define the more general loop composition to allow sequential compositions of dynamic length.

A loop composition ($m \circlearrowleft_t$) repeatedly composes an electoral module m sequentially with itself until either a fixed point is reached or a *termination condition* t, which is technically another component type that simply consists of a boolean predicate on a set-triple of alternatives, is satisfied.

The full definition can be found within the Isabelle/HOL theories provided with this paper.

3.6 A Simple Example

As a simple example, we illustrate the synthesis of a voting rule using the compositional structures defined above. Consider the well-known *Baldwin's rule*, which is a voting rule based on sequential elimination [2]. The rule repeatedly eliminates the alternative with the lowest Borda score (see Section 2) until only one alternative remains.

As basic modules, we use (a) a module that computes the Borda scores, rejects the alternative with the lowest such score, and defers the rest, as well as (b) a module that attaches all deferred alternatives to the elected set.

Moreover, we require a termination condition for a loop structure such that the loop of interest stops when the set of (deferred) alternatives has reached size one.

Therefore, Baldwin's rule can be obtained by

1. composing (a) by a loop structure with above mentioned termination condition, and
2. sequentially composing the loop composition with (b).

Moreover, loop composition can be used for many voting rules of a category called *tournament solutions*, where in multiple rounds, alternatives are compared to another regarding their rankings in the profile, and the winner of a comparison advances to the next round.

4 Logic-Based Synthesis Using Composition Rules

In the following, we describe how we model the concepts defined in Section 3 within a logic-based proof framework for synthesizing fair voting rules.

4.1 Isabelle and Higher-Order Logic (HOL)

We implemented and proved our logical concepts within the interactive theorem prover Isabelle/HOL [13]. The Isabelle/HOL system provides a generic infrastructure for implementing deductive systems in higher-order logics and enabling

to write tactics for human-readable and machine-checked proofs to show that the deductive conclusions are indeed correct. We decided to use Isabelle/HOL, because higher-order logic (HOL) allows to define very expressive, rigorous and general theorems. By this means, a theorem is –once proven correct within Isabelle– re-checked and confirmed by Isabelle/HOL within a few seconds every time the theorem is loaded. Proofs within Isabelle/HOL are based on the employed theories at the core of the Isabelle system. We made use of the possibility to define very general theorems to be reused for the synthesis of various voting rules and sorts of composition. Moreover, the framework allows for easy application and extension potentially within a larger framework for the automatic discovery and synthesis of voting rules, provided that the voting rule of interest can be composed from the given compositional rules and composable modules using the given composition structures and component types.

The definitions and theorems within our framework are mostly self-contained, i.e., for the most part they only rely on basic set theory as well as the theories of finite lists, relations, and order relations for defining the profiles and linear orders used within our notion of profiles and modules as seen in Listing 1. Therein, we introduce a handy type abbreviation (Line 1) for profiles which are lists of relations and therefrom define profiles on alternatives (Lines 3 to 4) based on the theory of order relations, and moreover finite profiles (Lines 5 to 6) which we use in a number of structures and concepts. Moreover, type abbreviations for results of electoral modules (Line 8), i.e., set-triples, are introduced, and electoral modules (Line 10) as defined in Section 3. We capture the partitioning with the two functions to express disjointness of the three sets in an electoral module result (Lines 20 to 21) and that their union yields the set of alternatives of the input (Lines 17 to 18). Finally, at the end of Listing 1, we can essentially define electoral modules on finite profiles and partitionings of the alternatives (Lines 23 to 25). We did not require any additional theories besides the ones provided off-the-shelf with the Isabelle system.

```
1  type_synonym 'a Profile = "('a rel) list"
2
3  definition profile_on :: "'a set ⇒ 'a Profile ⇒ bool" where
4    "profile_on A p ≡ (∀ b ∈ (set p). linear_order_on A b)"
5  abbreviation finite_profile :: "'a set ⇒ 'a Profile ⇒ bool" where
6  "finite_profile A p ≡ finite A ∧ profile_on A p"
7
8  type_synonym 'a Result = "'a set * 'a set * 'a set"
9
10 type_synonym 'a Electoral_module = "'a set ⇒ 'a Profile ⇒ 'a Result"
11
12 fun disjoint :: "'a Result ⇒ bool" where "disjoint (e, r, d) =
13   ((e ∩ r = {}) ∧
14    (e ∩ d = {}) ∧
15    (r ∩ d = {}))"
16
17 fun unify_to :: "'a set ⇒'a Result ⇒ bool"
18   where "unify_to A (e, r, d) ↔ (e ∪ r ∪ d = A)"
```

```
19
20 definition partition_of :: "'a set ⇒ 'a Result ⇒ bool" where
21   "partition_of A result  ≡ disjoint result ∧ unify_to A result"
22
23 definition electoral_module :: " 'a Electoral_module ⇒ bool"
24   where "electoral_module m ≡
25   ∀A p. finite_profile A p → partition_of A (m A p)"
```

Listing 1. Central Isabelle/HOL definitions for electoral modules.

As of yet, our logic-based synthesis framework comprises concepts and proofs for 18 composition rules with ten reusable auxiliary properties and eight properties which translate directly to common social choice properties from the literature. Thereof, we implemented the auxiliary properties and the monotonicity property with respective proofs within the Isabelle/HOL framework.

4.2 Compositional Synthesis based on Composition Rules

From devising composition rules and properties as described in the beginning of this section together with the component types and structures as described in Section 3, our framework now only requires a comparatively small set of basic components in order to construct interesting voting rules for the desired social choice properties which have been defined as properties and included in the rules for composing electoral modules. The power of our approach lies both in the generality of the composition rules and compositional structures such that various voting rules may be constructed for various properties, and in the reduction of complexity such that compositions for complex social choice properties can be defined by predominantly local composition rules in a step-by-step manner.

In general, the synthesis using composition rules works as follows: When we want to design a voting rule with a set of properties p from some basic components c and d which satisfy sets of properties p_c and p_d respectively, we might make use of a compositional structure X which guarantees that a composed module $m_c X m_d$ satisfies the properties p. Hence, we can design a desired voting rule by instantiating m_c and m_d by c and d respectively, which gives us the induced voting rule f_{cXd}.

One such example using structures from Section 3 and properties defined in this section is that p consists of the property Condorcet consistency, X is the sequential composition, p_c also consists of Condorcet consistency, and p_d is empty, i.e., we have no requirements for properties of any component d. On its own, this composition rule might not be very sensible, but may be used in combination with other composition rules to preserve Condorcet consistency of composed voting rules. A voting rule from the literature which is constructed in such a manner is *Black's rule*. Black's rule is a sequential composition of (a component which induces) the Condorcet rule and (a component which induces) the Borda rule.

5 Case Study

As a case study for demonstrating the applicability of our approach to existing voting rules and the merits of composition, we constructed the voting rule *sequential majority comparison* (SMC) from the literature (e.g., from Brandt et al. [5]), thereby producing a compositional proof that our constructed voting rule is monotone.

Sequential Majority Comparison (SMC). The voting rule of sequential majority comparison, also known as sequential pairwise majority, is simple enough for understanding, but still complex enough such that it demonstrates interesting properties such as monotonicity. Essentially, SMC fixes some (potentially arbitrary) order on all alternatives and then consecutively performs pairwise majority elections. We start by doing pairwise comparisons of the first and the second alternative, then compare the winner of this pairwise comparison to the third alternative, whose winner is then compared to the fourth alternative, and so on. SMC belongs to a category of voting rules called *tournament solutions*, for which we outline a possible construction for synthesis in the following.

Logic-Based Synthesis of Tournament Solution. As indicated in Section 3, loop composition appears sensible for *tournament solutions*, as a list of alternatives is processed by multiple rounds, whereof in each, the previously chosen alternative is compared to the next alternative on the list regarding their rankings in the profile, and the winner of a comparison advances to the next round.

To compare alternatives, we use any electoral module m which elects one alternative and rejects the rest (for example via plurality voting). To limit comparisons to two alternatives, we use the electoral module $pass^2_>$, which defers the two alternatives ranked highest in some fixed order $>$ and rejects the rest. Similarly, $drop^2_>$ rejects these two alternatives and defers the rest.

We can now describe a single comparison in our tournament as

$$c = (pass^2_> \triangleright m) \mid\mid_{agg_{\max}} drop^2_>$$

The first part of the parallel composition elects the winner of the current comparison and rejects all other alternatives. The second part defers all alternatives which are not currently being compared and therefore stay in the tournament.

The termination condition $t_{|d|=0}$ is satisfied iff the set of deferred alternatives passed to it is empty. Then we describe a single round of our tournament as

$$r = (c \circlearrowleft_{t_{|d|=0}}) \downarrow$$

Every single comparison elects a single alternative to advance to the next round and rejects the other. As long as alternatives are left, the next c compares the next two alternatives. If there ever is only a single alternative left, it advances to the next round automatically. At the end of the round, we need to revise to defer all winners to the next round instead of electing them.

Let m_{elect} be the electoral module which elects all alternatives passed to it and $t_{|d|=1}$ the termination condition that is satisfied when exactly one alternative is deferred. We can now define the whole tournament as

$$t = (r \circlearrowleft_{t_{|d|=1}}) \triangleright m_{\text{elect}}.$$

t repeats single rounds as long as there is more than one alternative left and then elects the single survivor.

Compositional Design of SMC. After having described a general approach for synthesizing tournament solutions, we give only structural information on the synthesis of SMC as the details are rather lengthy, but instead refer the reader to the Isabelle/HOL proofs.

We can construct SMC by combining six different basic components by using all of our composition structures, i.e., the sequential, parallel, loop, and revision structure, and thereby produce a proof that SMC is a monotone voting rule.

```
definition SMC :: "'a rel ⇒ 'a Electoral_module" where
  "SMC x ≡ let a = Max_aggregator; t = Defer_eq_condition 1 in
    ((((Pass_module 2 x) ▷ ((Plurality_module↓) ▷ (Pass_module 1 x))) ||a
      (Drop_module 2 x)) ↻t) ▷ Elect_module"

theorem SMC_sound:
  assumes order: "linear_order x"
  shows "electoral_module (SMC x)"

theorem SMC_monotone:
  assumes order: "linear_order x"
  shows "monotone (SMC x)"
```

Listing 2. The compositional synthesis of SMC in Isabelle/HOL.

The high-level compositional synthesis can be seen in Listing 2, where SMC stands for sequential majority comparison composed of a number of simple components. Each component, the largest of which is an electoral module inducing plurality voting (see Section 2), consists of not more than three lines in higher-order logic and we provided proofs within our Isabelle/HOL framework for easy reuse and modification for similar voting rules.

Moreover, Listing 2 shows the simplicity of the abstract proof obligations both that SMC is again an electoral module and satisfies the monotonicity property. Both tasks are proven fully modularly and are hence a direct result of SMC's composition, and is apt for an automated integration within a potential future logic-based synthesis assistant . We omit the proofs at this point, but they are available for download[2] and can be inspected and re-played for inspection and automatically checked using Isabelle/HOL. The full proof comprises 26 compositions using a set of six basic components within the theorem prover Isabelle/HOL.

[2] https://formal.iti.kit.edu/~kirsten/lopstr2019/theories.zip

6 Related Work

We base the core component type in our design framework on the electoral modules from the unified description of electoral systems in [7]. Therein, Grilli di Cortona et al. devise a complex component structure for describing hierarchical electoral systems with a focus on proportional voting rules including notions for electoral districts and concepts of proportionality. General informal advice on voting rule design is given by Taagepera [15]. Moreover, a first approach for composing voting rules in a limited setting is given by Narodytska et al. [12] that is readily expressible by our structures. Other work designs voting rules less modularly for statistically guaranteeing social choice properties by machine learning [16]. Prior modular approaches target verification [10] or declarative combinations of voting rules [6], but ignore social choice properties.

We have defined our compositional approach within Isabelle/HOL [13], a theorem prover for higher-order logic. Isabelle/HOL provides interactive theorem proving for rigorous systems design. Further work on computer-aided verification of social choice properties for voting rules using HOL4 has been done by Dawson et al. [8]. More light-weight approaches with some loss of generality, but the merit of generating counterexamples for failing properties has been devised by Beckert et al. [3] and Kirsten and Cailloux [11]. Therein, techniques for relational verification of more involved social choice properties have been applied. Another interesting approach has been followed by Pattinson and Schürmann [14], where voting rules are directly encoded into HOL rules within tactical theorem provers.

7 Conclusion

Within this work, we introduced an approach to systematically synthesize voting rules from compact composable modules to satisfy formal social choice properties. We devised composition rules for a selection of common social choice properties, such as monotonicity or Condorcet consistency, as well as for reusable auxiliary properties. These composition rules give by design formal guarantees that a synthesized voting rule fulfills the social choice property of interest as long as its components satisfy specific properties.

Our approach is applicable to the synthesis of a wide range of voting rules which use sequential or parallel modular structures, notably voting rules with tie-breakers, elimination procedures, or tournament structures. This includes well-known rules such as instant-runoff voting, Nanson's method, or sequential majority comparison (SMC). We synthesized SMC from simple components, which we presume to be reusable for synthesizing further rules, and automatically by construction synthesized a proof that SMC satisfies monotonicity from basic formal proofs for the structures, compositions and components which we compositionally synthesized. This case study and all required definitions were implemented and verified with the theorem prover Isabelle/HOL. Finally, our approach can be safely extended with additional modules, compositional structures, and rules, for integration into voting rule design or verification frameworks.

Outlook. So far, composition is realized mostly by transferring sets of deferred alternatives between modules. We also intend to inspect the more involved modular structures already incorporated in some more complicated voting rules, in order to achieve a more flexible notion of composition. This, however, also involves making more detailed assumptions on how exactly information is passed between modules, which might come with a loss of generality. This however, seems to be necessary for voting rules such as Single-Transferable Vote (STV), which are not composable for sensible social choice properties with our strong notion of locality in composition rules.

References

1. Arrow, K.J.: Social Choice and Individual Values. Yale University Press, third edn. (2012)
2. Baldwin, J.M.: The technique of the nanson preferential majority system of election. Transactions and Proceedings of the Royal Society of Victoria **39** (1926)
3. Beckert, B., Bormer, T., Kirsten, M., Neuber, T., Ulbrich, M.: Automated verification for functional and relational properties of voting rules. In: Sixth International Workshop on Computational Social Choice (COMSOC 2016) (2016)
4. Beckert, B., Goré, R., Schürmann, C.: Analysing vote counting algorithms via logic. In: Hutchison, D., et al. (eds.) Automated Deduction – CADE-24. LNCS, vol. 7898. Springer (2013)
5. Brandt, F., Conitzer, V., Endriss, U., Lang, J., Procaccia, A.D.: Handbook of Computational Social Choice. Cambridge University Press (2016)
6. Charwat, G.: Democratix: A declarative approach to winner determination. In: ADT. LNCS, vol. 9346. Springer (2015)
7. Grilli di Cortona, P., Manzi, C., Pennisi, A., Ricca, F., Simeone, B.: Evaluation and optimization of electoral systems. SIAM (1999)
8. Dawson, J.E., Goré, R., Meumann, T.: Machine-checked reasoning about complex voting schemes using higher-order logic. In: Haenni, R., et al. (eds.) 5th International Conference on E-Voting and Identity (VoteID 2015). LNCS, vol. 9269. Springer (2015)
9. Diekhoff, K., Kirsten, M., Krämer, J.: Formal property-oriented design of voting rules using composable modules. In: Venable, K., Pekec, S. (eds.) 6th International Conference on Algorithmic Decision Theory (ADT 2019). LNAI (2019), to appear.
10. Ghale, M.K., Goré, R., Pattinson, D., Tiwari, M.: Modular formalisation and verification of STV algorithms. In: Third International Joint Conference on Electronic Voting (E-Vote-ID 2018) (2018)
11. Kirsten, M., Cailloux, O.: Towards automatic argumentation about voting rules. In: Bringay, S., Mattioli, J. (eds.) 4ème conférence sur les Applications Pratiques de l'Intelligence Artificielle (APIA 2018), Nancy, France, July 2-6, 2018 (2018)
12. Narodytska, N., Walsh, T., Xia, L.: Combining voting rules together. In: De Raedt, L., et al. (eds.) 20th European Conference on Artificial Intelligence (ECAI 2012). vol. 242. IOS Press (2012)
13. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
14. Pattinson, D., Schürmann, C.: Vote counting as mathematical proof. In: Pfahringer, B., Renz, J. (eds.) 28th Australasian Joint Conference on Advances in Artificial Intelligence (AI 2015). LNCS, vol. 9457. Springer (2015)

15. Taagepera, R.: Designing electoral rules and waiting for an electoral system to evolve. The Architecture of Democracy: Institutional Design, Conflict Management, and Democracy in the Late Twentieth Century (2002)
16. Xia, L.: Designing social choice mechanisms using machine learning. In: Gini, M.L., et al. (eds.) International conference on Autonomous Agents and Multi-Agent Systems (AAMAS '13). IFAAMAS (2013)

Solving Proximity Constraints*

Temur Kutsia and Cleo Pau

RISC, Johannes Kepler University Linz, Austria

Abstract. Proximity relations are binary fuzzy relations that satisfy reflexivity and symmetry properties, but are not transitive. This relation induces the notion of distance between function symbols, which is further extended to terms. Given two terms, we aim at bringing them "sufficiently close" to each other, by finding an appropriate substitution. We impose no extra restrictions on proximity relations, allowing a term in unification to be close to two terms that themselves are not close to each other. Our unification algorithm works in two phases: first reducing the equation solving problem to constraints over sets of function symbols, and then solving the obtained constraints. Termination, soundness and completeness of both algorithms are shown. The unification problem has finite minimal complete set of unifiers.

1 Introduction

Proximity relations are reflexive and symmetric fuzzy binary relations. Introduced in [2], they generalize similarity relations (a fuzzy version of equivalence), by dropping transitivity. Proximity relations help to represent fuzzy information in situations, where similarity is not adequate, providing more flexibility in expressing vague knowledge. For unification, working modulo proximity or similarity means to treat different function symbols as if they were the same when the given relation asserts they are "close enough" to each other.

Unification for similarity relations was studied in [3, 4, 12] in the context of fuzzy logic programming. In [1], the authors extended the algorithm from [12] to full fuzzy signature (permitting arity mismatches between function symbols) and studied also anti-unification, a dual technique to unification.

A constraint logic programming schema with proximity relations has been introduced in [11]. The similarity-based unification algorithm from [12] was generalized for proximity relations in [5], where the authors introduced the notion of proximity-based unification under a certain restriction imposed on the proximity relation. The restriction requires that the same symbol can not be close to two symbols at the same time, when those symbols are not close to each other. One of them should be chosen as the proximal candidate to the given symbol. Essentially, proximal neighborhoods of two function symbols are treated as being either identical or disjoint. Looking at the proximity relation as an undirected graph, this restriction implies that maximal cliques in the graph are disjoint. From the unification point of view, it means that $p(x, x)$ can not be unified with

* Supported by Austrian Science Fund (FWF) under project 28789-N32.

$p(a, c)$ when a and c are not close to each other, even if there exists a b which is close both to a and c. Anti-unification for such kind of proximity relations, and its subalgorithm for computing all maximal partitions of a graph into maximal cliques have been considered in [8, 9].

In this paper we consider the general case of unification for a proximity relation without any restrictions, and develop an algorithm which computes a compact representation of the set of solutions. Considering neighborhoods of function symbols as finite sets, we work with term representation where in place of function symbols we permit neighborhoods or names. The latter are some kind of variables, which stand for unknown neighborhoods. The algorithm is split into two phases. In the first one, which is a generalization of syntactic unification for proximity relations, we produce a substitution together with a constraint over neighborhoods and their names. A crucial step in the algorithm is variable elimination, which is done not with a term to which a variable should be unified, but with a copy of that term with fresh names and variables. This step also introduces new neighborhood constraints to ensure that the copy of the term remains close to its original.

In the second phase, the constraint is solved by a constraint solving algorithm. Combining each solution from the second phase with the substitution computed in the first phase, we obtain a compact representation of the minimal complete set of unifiers of the original problem. We prove termination, soundness and completeness of both algorithms. To the best of our knowledge, this is the first detailed study of full-scale proximity-based unification.

In the rest of the paper, the necessary notions and terminology are introduced in Sect. 2, and both algorithms are developed and studied in Sect. 3. Due to space limits, long proofs are put in the appendix, which also contains some illustrative examples. The full version of the paper can be found in the technical report [10].

2 Preliminaries

Proximity relations. We define basic notions about proximity relations following [5]. A binary *fuzzy relation* on a set S is a mapping from $S \times S$ to the real interval $[0, 1]$. If $\mathcal{R}$ is a fuzzy relation on S and λ is a number $0 \leq \lambda \leq 1$, then the λ-*cut* of $\mathcal{R}$ on S, denoted $\mathcal{R}_\lambda$, is an ordinary (crisp) relation on S defined as $\mathcal{R}_\lambda := \{(s_1, s_2) \mid \mathcal{R}(s_1, s_1) \geq \lambda\}$. In the role of T-norm $\wedge$ we take the minimum.

A fuzzy relation $\mathcal{R}$ on a set S is called a *proximity relation* on S iff it is reflexive and symmetric. The λ-*proximity class of* $s \in S$ (a λ-*neighborhood* of s) is a set $\mathbf{pc}(s, \mathcal{R}, \lambda) = \{s' \mid \mathcal{R}(s, s') \geq \lambda\}$. When $\mathcal{R}$ and λ are fixed, we simply write $\mathbf{pc}(s)$.

Terms and substitutions. Given a set of variables $\mathcal{V}$ and a set of fixed arity function symbols $\mathcal{F}$, *terms* over $\mathcal{F}$ and $\mathcal{V}$ are defined as usual, by the grammar $t := x \mid f(t_1, \ldots, t_n)$, where $x \in \mathcal{V}$ and $f \in \mathcal{F}$ is n-ary.

The set of terms over $\mathcal{V}$ and $\mathcal{F}$ is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We denote variables by x, y, z, arbitrary function symbols by f, g, h, constants by a, b, c, and terms by s, t, r.

Substitutions are mappings from variables to terms, where all but finitely many variables are mapped to themselves. The identity substitution is denoted by Id. We use the usual set notation for substitutions, writing, e.g., a substitution σ as $\{x \mapsto \sigma(x) \mid x \neq \sigma(x)\}$. Substitution application and composition are defined in the standard way. We use the postfix notation for substitution application, e.g., $t\sigma$, and juxtaposition for composition, e.g., $\sigma\vartheta$ means the composition of σ and ϑ (the order matters).

Extended terms, extended substitutions, name-neighborhood mappings. Assume that $\mathcal{N}$ is a countable *set of names*, which are symbols together with associated arity (like function symbols). We use the letters $\mathrm{N}, \mathrm{M}, \mathrm{K}$ for them. It is assumed that $\mathcal{N} \cap \mathcal{F} = \emptyset$ and $\mathcal{N} \cap \mathcal{V} = \emptyset$.

Neighborhood is either a name, or a finite subset of $\mathcal{F}$, where all elements have the same arity. Since in the construction of extended terms below neighborhoods will behave like function symbols, we will use the letters F and G to denote them. $\mathit{arity}(\mathsf{F})$ is defined as the arity of elements of F. The set of all neighborhoods is denoted by Nb.

An *extended term* (or, shortly, an *X-term*) t over $\mathcal{F}$, $\mathcal{N}$, and $\mathcal{V}$ is defined by the grammar:

$$\mathsf{t} := x \mid \mathsf{F}(\mathsf{t}_1, \ldots, \mathsf{t}_n), \text{ where } \mathit{arity}(\mathsf{F}) = n.$$

The set of X-terms is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{N}, \mathcal{V})$. X-terms, in which every neighborhood set is a singleton, are called *singleton X-terms* or, shortly, *SX-terms*. Slightly abusing the notation, we assume that a term (i.e., an element of $\mathcal{T}(\mathcal{F}, \mathcal{V})$) is a special case of an SX-term (as an SX-term without names), identifying a function symbol f with the singleton neighborhood $\{f\}$. We will use this assumption in the rest of the paper.

The notion of *head* is defined as $\mathit{head}(x) = x$ and $\mathit{head}(\mathsf{F}(\mathsf{t}_1, \ldots, \mathsf{t}_n)) = \mathsf{F}$.

The set of variables (resp. names) occurring in an X-term t is denoted by $\mathcal{V}(\mathsf{t})$ (resp. $\mathcal{N}(\mathsf{t})$). Approximate extended equations (X-equations) are pairs of X-terms.

The notion of substitution (and the associated relations and operations) are extended to X-terms straightforwardly. We use the term "*X-substitution*" and denote them by upright Greek letters $\mathsf{\mu}$, $\mathsf{\nu}$, and $\mathsf{\xi}$. When we want to emphasize that we are talking about substitutions for terms, we use the letters σ, ϑ, and φ.

The restriction of an X-substitution $\mathsf{\mu}$ to a set of variables V is denoted by $\mathsf{\mu}|_V := \{x \mapsto x\mathsf{\mu} \mid x \in \mathit{dom}(\mathsf{\mu}) \cap V\}$.

A *name-neighborhood mapping* $\Phi : \mathcal{N} \longrightarrow \mathsf{Nb} \setminus \mathcal{N}$ is a finite mapping from names to non-name neighborhoods (i.e., finite sets of function symbols of the same arity) such that if $\mathrm{N} \in \mathit{dom}(\Phi)$ (where dom is the domain of mapping), then $\mathit{arity}(\mathrm{N}) = \mathit{arity}(\Phi(\mathrm{N}))$. They are also represented as finite sets, writing Φ as $\{\mathrm{N} \mapsto \Phi(\mathrm{N}) \mid \mathrm{N} \in \mathit{dom}(\Phi)\}$.

A name-neighborhood mapping Φ can *apply* to an X-term t, resulting in an X-term $\Phi(\mathsf{t})$, which is obtained by replacing each name N in t by the neighborhood $\Phi(\mathrm{N})$. The *application of Φ to a set of X-equations P,* denoted by $\Phi(P)$, is a set of equations obtained from P by applying Φ to both sides of each equation in P.

Proximity relations over X-terms. The proximity relation $\mathcal{R}$ is defined on the set $\mathsf{Nb} \cup \mathcal{V}$ (where neighborhoods are assumed to be nonempty) in such a way that it satisfies the following conditions (in addition to reflexivity and symmetry):

(a) $\mathcal{R}(\mathsf{F}, \mathsf{G}) = 0$ if $\mathit{arity}(\mathsf{F}) \neq \mathit{arity}(\mathsf{G})$;
(b) $\mathcal{R}(\mathsf{F}, \mathsf{G}) = \min\{\mathcal{R}(f, g) \mid f \in \mathsf{F}, g \in \mathsf{G}\}$, if $\mathsf{F} = \{f_1, \ldots, f_n\}$, $\mathsf{G} = \{g_1, \ldots, g_m\}$, and $\mathit{arity}(\mathsf{F}) = \mathit{arity}(\mathsf{G})$;
(c) $\mathcal{R}(\mathrm{N}, \mathsf{F}) = 0$, if $\mathsf{F} \notin \mathcal{N}$.
(d) $\mathcal{R}(\mathrm{N}, \mathrm{M}) = 0$, if $\mathrm{N} \neq \mathrm{M}$;
(e) $\mathcal{R}(x, y) = 0$, if $x \neq y$ for all $x, y \in \mathcal{V}$.
(f) $\mathcal{R}(\mathsf{F}, \mathsf{G})$ is undefined, if $\mathsf{F} = \emptyset$ or $\mathsf{G} = \emptyset$.

We write $\mathsf{F} \approx_{\mathcal{R},\lambda} \mathsf{G}$ if $\mathcal{R}(\mathsf{F}, \mathsf{G}) \geq \lambda$. Note that for $\mathsf{F} = \{f_1, \ldots, f_n\}$ and $\mathsf{G} = \{g_1, \ldots, g_m\}$, $\mathsf{F} \approx_{\mathcal{R},\lambda} \mathsf{G}$ is equivalent to $\mathcal{R}(f, g) \geq \lambda$ for all $f \in \mathsf{F}$ and $g \in \mathsf{G}$. It is easy to see that the obtained relation is again a proximity relation. Furthermore, it can be extended to X-terms (which do not contain the empty neighborhood):

1. $\mathcal{R}(\mathsf{s}, \mathsf{t}) := 0$ if $\mathcal{R}(\mathit{head}(\mathsf{s}), \mathit{head}(\mathsf{t})) = 0$.
2. $\mathcal{R}(\mathsf{s}, \mathsf{t}) := 1$ if $\mathsf{s} = \mathsf{t}$ and $\mathsf{s}, \mathsf{t} \in \mathcal{V}$.
3. $\mathcal{R}(\mathsf{s}, \mathsf{t}) := \mathcal{R}(\mathsf{F}, \mathsf{G}) \wedge \mathcal{R}(\mathsf{s}_1, \mathsf{t}_1) \wedge \cdots \wedge \mathcal{R}(\mathsf{s}_n, \mathsf{t}_n)$, if $\mathsf{s} = \mathsf{F}(\mathsf{s}_1, \ldots, \mathsf{s}_n)$, $\mathsf{t} = \mathsf{G}(\mathsf{t}_1, \ldots, \mathsf{t}_n)$.
4. $\mathcal{R}(\mathsf{s}, \mathsf{t})$ is not defined, if s or t contains the empty neighborhood $\emptyset$.

Two X-terms s and t are $(\mathcal{R}, \lambda)$-*close* to each other, written $\mathsf{s} \simeq_{\mathcal{R},\lambda} \mathsf{t}$, if $\mathcal{R}(\mathsf{s}, \mathsf{t}) \geq \lambda$. We say that s is $(\mathcal{R}, \lambda)$-*more general than* t and write $\mathsf{s} \precsim_{\mathcal{R},\lambda} \mathsf{t}$, if there exists an substitution σ such that $\mathsf{s}\sigma \simeq_{\mathcal{R},\lambda} \mathsf{t}$.

Neighborhood equations, unification problems. We introduce the notions of problems we would like to solve.

Definition 1 (Neighborhood equations). *Given $\mathcal{R}$ and λ, an* $(\mathcal{R}, \lambda)$-neighborhood equation *is a pair of neighborhoods, written as $\mathsf{F} \approx^{?}_{\mathcal{R},\lambda} \mathsf{G}$. The question mark indicates that it has to be solved.*

A name-neighborhood mapping Φ is a solution *of an $(\mathcal{R}, \lambda)$-neighborhood equation $\mathsf{F} \approx^{?}_{\mathcal{R},\lambda} \mathsf{G}$ if $\Phi(\mathsf{F}) \simeq_{\mathcal{R},\lambda} \Phi(\mathsf{G})$. The notation implies that $\mathcal{R}(\Phi(\mathsf{F}), \Phi(\mathsf{G}))$ is defined, i.e., neither $\Phi(\mathsf{F})$ nor $\Phi(\mathsf{G})$ contains the empty neighborhood.*

An $(\mathcal{R}, \lambda)$-neighborhood constraint is a finite set of $(\mathcal{R}, \lambda)$-neighborhood equations. A name-neighborhood mapping Φ is a solution of an $(\mathcal{R}, \lambda)$-neighborhood constraint C if it is a solution of every $(\mathcal{R}, \lambda)$-neighborhood equation in C.

We shortly write "an $(\mathcal{R}, \lambda)$-solution to C" instead of "a solution to an $(\mathcal{R}, \lambda)$-neighborhood constraint C".

To each X-term t we associate a set of SX-terms $\mathsf{Singl}(\mathsf{t})$ defined as follows:

$$\mathsf{Singl}(x) := \{x\},$$
$$\mathsf{Singl}(\mathrm{N}(\mathsf{t}_1, \ldots, \mathsf{t}_n)) := \{\mathrm{N}(\mathsf{s}_1, \ldots, \mathsf{s}_n) \mid \mathsf{s}_i \in \mathsf{Singl}(\mathsf{t}_i),\ 1 \leq i \leq n\}.$$
$$\mathsf{Singl}(\mathsf{F}(\mathsf{t}_1, \ldots, \mathsf{t}_n)) := \{f(\mathsf{s}_1, \ldots, \mathsf{s}_n) \mid f \in \mathsf{F},\ \mathsf{s}_i \in \mathsf{Singl}(\mathsf{t}_i),\ 1 \leq i \leq n\}.$$

The notation extends to substitutions as well:

$$\mathsf{Singl}(\mu) := \{\vartheta \mid x\vartheta \in \mathsf{Singl}(x\mu) \text{ for all } \ x \in \mathcal{V}\}.$$

Definition 2 (Approximate X-unification). *Given $\mathcal{R}$ and λ, a finite set P of $(\mathcal{R},\lambda)$-equations between X-terms is called an $(\mathcal{R},\lambda)$-X-unification problem. A mapping-substitution pair (Φ,μ) is called an* $(\mathcal{R},\lambda)$-solution *of an $(\mathcal{R},\lambda)$-X-equation $\mathsf{t} \simeq^{?}_{\mathcal{R},\lambda} \mathsf{s}$, if $\Phi(\mathsf{t}\mu) \simeq_{\mathcal{R},\lambda} \Phi(\mathsf{s}\mu)$. An $(\mathcal{R},\lambda)$-solution of P is a pair (Φ,μ) which solves each equation in P.*

If (Φ,μ) is an $(\mathcal{R},\lambda)$-solution of P, then the X-substitution $\Phi(\mu)$ is called an $(\mathcal{R},\lambda)$-X-unifier *of P.*

SX-unification problems, SX-solutions and SX-unifiers are defined analogously. For unification between terms, we do not use any prefix, talking about unification problems, solutions, and unifiers.

Instead of writing "a $\cdots$-unifier of an $(\mathcal{R},\lambda)$-unification problem P", we often shortly say "an $(\mathcal{R},\lambda)$-$\cdots$-unifier of P".

The notion of more generality for substitutions is defined with the help of syntactic equality: μ is *more general* than ν, written $\mu \preceq \nu$, if there exists ξ such that $\mu\xi = \nu$.

Remark 1. Note that we did not use proximity in the definition of this notion. The reason is that in our definition, $\preceq$ is a quasi-order and preserves good properties of unifiers. In particular, if μ is an $(\mathcal{R},\lambda)$-X-unifier of P, then so is any ν for which $\mu \preceq \nu$ holds.

If we defined this notion as "$\mu \precsim_{\mathcal{R},\lambda} \nu$ if there exists a substitution ξ such that $x\mu\nu \simeq_{\mathcal{R},\lambda} x\xi$ for all x", it would not be a quasi-order, because it is not transitive. Therefore, it might happen that μ is an $(\mathcal{R},\lambda)$-X-unifier of P, but ν with $\mu \precsim_{\mathcal{R},\lambda} \nu$ is not. A simple example is $P = \{x \simeq^{?}_{\mathcal{R},\lambda} a\}$ and $\mathcal{R}_\lambda = \{(a,b),(b,c)\}$. Then $\mu = \{x \mapsto b\}$ is an $(\mathcal{R},\lambda)$-unifier of P, but $\nu = \{x \mapsto c\}$ is not. However, $\mu \precsim_{\mathcal{R},\lambda} \nu$.

Two substitutions μ and ν are called *equigeneral* iff $\mu \preceq \nu$ and $\nu \preceq \mu$. In this case we write $\mu \simeq \nu$. It is an equivalence relation.

Unification between terms. Our unification problem will be formulated between terms, and we would like to have a characterization of the set of its unifiers. (Extended terms and substitutions will play a role in the formulating of algorithms, proving their properties, and representing the mentioned unifier set compactly.)

Definition 3 (Complete set of unifiers). *Given a proximity relation $\mathcal{R}$, a cut value λ, and an $(\mathcal{R},\lambda)$-proximity unification problem P, the set of substitutions Σ is a* complete set of $(\mathcal{R},\lambda)$-unifiers *of P if the following conditions hold:*

Soundness: *Every substitution $\sigma \in \Sigma$ is an $(\mathcal{R},\lambda)$-unifier of P.*
Completeness: *For any $(\mathcal{R},\lambda)$-unifier ϑ of P, there exists $\sigma \in \Sigma$ such that $\sigma \preceq \vartheta$.*

Σ is *a minimal complete set of unifiers* of P if it is its complete set of unifiers and, in addition, the following condition holds:

Minimality: No two elements in Σ are comparable with respect to $\preceq$: For all $\sigma,\vartheta \in \Sigma$, if $\sigma \preceq \vartheta$, then $\sigma = \vartheta$.

Under this definition, $\{x \simeq_{\mathcal{R},0.5} b\}$ for $\mathcal{R}(a,b) = 0.6$, $\mathcal{R}(b,c) = 0.5$ has a minimal complete set of unifiers $\{\{x \mapsto a\}, \{x \mapsto b\}, \{x \mapsto c\}\}$. Note that the substitutions $\{x \mapsto a\}$ and $\{x \mapsto b\}$ are $\preceq$-incomparable, but $\precsim_{\mathcal{R},\lambda}$-comparable. The same is true for $\{x \mapsto a\}$ and $\{x \mapsto c\}$.

Given an approximate unification problem P, our goal is to obtain a compact representation of its minimal complete set of $(\mathcal{R},\lambda)$-unifiers. The representation will be constructed as a set of X-unifiers $\mathcal{U}^X_{\mathcal{R},\lambda}(P) = \{\Phi_1(\mu), \ldots, \Phi_n(\mu)\}$. The algorithms below construct this representation.

3 Solving unification problems

We start with a high-level view of the process of solving an approximate unification problem $s \simeq^?_{\mathcal{R},\lambda} t$ between terms s and t (we omit $\mathcal{R}$ and λ below):

- First, we treat the input equation as an SX-equation and apply rules of the pre-unification algorithm. Pre-unification works on SX-equations. It either fails (in this case the input terms are not unifiable) or results in a neighborhood constraint C and a substitution μ over $\mathcal{T}(\emptyset, \mathcal{N}, \mathcal{V})$.
- Next, we solve C by the neighborhood constraint solving algorithm. If the process fails, then the input terms are not unifiable. Otherwise, we get a finite set of name-neighborhood mappings $\mathcal{M} = \{\Phi_1, \ldots, \Phi_n\}$. Note that Φ's do not necessarily map names to singleton sets here.
- For each $\Phi_i \in \mathcal{M}$, the pair (Φ_i, μ) solves the original unification problem, i.e., the X-substitution $\Phi_i(\mu)$ is an X-unifier of it.
- From the obtained set $\{\Phi_1(\mu), \ldots, \Phi_n(\mu)\}$ of computed $(\mathcal{R},\lambda)$-X-unifiers of s and t we can construct a minimal complete set of unifiers $mcsu_{\mathcal{R},\lambda}(s,t)$ of s and t as the set $mcsu_{\mathcal{R},\lambda}(s,t) = \mathsf{Singl}(\Phi(\mu_1)) \cup \cdots \cup \mathsf{Singl}(\Phi(\mu_1))$.

Hence, the algorithm consists of two phases: pre-unification and constraint solving. They are described in separate subsections below.

3.1 Pre-unification rules

We start with the definition of a technical notion needed later:

Definition 4. *We say that a set of SX-equations* $\{x \simeq^?_{\mathcal{R},\lambda} \mathsf{t}\} \uplus P$ *contains an* occurrence cycle *for the variable* x *if* $\mathsf{t} \notin \mathcal{V}$ *and there exist SX-term-pairs* $(x_0, \mathsf{t}_0), (x_1, \mathsf{t}_1), \ldots, (x_n, \mathsf{t}_n)$ *such that* $x_0 = x$, $\mathsf{t}_0 = \mathsf{t}$, *for each* $0 \le i \le n$ P *contains an equation* $x_i \simeq^?_{\mathcal{R},\lambda} \mathsf{t}_i$ *or* $\mathsf{t}_i \simeq^?_{\mathcal{R},\lambda} x_i$, *and* $x_{i+1} \in \mathcal{V}(\mathsf{t}_i)$ *where* $x_{n+1} = x_0$.

Lemma 1. *If a set of SX-equations* P *contains an occurrence cycle for some variable, then it has no* $(\mathcal{R},\lambda)$*-solution for* P *for any* $\mathcal{R}$ *and* λ.

Proof. The requirement that neighborhoods of different arity are not $(\mathcal{R},\lambda)$-close to each other guarantees that an SX-term can not be $(\mathcal{R},\lambda)$-close to its proper subterm. Therefore, equations containing an occurrence cycle can not have an $(\mathcal{R},\lambda)$-solution. □

In the rules below we will use the *renaming function* $\rho : \mathcal{T}(\mathcal{F}, \mathcal{N}, \mathcal{V}) \to \mathcal{T}(\mathcal{N}, \mathcal{V})$. Applied to a term, ρ gives its fresh copy, obtained by replacing each occurrence of a symbol from $\mathcal{F} \cup \mathcal{N}$ by a new name and each variable occurrence by a fresh variable. For instance, if the term is $f(\mathrm{N}(a, x, x, f(a)))$, where $f, a \in \mathcal{F}$ and $\mathrm{N} \in \mathcal{N}$, then $\rho(f(\mathrm{N}(a, x, x, f(a)))) = \mathrm{N}_1(\mathrm{N}_2(\mathrm{N}_3, x_1, x_2, \mathrm{N}_4(\mathrm{N}_5)))$, where $\mathrm{N}_1, \mathrm{N}_2, \mathrm{N}_3, \mathrm{N}_4, \mathrm{N}_5 \in \mathcal{N}$ are new names and x_1, x_2 are new variables.

Given $\mathcal{R}$ and λ, an *equational* $(\mathcal{R}, \lambda)$*-configuration* is a triple $P; C; \mu$, where

- P is a finite set of $(\mathcal{R}, \lambda)$-SX-equations. It is initialized with the unification equation between the original terms;
- C is a $(\mathcal{R}, \lambda)$-neighborhood constraint;
- μ is an X-substitution over $\mathcal{T}(\emptyset, \mathcal{N}, \mathcal{V})$, initialized by Id. It serves as an accumulator, keeping the pre-unifier computed so far.

The pre-unification algorithm takes given terms s and t, creates the initial configuration $\{s \simeq^?_{\mathcal{R},\lambda} t\}; \emptyset; Id$ and applies the rules given below exhaustively.

The rules are very similar to the syntactic unification algorithm with the difference that here the function symbol clash does not happen unless their arities differ, and variables are not replaced by other variables until the very end. (The notation $\overline{exp_n}$ in the rules below abbreviates the sequence $exp_1, \ldots, exp_n$.)

(Tri) Trivial: $\{x \simeq^?_{\mathcal{R},\lambda} x\} \uplus P; C; \mu \Longrightarrow P; C; \mu$.

(Dec) Decomposition:

$\{\mathsf{F}(\overline{\mathsf{s}_n}) \simeq^?_{\mathcal{R},\lambda} \mathsf{G}(\overline{\mathsf{t}_n})\} \uplus P; C; \mu \Longrightarrow \{\overline{\mathsf{s}_n \simeq^?_{\mathcal{R},\lambda} \mathsf{t}_n}\} \cup P; \{\mathsf{F} \approx^?_{\mathcal{R},\lambda} \mathsf{G}\} \cup C; \mu$,
where each of F and G is a name or a function symbol treated as a singleton neighborhood.

(VE) Variable Elimination:

$\{x \simeq^?_{\mathcal{R},\lambda} \mathsf{t}\} \uplus P; C; \mu \Longrightarrow \{\mathsf{t}' \simeq^?_{\mathcal{R},\lambda} \mathsf{t}\} \cup P\{x \mapsto \mathsf{t}'\}; C; \mu\{x \mapsto \mathsf{t}'\}$,
where $\mathsf{t} \notin \mathcal{V}$, there is no occurrence cycle for x in $\{x \simeq^?_{\mathcal{R},\lambda} \mathsf{t}\} \uplus P$, and $\mathsf{t}' = \rho(\mathsf{t})$.

(Ori) Orient: $\{\mathsf{t} \simeq^?_{\mathcal{R},\lambda} x\} \uplus P; C; \mu \Longrightarrow \{x \simeq^?_{\mathcal{R},\lambda} \mathsf{t}\} \cup P; C; \mu$, if $\mathsf{t} \notin \mathcal{V}$.

(Cla) Clash: $\mathsf{F}(\overline{\mathsf{s}_n}) \simeq^?_{\mathcal{R},\lambda} \mathsf{G}(\overline{\mathsf{t}_m})\} \uplus P; C; \mu \Longrightarrow \bot$, where $n \neq m$.

(Occ) Occur Check: $\{x \simeq^?_{\mathcal{R},\lambda} \mathsf{t}\} \uplus P; C; \mu \Longrightarrow \bot$,
if there is an occurrence cycle for x in $\{x \simeq^?_{\mathcal{R},\lambda} \mathsf{t}\} \uplus P$.

(VO) Variables Only:

$\{x \simeq^?_{\mathcal{R},\lambda} y, \overline{x_n \simeq^?_{\mathcal{R},\lambda} y_n}\}; C; \mu \Longrightarrow \{\overline{x_n \simeq^?_{\mathcal{R},\lambda} y_n}\}\{x \mapsto y\}; C; \mu\{x \mapsto y\}$.

Informally, in the (VE) rule, we imitate the structure of t in t' by ρ, replace x by t', and then try to bring t' close to t by solving the equation $\mathsf{t}' \simeq^?_{\mathcal{R},\lambda} \mathsf{t}$.

Theorem 1 (Termination of pre-unification). *The pre-unification algorithm terminates either with $\bot$ or with a configuration of the form $\emptyset; C; \mu$.*

Proof. The rules (Tri) and (Dec) strictly decrease the size of P. (Ori) does not changes the size, but strictly decreases the number of equations of the form $\mathsf{t} \simeq^?_{\mathcal{R},\lambda} x$, where $\mathsf{t} \notin \mathcal{V}$. (VO) stands separately, because once it starts applying, no other rule is applicable and (VO) itself is terminating. So are the failure rules.

To see what is decreased by (VE), we need some definitions. First, with each variable occurring in the initial unification problem we associate the set of its copies, which is initialized with the singleton set consisting of the variables themselves. For instance, if the problem contains variables x, y, z, and u, we will have four copy sets: $\{x\}$, $\{y\}$, $\{z\}$, and $\{u\}$. Rules may add new copies to these sets, or remove some copies from them. However, the copy sets themselves are fixed. None of them will be removed, and no new copy sets will be created.

In the process of rule applications, we will maintain a directed acyclic graph, whose vertices are labeled by copy sets, and there is an edge from a vertex V_1 to a vertex V_2 if we have encountered an equation of the form $x \simeq^?_{\mathcal{R},\lambda} \mathsf{t}$ such that $x \in V_1$ and t contains a variable $y \in V_2$.

One can notice that the graph is a variable dependency graph. If it contains a cycle, the algorithm stops with failure by the (Occ) rule. From the beginning, the vertices (i.e. the copy sets) are isolated. In the process of rule applications, assume that we reach a configuration that is transformed by the (VE) rule, applied to an equation $x \simeq^?_{\mathcal{R},\lambda} \mathsf{t}$, where t contains variables y, z', and z'' (the latter two are copies of z). The rule creates a fresh copy of t, which contains copies of variables: $\rho(y)$, $\rho(z')$, and $\rho(z'')$. They are added to the corresponding copy sets (graph vertices): $\rho(y)$ to the copy set of y, and $\rho(z')$ and $\rho(z'')$ to the copy set of z. Let us call those vertices V_y and V_z. Besides, if there was no edge connecting the vertex V_x (containing the copy set of x) to the vertices V_y and V_z, the edges are created. Finally, x is removed from V_x.

Hence, after each application of the (VE) rule, the copy set decreases in one vertex V (in the example above it is V_x), and stays unchanged in all vertices that are not reachable from V. We say in this case that the graph measure decreases. This ordering can be seen as a generalization of lexicographic ordering to graphs. It is well-founded. (VE) strictly decreases it, and the other rules have no effect.

Hence, if we take the lexicographic combination of three measures: copy set dags, the size of the set of equations, and the number of equations with non-variable term in the left and variable in the right, each rule except (VO) strictly decrease it. After finitely many steps, either failure will occur, or one reaches the variable-only equations, which are solved in finitely many steps by (VO). Then it stops with the configuration $\emptyset; C; \mu$. □

We say that a mapping-substitution pair (Φ, ν) is a *solution of an equational* $(\mathcal{R}, \lambda)$*-configuration* $P; C; \mu$ if the following conditions hold:

- (Φ, ν) is an $(\mathcal{R}, \lambda)$-solution of P;
- Φ is an $(\mathcal{R}, \lambda)$-solution of C;
- For each $x \in dom(\mu)$, we have $x\nu = x\mu\nu$ (syntactic equality).

Lemma 2. *1. If $P; C; \mu \Longrightarrow \bot$ by* (Cla) *or* (Occ) *rules, then $P; C; \mu$ does not have a solution.*

2. Let $P_1; C_1; \mu_1 \Longrightarrow P_2; C_2; \mu_2$ be a step performed by a pre-unification rule (except (Cla) *and* (Occ)*). Then every solution of $P_2; C_2; \mu_2$ is a solution of $P_1; C_1; \mu_1$.*

Proof. See the appendix. □

Theorem 2 (Soundness of pre-unification). *Let s and t be two terms, such that the pre-unification algorithm gives $\{s \simeq^?_{\mathcal{R},\lambda} t\}; \emptyset; Id \Longrightarrow^* \emptyset; C; \mu$. Let Φ be an $(\mathcal{R}, \lambda)$-solution of C. Then the X-substitution $\Phi(\mu)$ contains no names and is an $(\mathcal{R}, \lambda)$-X-unifier of $\{s \simeq^?_{\mathcal{R},\lambda} t\}$.*

Proof. Note that (Φ, μ) is an $(\mathcal{R}, \lambda)$-solution of $\emptyset; C; \mu$, since Φ solves C. From $\{s \simeq^?_{\mathcal{R},\lambda} t\}; \emptyset; Id \Longrightarrow^* \emptyset; C; \mu$, by induction on the length of the derivation, using Lemma 2, we get that (Φ, μ) is a solution of $\{s \simeq^?_{\mathcal{R},\lambda} t\}; \emptyset; Id$.

Since s and t do not contain names, Φ has no effect on them: $\Phi(s\mu) = s\Phi(\mu)$ and $\Phi(t\mu) = t\Phi(\mu)$. From these equalities and $\Phi(s\mu) \simeq_{\mathcal{R},\lambda} \Phi(t\mu)$ we get $s\Phi(\mu) \simeq_{\mathcal{R},\lambda} t\Phi(\mu)$, which implies that $\Phi(\mu)$ is an $(\mathcal{R}, \lambda)$-X-solution of $\{s \simeq^?_{\mathcal{R},\lambda} t\}$. By construction of pre-unification derivations, all the names in μ are in the domain of Φ. Hence, $\Phi(\mu)$ contains no names. □

Corollary 1. *Let s and t be two terms, such that the pre-unification algorithm gives $\{s \simeq^?_{\mathcal{R},\lambda} t\}; \emptyset; Id \Longrightarrow^* \emptyset; C; \mu$. Let Φ be an $(\mathcal{R}, \lambda)$-solution of C. Then every substitution in $\mathsf{Singl}(\Phi(\mu))$ is an $(\mathcal{R}, \lambda)$-unifier of s and t.*

Proof. Direct consequence of Theorem 2 and the definition of Singl. Note that $\mathsf{Singl}(\Phi(\mu))$ in this case is a set of substitutions, not a set of SX-substitutions, because $\Phi(\mu)$ does not contain names. □

For any solution of an approximate unification problem, we can not always compute a solution which is more general than the given one. For instance, for the problem $x \simeq^?_{\mathcal{R},\lambda} y$ we compute $\{x \mapsto y\}$, but the problem might have a solution, e.g., $\{x \mapsto a, y \mapsto b\}$ (when a and b are close to each other). Strictly speaking, $\{x \mapsto y\}$ is not more general than $\{x \mapsto a, y \mapsto b\}$, because there is no substitution σ such that $\{x \mapsto y\}\sigma = \{x \mapsto a, y \mapsto b\}$, but we have the relation $\{x \mapsto a, y \mapsto b\} \in \mathsf{Singl}(\{x \mapsto y\}\{y \mapsto \{a, b\}\}) = \mathsf{Singl}(\{x \mapsto \{a, b\}, y \mapsto \{a, b\}\})$.

The Completeness Theorem below proves a more general statement. It states that for any solution of an approximate unification problem, we can always compute a solution which is more general than a solution close to the given one.

Theorem 3 (Completeness of pre-unification). *Let an substitution ϑ be an $(\mathcal{R}, \lambda)$-unifier of two terms s and t. Then any maximal derivation that starts at $\{s \simeq^?_{\mathcal{R},\lambda} t\}; \emptyset; Id$ must end with an equational configuration $\emptyset; C; \mu$, such that for some $(\mathcal{R}, \lambda)$-solution Φ of C, which maps names to singleton neighborhoods, and some substitution $\sigma \simeq_{\mathcal{R},\lambda} \vartheta$, we have $\Phi(\mu|_{\mathcal{V}(s)\cup\mathcal{V}(t)}) \preceq \sigma$.*

Proof. The full proof can be found in the appendix. Here we sketch the idea.

Since $\{s \simeq^?_{\mathcal{R},\lambda} t\}$ is solvable, the derivation can not end with $\bot$ by soundness of pre-unification. Since for every equation there is a rule, and the algorithm terminates, the final configuration should have a form $\emptyset; C; \mu$.

Let V be $\mathcal{V}(s) \cup \mathcal{V}(t)$. In the construction of Φ, we need the proposition:

Proposition: Assume $P_1; C_1; \mu_1 \Longrightarrow P_2; C_2; \mu_2$ is a single step rule application in the above mentioned derivation. Let Φ_1 be a name-neighborhood mapping which maps names occurring in μ_1 to singleton neighborhoods such that equations in $\Phi_1(P_1)$ and in $\Phi_1(C_1)$ remain solvable. Assume that there exists a substitution $\sigma_1 \simeq_{\mathcal{R},\lambda} \vartheta$ such that $\Phi_1(\mu_1|_V) \preceq \sigma_1$. Then there exist a substitution $\sigma_2 \simeq_{\mathcal{R},\lambda} \vartheta$ and a name-neighborhood mapping Φ_2 which maps names occurring in μ_2 to singleton neighborhoods such that equations in $\Phi_2(P_2)$ and in $\Phi_2(C_2)$ remain solvable, and $\Phi_2(\mu_2|_V) \preceq \sigma_2$.

In the constructed derivation, for the initial configuration $P_0; C_0; \mu_0 = \{s \simeq^?_{\mathcal{R},\lambda} t\}; \emptyset; Id$ we take $\Phi_0 = \emptyset$. Then $\Phi_0(P_0) = P_0 = \{s \simeq^?_{\mathcal{R},\lambda} t\}$ and $\Phi_0(C_0) = \emptyset$ are solvable, and $\Phi_0(\mu_0|_V) = Id \preceq \vartheta$. By applying the proposition iteratively, we get that for the final configuration $\emptyset; C; \mu$, there exists Φ which maps names to singleton neighborhoods such that $\Phi(C)$ is solvable. By the way how Φ is constructed, we have $dom(\Phi) = \mathcal{N}(\mu)$, but $\mathcal{N}(\mu) = \mathcal{N}(C)$. Hence, $\mathcal{N}(\Phi(C)) = \emptyset$ and its solvability means that it is already solved (trivially solvable). It implies that Φ is a solution of C. Besides, again by an iterative application of the proposition, we show the existence of a $\sigma \simeq_{\mathcal{R},\lambda} \vartheta$ such that $\Phi(\mu|_V) \preceq \sigma$. □

By Theorem 2, $\Phi(\mu|_{\mathcal{V}(s)\cup\mathcal{V}(t)})$ in the completeness theorem is an X-unifier of the given problem. Then by the definition of $\preceq$, σ there is also a unifier. From Theorem 2 and Theorem 3, by definition of Singl we get the following result:

Theorem 4. *Given $\mathcal{R}$, λ and two terms s and t, let the pre-unification algorithm produce a derivation $\{s \simeq^?_{\mathcal{R},\lambda} t\}; \emptyset; Id \Longrightarrow^+ \emptyset; C; \mu$. Let $\{\Phi_1, \ldots, \Phi_n\}$ be a complete set of solutions of C, restricted to $\mathcal{N}(C)$. Then the set $\mathsf{Singl}(\Phi_1(\mu)) \cup \cdots \cup \mathsf{Singl}(\Phi_1(\mu))$ is a minimal complete set of unifiers of s and t.*

We introduce a neighborhood constraint solving algorithm in the next section. But before that we illustrate the pre-unification rules with a couple of examples:

Example 1. Let $s = p(x, y, x)$ and $t = q(f(a), g(d), y)$. Then the pre-unification algorithm stops with the configuration $\emptyset; C, \mu$, where $C = \{p \approx^?_{\mathcal{R},\lambda} q$, $\mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f$, $\mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a$, $\mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} g$, $\mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d$, $\mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3$, $\mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\}$ and $\mu = \{x \mapsto \mathrm{N}_1(\mathrm{N}_2), y \mapsto \mathrm{N}_3(\mathrm{N}_4)\}$.

Assume that for the given λ-cut, the proximity relation consists of pairs $\mathcal{R}_\lambda = \{(a, b), (b, c), (c, d), (a, b'), (b', c'), (c', d), (f, g), (p, q)\}$. The obtained constraint can be solved, e.g., by the name-neighborhood mappings $\Phi = [\mathrm{N}_1 \mapsto \{f, g\}, \mathrm{N}_2 \mapsto \{b\}, \mathrm{N}_3 \mapsto \{f, g\}, \mathrm{N}_4 \mapsto \{c\}]$ and $\Phi' = [\mathrm{N}_1 \mapsto \{f, g\}, \mathrm{N}_2 \mapsto \{b'\}, \mathrm{N}_3 \mapsto \{f, g\}, \mathrm{N}_4 \mapsto \{c'\}]$. From them and μ we can get the sets $\Phi(\mu)$ and $\Phi'(\mu)$ of $(\mathcal{R}, \lambda)$-unifiers of s and t.

If we did not have the VO rule and allowed the use of VE rule instead, we might have ended up with the unification problem $\{y \simeq^?_{\mathcal{R},\lambda} f(a), y \simeq^?_{\mathcal{R},\lambda} g(d)\}$, which does not have a solution, because the neighborhoods of a and d do not have a common element. Hence, we would have lost a solution.

Example 2. Let $s = p(x, x)$ and $t = q(f(y, y), f(a, c))$. The pre-unification algorithm stops with $\emptyset; P; \mu$, where $P = \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{M}\}$ and $\mu = \{x \mapsto \mathrm{N}_1(\mathrm{N}_2, \mathrm{N}_3),\ y_1 \mapsto \mathrm{N}_2,\ y_2 \mapsto \mathrm{N}_3,\ y \mapsto \mathrm{M}\}$. Let $\mathcal{R}_\lambda = \{(a, a_1), (a_1, b), (b, c_1), (c_1, c), (p, q)\}$. Then C is solved by $\Phi = [\mathrm{N}_1 \mapsto \{f\},\ \mathrm{N}_2 \mapsto \{a_1\},\ \mathrm{M} \mapsto \{b\},\ \mathrm{N}_3 \mapsto \{c_1\}]$ and $\Phi(\mu|_{\mathcal{V}(s)\cup\mathcal{V}(t)})$ contains only one element, an $(\mathcal{R}, \lambda)$-unifier $\sigma = \{x \mapsto f(a_1, c_1),\ y \mapsto b\}$ of s and t. Indeed, $s\sigma = p(f(a_1, c_1), f(a_1, c_1)) \simeq_{\mathcal{R},\lambda} q(f(b, b), f(a, c)) = t\sigma$.

This example illustrates the necessity of introducing a fresh variable for *each occurrence* of a variable by the renaming function in the VE rule. If we used the same new variable, say y', for both occurrences of y in $f(y, y)$ (instead of using y_1 and y_2 as above), we would get the configuration $\emptyset; \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\}; \{x \mapsto \mathrm{N}_1(\mathrm{N}_2, \mathrm{N}_2),\ y' \mapsto \mathrm{N}_2,\ y \mapsto \mathrm{N}_2\}$. But for the given $\mathcal{R}_\lambda$, the constraint $\{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\}$ does not have a solution (because the neighborhoods of a and c are not close to each other). Hence, we would lose a unifier.

3.2 Rules for neighborhood constraints

Let Φ be a *name-neighborhood mapping*. The *combination* of two mappings Φ and Ψ, denoted by $\Phi \odot \Psi$, is defined as

$$\begin{aligned}\Phi \odot \Psi := {} & \{\mathrm{N} \mapsto \Phi(\mathrm{N}) \mid \mathrm{N} \in dom(\Phi) \setminus dom(\Psi)\} \cup \\ & \{\mathrm{N} \mapsto \Psi(\mathrm{N}) \mid \mathrm{N} \in dom(\Psi) \setminus dom(\Phi)\} \cup \\ & \{\mathrm{N} \mapsto \Phi(\mathrm{N}) \cap \Psi(\mathrm{N}) \mid \mathrm{N} \in dom(\Psi) \cap dom(\Phi)\}.\end{aligned}$$

We call Φ and Ψ *compatible*, if $(\Phi \odot \Psi)(\mathrm{N}) \neq \emptyset$ for all $\mathrm{N} \in dom(\Phi \odot \Psi)$. Otherwise they are *incompatible*.

A *constraint configuration* is a pair $C; \Phi$, where C is a set of $(\mathcal{R}, \lambda)$-neighborhood constraints to be solved, and Φ is a name-neighborhood mapping (as a set of rules), representing the $(\mathcal{R}, \lambda)$-solution computed so far. We say that Ψ is an $(\mathcal{R}, \lambda)$-*solution of a constraint configuration* $C; \Phi$ if Ψ is an $(\mathcal{R}, \lambda)$-solution to C, and Ψ and Φ are compatible.

The constraint simplification algorithm $\mathcal{CS}$ transforms constraint configurations, exhaustively applying the following rules ($\bot$ indicates failure):

(FFS) Function Symbols: $\{f \approx^?_{\mathcal{R},\lambda} g\} \uplus C; \Phi; \Longrightarrow C; \Phi;$ if $\mathcal{R}(f, g) \geq \lambda$.

(NFS) Name vs Function Symbol:

$\{\mathrm{N} \approx^?_{\mathcal{R},\lambda} g\} \uplus C; \Phi \Longrightarrow C; \Phi \odot \{\mathrm{N} \mapsto \mathbf{pc}(g, \mathcal{R}, \lambda)\}$.

(FSN) Function Symbol vs Name: $\{g \approx^?_{\mathcal{R},\lambda} \mathrm{N}\} \uplus C; \Phi \Longrightarrow \{\mathrm{N} \approx^?_{\mathcal{R},\lambda} g\} \cup C; \Phi$.

(NN1) Name vs Name 1:

$\{\mathrm{N} \approx^?_{\mathcal{R},\lambda} \mathrm{M}\} \uplus C;\ \Phi \Longrightarrow C;\ \Phi \odot \{\mathrm{N} \mapsto \{f\}, \mathrm{M} \mapsto \mathbf{pc}(f, \mathcal{R}, \lambda)\}$,
where $\mathrm{N} \in dom(\Phi)$, $f \in \Phi(\mathrm{N})$.

(NN2) Name vs Name 2: $\{\mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}\} \uplus C;\ \Phi \Longrightarrow \{\mathrm{N} \approx^?_{\mathcal{R},\lambda} \mathrm{M}\} \cup C;\ \Phi$,
where $\mathrm{M} \notin dom(\Phi)$, $\mathrm{N} \in dom(\Phi)$.

(Fail1) Failure 1: $\{f \approx^?_{\mathcal{R},\lambda} g\} \uplus C;\ \Phi \Longrightarrow \bot$, if $\mathcal{R}(f, g) < \lambda$.

(Fail2) Failure 2: $C;\ \Phi \Longrightarrow \bot$, if there exists $\mathrm{N} \in dom(\Phi)$ with $\Phi(\mathrm{N}) = \emptyset$.

The NN1 rule causes branching, generating n branches where n is the number of elements in $\Phi(\mathrm{N})$. (Remember that by definition, the proximity class of each symbol is finite.) When the derivation does not fail, the terminal configuration has the form $\{\mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{M}_1, \ldots, \mathrm{N}_n \approx^?_{\mathcal{R},\lambda} \mathrm{M}_n\};\ \Phi$, where for each $1 \leq i \leq n$, $\mathrm{N}_i, \mathrm{M}_i \notin dom(\Phi)$. Such a constraint is trivially solvable.

Theorem 5. *The constraint simplification algorithm $\mathcal{CS}$ is terminating.*

Proof. With each configuration $C; \Phi$ we associate a complexity measure, which is a triple of natural numbers (n_1, n_2, n_3): n_1 is the number of symbols occurrences in C, n_2 is the number of equations of the form $g \approx^?_{\mathcal{R},\lambda} \mathrm{N}$ in C, and n_3 is the number of equations of the form $\mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}$ in C, where $\mathrm{M} \notin dom(\Phi)$ and $\mathrm{N} \in dom(\Phi)$. Measures are compared by the lexicographic extension of the ordering $>$ on natural numbers. It is a well-founded ordering. The rules (FFS), (NFS), (NN1) decrease n_1. The rule (FSN) does not change n_1 and decreases n_2. The rule (NN2) does not change n_1 and n_2 and decreases n_3. The failing rules cause termination immediately. Hence, each rule reduces the measure or terminates. It implies the termination of the algorithm. □

In the statements below, we assume $\mathcal{R}$ and λ to be given and the problems are to be solved with respect to them.

Lemma 3. *1. If $C; \Phi \Longrightarrow \bot$ by* (Fail1) *or* (Fail2) *rules, then (C, Φ) does not have an $(\mathcal{R}, \lambda)$-solution.*
2. Let $C_1; \Phi_1 \Longrightarrow C_2; \Phi_2$ be a step performed by a constraint solving (nonfailing) rule. Then any $(\mathcal{R}, \lambda)$-solution of $C_1; \Phi_2$ is also an $(\mathcal{R}, \lambda)$-solution of $C_1; \Phi_1$.

Proof. 1. For (Fail1), the lemma follows from the definition of $(\mathcal{R}, \lambda)$-solution. For (Fail2), no Ψ is compatible with Φ which maps a name to the empty set.
2. The lemma is straightforward for (FFS), (FSN), and (NN2). To show it for (NFS), we take Ψ, which solves $C;\ \Phi \odot \{\mathrm{N} \mapsto \mathbf{pc}(g, \mathcal{R}, \lambda)\}$. By definition of $\odot$, we get $\Psi(\mathrm{N}) \subseteq \mathbf{pc}(g, \mathcal{R}, \lambda)$. But then Ψ is a solution to $\{\mathrm{N} \approx^?_{\mathcal{R},\lambda} g\} \uplus C;\ \Phi$. To show the lemma holds for (NN1), we take a solution Ψ of $C;\ \Phi \odot \{\mathrm{N} \mapsto \{f\}, \mathrm{M} \mapsto \mathbf{pc}(f, \mathcal{R}, \lambda)\}$. It implies that $\Psi(\mathrm{N}) = \{f\}$ and $\Psi(\mathrm{M}) \subseteq \mathbf{pc}(f, \mathcal{R}, \lambda)$. But then we immediately get that Ψ solves $\{\mathrm{N} \approx^?_{\mathcal{R},\lambda} \mathrm{M}\} \uplus C;\ \Phi$. □

Theorem 6 (Soundness of $\mathcal{CS}$). *Let C be an $(\mathcal{R},\lambda)$-neighborhood constraint such that $\mathcal{CS}$ produces a maximal derivation $C;\emptyset \Longrightarrow^* C';\Phi$. Then Φ is an $(\mathcal{R},\lambda)$-solution of $C \setminus C'$, and C' is a set of constraints between names which is trivially $(\mathcal{R},\lambda)$-satisfiable.*

Proof. If a neighborhood equation is not between names, there is a rule in $\mathcal{CS}$ which applies to it. Hence, a maximal derivation can not stop with a C' that contains such an equation. As for neighborhood equations between names, only two rules deal with them: (NN1) and (NN2). But they apply only if at least one of the involved names belongs to the domain of the corresponding mapping. Hence, it can happen that an equation of the form $\mathrm{N} \approx^?_{\mathcal{R},\lambda} \mathrm{M}$ is never transformed by $\mathcal{CS}$. When the algorithm stops, such equations remain in C' and are trivially solvable. We can remove C' from each configuration in $C;\emptyset \Longrightarrow^* C';\Phi$ without affecting any step, getting a derivation $C \setminus C';\emptyset \Longrightarrow^* \emptyset;\Phi$. Obviously, Φ is an $(\mathcal{R},\lambda)$-solution of $\emptyset;\Phi$. By induction on the length of the derivation and Lemma 3, we get that Φ is an $(\mathcal{R},\lambda)$-solution of $C \setminus C';\emptyset$ and, hence, it solves $C \setminus C'$. □

Remark 2. When a neighborhood constraint C is produced by the pre-unification algorithm, then every maximal $\mathcal{CS}$-derivation starting from $C;\emptyset$ ends either in $\bot$ or in the pair of the form $\emptyset;\Phi$. This is due to the fact that the VE rule (which introduces names in pre-unification problems) and the subsequent decomposition steps always produce chains of neighborhood equations of the form $\mathrm{N}_1 \approx_{\mathcal{R},\lambda} \mathrm{N}_2, \mathrm{N}_2 \approx_{\mathcal{R},\lambda} \mathrm{N}_3, \ldots, \mathrm{N}_n \approx_{\mathcal{R},\lambda} f$, $n \geq 1$, for the introduced N's and for some f.

Theorem 7 (Completeness of $\mathcal{CS}$). *Let C be an $(\mathcal{R},\lambda)$-neighborhood constraint produced by the pre-unification algorithm, and Φ be one of its solutions. Let $dom(\Phi) = \{\mathrm{N}_1,\ldots,\mathrm{N}_n\}$. Then for each n-tuple $c_1 \in \Phi(\mathrm{N}_1),\ldots,c_n \in \Phi(\mathrm{N}_n)$ there exists a $\mathcal{CS}$-derivation $C;\emptyset \Longrightarrow^* \emptyset;\Psi$ with $c_i \in \Psi(\mathrm{N}_i)$ for each $1 \leq i \leq n$.*

Proof. We fix $c_1,\ldots,c_n$ such that $c_1 \in \Phi(\mathrm{N}_1),\ldots,c_n \in \Phi(\mathrm{N}_n)$.

First, note that $dom(\Phi)$ coincides with $\mathcal{N}(C)$. It is implied by the assumption that C is produced by pre-unification, and Remark 2 above.

The desired derivation is constructed recursively, where the important step is to identify a single inference. To see how such a single step is made, we consider a configuration $C_i;\Phi_i$ in this derivation ($i \geq 0$, $C_0 = C$, $\Phi_0 = \emptyset$). We have $dom(\Phi_i) \subseteq dom(\Phi)$. During construction, we will maintain the following two invariants for each $i \geq 0$ (easy to check that they hold for $i = 0$):

(I1) Φ is an $(\mathcal{R},\lambda)$-solution of (C_i,Φ_i), and
(I2) for all $1 \leq j \leq n$, if $\mathrm{N}_j \in dom(\Phi_i)$, then $c_j \in \Phi_i(\mathrm{N}_j)$.

We consider the following cases:

- C_i contains an equation of the form $f \approx^?_{\mathcal{R},\lambda} g$. By **(I1)**, $\mathcal{R}(f,g) \geq \lambda$. Then we make the (FFS) step with $f \approx^?_{\mathcal{R},\lambda} g$, obtaining $C_{i+1} = C_i \setminus \{f \approx^?_{\mathcal{R},\lambda} g\}$ and $\Phi_{i+1} = \Phi_i$. Obviously, **(I1)** and **(I2)** hold also for the new configuration.
- Otherwise, assume C_i contains an equation $\mathrm{N}_k \approx^?_{\mathcal{R},\lambda} g$, where $1 \leq k \leq n$. Since Φ solves C, Φ_i, we have $\Phi_i(\mathrm{N}) \neq \emptyset$ for all $\mathrm{N} \in dom(\Phi_i)$ and there is only

one choice to make the step: the (NFS) rule. It gives $C_{i+1} = C_i \setminus \{\mathrm{N}_k \approx^?_{\mathcal{R},\lambda} g\}$ and $\Phi_{i+1} = \Phi_i \odot \{\mathrm{N}_k \mapsto \mathbf{pc}(g, \mathcal{R}, \lambda)\}$. Since Φ solves, in particular, $\mathrm{N}_k \approx^?_{\mathcal{R},\lambda} g$, we have $\Phi(\mathrm{N}_k) \subseteq \mathbf{pc}(g, \mathcal{R}, \lambda)$ and, hence, $c_k \in \mathbf{pc}(g, \mathcal{R}, \lambda)$. First, assume $\mathrm{N}_k \notin \mathit{dom}(\Phi_i)$. Then $\Phi_{i+1}(\mathrm{N}_k) = \mathbf{pc}(g, \mathcal{R}, \lambda)$ and both **(I1)** and **(I2)** hold for $i+1$. Now, assume $\mathrm{N}_k \in \mathit{dom}(\Phi_i)$. Then $\Phi_{i+1}(\mathrm{N}_k) = \Phi_i(\mathrm{N}_k) \cap \mathbf{pc}(g, \mathcal{R}, \lambda)$. Besides, **(I2)** implies $c_k \in \Phi_i(\mathrm{N}_k)$. Hence, $c_k \in \Phi_{i+1}(\mathrm{N}_k)$, which implies that **(I2)** holds for $i+1$. From $c_k \in \Phi_{i+1}(\mathrm{N}_k)$ and $c_k \in \Phi(\mathrm{N}_k)$ we get that Φ is compatible with Φ_{i+1}. Moreover, Φ solves C_i, therefore, it solves C_{i+1}. Hence, Φ solves C_{i+1}; Φ_{i+1} and **(I1)** holds for $i+1$ as well.

– Otherwise, assume C_i contains an equation of the form $\mathrm{N}_k \approx^?_{\mathcal{R},\lambda} N_j$, where $1 \leq k, j \leq n$ and $\mathrm{N}_k \in \mathit{dom}(\Phi_i)$. By **(I1)**, we have $\mathrm{N}_k, \mathrm{N}_j \in \mathit{dom}(\Phi)$, $\Phi(\mathrm{N}_k) \cap \Phi(\mathrm{N}_j) \neq \emptyset$, and $\Phi(\mathrm{N}_k) \cap \Phi_i(\mathrm{N}_k) \neq \emptyset$. By **(I2)**, we have $c_k \in \Phi_i(\mathrm{N}_k)$. But since $c_k \in \Phi(\mathrm{N}_k)$, we have $c_k \in \Phi(\mathrm{N}_k) \cap \Phi_i(\mathrm{N}_k)$. We make the step with (NN1) rule, choosing the mapping $\mathrm{N}_k \mapsto \{c_k\}$. It gives $C_{i+1} = C_i \setminus \{\mathrm{N}_k \approx^?_{\mathcal{R},\lambda} \mathrm{N}_j\}$ and $\Phi_{i+1} = \Phi_i \odot \{\mathrm{N}_k \mapsto \{c_k\}, \mathrm{N}_j \mapsto \mathbf{pc}(c_k, \mathcal{R}, \lambda)\}$.
To see that **(I1)** holds for $i+1$, the only nontrivial thing is to check that Φ and Φ_{i+1} are compatible. For this, $\Phi_{i+1}(\mathrm{N}_k) \cap \Phi(\mathrm{N}_k) \neq \emptyset$ and $\Phi_{i+1}(\mathrm{N}_j) \cap \Phi(\mathrm{N}_j) \neq \emptyset$ should be shown.
Proving $\Phi_{i+1}(\mathrm{N}_k) \cap \Phi(\mathrm{N}_k) \neq \emptyset$: By construction, $\Phi_{i+1}(\mathrm{N}_k) = \{c_k\}$. By assumption, $c_k \in \Phi(\mathrm{N}_k)$. Hence, $\Phi_{i+1}(\mathrm{N}_k) \cap \Phi(\mathrm{N}_k) \neq \emptyset$.
Proving $\Phi_{i+1}(\mathrm{N}_j) \cap \Phi(\mathrm{N}_j) \neq \emptyset$: Since Φ solves $\mathrm{N}_k \approx^?_{\mathcal{R},\lambda} N_j$ and $c_k \in \Phi(\mathrm{N}_k)$, we have $\Phi(\mathrm{N}_j) \subseteq \mathbf{pc}(c_k, \mathcal{R}, \lambda)$. If $\mathrm{N}_j \notin \mathit{dom}(\Phi_i)$, then $\Phi_{i+1}(\mathrm{N}_j) = \mathbf{pc}(c_k, \mathcal{R}, \lambda)$ and $\Phi_{i+1}(\mathrm{N}_j) \cap \Phi(\mathrm{N}_j) \neq \emptyset$. If $\mathrm{N}_j \in \mathit{dom}(\Phi_i)$, then by **(I2)**, $c_j \in \Phi_i(\mathrm{N}_j)$. On the other hand, $c_j \in \Phi(\mathrm{N}_j)$ and, therefore, $c_j \in \mathbf{pc}(c_k, \mathcal{R}, \lambda)$. Since $\Phi_{i+1}(\mathrm{N}_j) = \Phi_i(\mathrm{N}_j) \cap \mathbf{pc}(c_k, \mathcal{R}, \lambda)$, we get $c_j \in \Phi_{i+1}(\mathrm{N}_j)$ and, hence, $\Phi_{i+1}(\mathrm{N}_j) \cap \Phi(\mathrm{N}_j) \neq \emptyset$.
To see that **(I2)** holds is easier. In fact, we have already shown above that $\Phi_{i+1}(\mathrm{N}_k) = \{c_k\}$. As for N_j, we have $\mathrm{N}_j \in \mathit{dom}(\Phi_{i+1})$ iff $\mathrm{N}_j \in \mathit{dom}(\Phi_i)$. In the latter case, we have seen in the proof of **(I1)** that $c_j \in \Phi_{i+1}(\mathrm{N}_j)$.

– The other cases will be dealt by the rules (FSN) and (NN2). The invariants for them trivially hold, since these rules do not change the problem.

By **(I1)**, the configurations in our derivation are solvable. Therefore, the failing rules do not apply. Hence, the derivation ends with $\emptyset; \Psi$ for some Ψ. By construction, $\mathit{dom}(\Psi) = \{\mathrm{N}_1, \ldots, \mathrm{N}_n\}$. By **(I2)**, $c_i \in \Psi(\mathrm{N}_i)$ for each $1 \leq i \leq n$. □

Example 3. The pre-unification derivation in Example 1 gives the neighborhood constraint $C = \{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f, \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} g, \mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3, \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\}$. For $\mathcal{R}_\lambda = \{(a,b), (b,c), (c,d), (a,b'), (b',c'), (c',d), (f,g), (p,q)\}$, the algorithm $\mathcal{CS}$ gives four solutions:

$$\Phi_1 = \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{b\}, \mathrm{N}_3 \mapsto \{f,g\}, \mathrm{N}_4 \mapsto \{c\}\}$$
$$\Phi_2 = \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{b'\}, \mathrm{N}_3 \mapsto \{f,g\}, \mathrm{N}_4 \mapsto \{c'\}\}.$$
$$\Phi_3 = \{\mathrm{N}_1 \mapsto \{g\}, \mathrm{N}_2 \mapsto \{b\}, \mathrm{N}_3 \mapsto \{f,g\}, \mathrm{N}_4 \mapsto \{c\}\}$$
$$\Phi_4 = \{\mathrm{N}_1 \mapsto \{g\}, \mathrm{N}_2 \mapsto \{b'\}, \mathrm{N}_3 \mapsto \{f,g\}, \mathrm{N}_4 \mapsto \{c'\}\}.$$

Referring to the mappings Φ and Φ' and the substitution μ in Example 1, it is easy to observe that $\Phi(\mu) \cup \Phi'(\mu) = \Phi_1(\mu) \cup \Phi_2(\mu) \cup \Phi_3(\mu) \cup \Phi_4(\mu)$.

4 Final remarks

We described an algorithm that solves unification problems over unrestricted proximity relations. It is terminating, sound, complete, and computes a compact representation of a minimal complete set of unifiers. A next step is to incorporate the computation of unification degree into the procedure and use it to characterize the "best" unifiers. Another future work involves a combination of unranked unification [6,7] and proximity relations to permit proximal function symbols with possibly different arities, similarly to the analogous extension of similarity-based unification described in [1].

References

1. H. Aït-Kaci and G. Pasi. Fuzzy unification and generalization of first-order terms over similar signatures. In F. Fioravanti and J. P. Gallagher, editors, *Logic-Based Program Synthesis and Transformation - 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017, Revised Selected Papers*, volume 10855 of *Lecture Notes in Computer Science*, pages 218–234. Springer, 2017.
2. D. Dubois and H. Prade. *Fuzzy sets and systems: theory and applications*, volume 144 of *Mathematics in science and engineering*. Acad. Press, 1980.
3. F. Formato, G. Gerla, and M. I. Sessa. Extension of logic programming by similarity. In M. C. Meo and M. V. Ferro, editors, *1999 Joint Conference on Declarative Programming, AGP'99, L'Aquila, Italy, September 6-9, 1999*, pages 397–410, 1999.
4. F. Formato, G. Gerla, and M. I. Sessa. Similarity-based unification. *Fundam. Inform.*, 41(4):393–414, 2000.
5. P. Julián-Iranzo and C. Rubio-Manzano. Proximity-based unification theory. *Fuzzy Sets and Systems*, 262:21–43, 2015.
6. T. Kutsia. Unification with sequence variables and flexible arity symbols and its extension with pattern-terms. In J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, and V. Sorge, editors, *Artificial Intelligence, Automated Reasoning, and Symbolic Computation, Joint International Conferences, AISC 2002 and Calculemus 2002, Marseille, France, July 1-5, 2002, Proceedings*, volume 2385 of *Lecture Notes in Computer Science*, pages 290–304. Springer, 2002.
7. T. Kutsia. Solving equations with sequence variables and sequence functions. *J. Symb. Comput.*, 42(3):352–388, 2007.
8. T. Kutsia and C. Pau. Proximity-based generalization. In M. Ayala Rincón and P. Balbiani, editors, *Proceedings of the 32nd International Workshop on Unification, UNIF 2018*, 2018.
9. T. Kutsia and C. Pau. Computing all maximal clique partitions in a graph. RISC Report 19-04, RISC, Johannes Kepler University Linz, 2019.
10. T. Kutsia and C. Pau. Solving proximity constraints. RISC Report 19-06, RISC, Johannes Kepler University Linz, 2019.
11. M. Rodríguez-Artalejo and C. A. Romero-Díaz. A declarative semantics for CLP with qualification and proximity. *TPLP*, 10(4-6):627–642, 2010.
12. M. I. Sessa. Approximate reasoning by similarity-based SLD resolution. *Theor. Comput. Sci.*, 275(1-2):389–426, 2002.

A Proofs

Lemma 2. *1. If $P;C;\mu \Longrightarrow \bot$ by* (Cla) *or* (Occ) *rules, then $P;C;\mu$ does not have a solution.*

2. Let $P_1;C_1;\mu_1 \Longrightarrow P_2;C_2;\mu_2$ be a step performed by a pre-unification rule (except (Cla) *and* (Occ)*). Then every solution of $P_2;C_2;\mu_2$ is a solution of $P_1;C_1;\mu_1$.*

Proof. 1. If (Cla) applies, we have an arity mismatch in an equation in P, which can not be repaired by substitutions: P is not unifiable. For the case of (Occ), the lemma follows from Lemma 1.

2. We shall prove the lemma for each non-failing rule. The nontrivial cases are (Dec), (VE), and (VO) rules.

   (Dec): $P_1 = \{\mathsf{F}(\mathsf{s}_1,\dots,\mathsf{s}_n) \simeq^?_{\mathcal{R},\lambda} \mathsf{G}(\mathsf{t}_1,\dots,\mathsf{t}_n)\} \uplus P$, $P_2 = \{\mathsf{s}_1 \simeq^?_{\mathcal{R},\lambda} \mathsf{t}_1,\dots, \mathsf{s}_n \simeq^?_{\mathcal{R},\lambda} \mathsf{t}_n\} \cup P$, $C_2 = C_1 \cup \{\mathsf{F} \approx^?_{\mathcal{R},\lambda} \mathsf{G}\}$, $\mu_1 = \mu_2$. Hence, the information about F and G to be $(\mathcal{R},\lambda)$-close to each other just moves from P_1 to C_2. The rest does not change. Therefore, $P_1;C_1;\mu_1$ and $P_2;C_2;\mu_2$ have the same set of solutions.

   (VE): Let $\xi = \{x \mapsto \mathsf{t}'\}$. Then $P_1 = \{x \simeq^?_{\mathcal{R},\lambda} \mathsf{t}\} \uplus P$, $P_2 = \{\mathsf{t}' \simeq^?_{\mathcal{R},\lambda} \mathsf{t}\} \cup P\xi$, $C_1 = C_2$, and $\mu_2 = \mu_1\xi$.

   Note that μ_2 contains $x \mapsto \mathsf{t}'$, because $x \notin dom(\mu_1)$ by the rules. Let (Φ,ν) be a solution of $P_2;C_2;\mu_2$. Then we have $\Phi(x\nu) = \Phi(\mathsf{t}'\nu)$. We also have $\Phi(\mathsf{t}'\nu) \simeq_{\mathcal{R},\lambda} \Phi(\mathsf{t}\nu)$, since ν solves $\mathsf{t}' \simeq^?_{\mathcal{R},\lambda} \mathsf{t}$. Hence, $\Phi(x\nu) \simeq_{\mathcal{R},\lambda} \Phi(\mathsf{t}\nu)$ and we get that (Φ,ν) solves the equation $x \simeq^?_{\mathcal{R},\lambda} \mathsf{t}$. For an equation $eq \in P$, we have $eq\nu = eq\xi\nu$, because $x\nu = t'\nu = x\xi\nu$, and for any $y \neq x$, it holds trivially that $y\nu = y\xi\nu$. Hence, (Φ,ν) solves P as well. For a $y \in dom(\mu_1)$, we have $y\nu = y\mu_2\nu = y\mu_1\xi\nu = y\mu_1\nu$. Therefore, (Φ,ν) is a solution of $P_1;C_1;\mu_1$.

   (VO): $P_1 = \{x \simeq^?_{\mathcal{R},\lambda} y, \overline{x_n \simeq^?_{\mathcal{R},\lambda} y_n}\}$, $P_2 = \{\overline{x_n \simeq^?_{\mathcal{R},\lambda} y_n}\}\{x \mapsto y\}$, $C_1 = C_2$, and $\mu_2 = \mu_1\{x \mapsto y\}$. We have $x \mapsto y \in \mu_2$, because the rules guarantee that $x \notin dom(\mu_1)$. Then for any solution (Φ,ν) of $P_2;C_2;\mu_2$ we have $x\nu = y\nu$, which implies the lemma.

Theorem 3 (Completeness of pre-unification). *Let an substitution ϑ be an $(\mathcal{R},\lambda)$-unifier of two terms s and t. Then any maximal derivation that starts at $\{s \simeq^?_{\mathcal{R},\lambda} t\};\emptyset;Id$ must end with an equational configuration $\emptyset;C;\mu$, such that for some $(\mathcal{R},\lambda)$-solution Φ of C, which maps names to singleton neighborhoods, and some substitution $\sigma \simeq_{\mathcal{R},\lambda} \vartheta$, we have $\Phi(\mu|_{\mathcal{V}(s)\cup\mathcal{V}(t)}) \preceq \sigma$.*

Proof. Since $\{s \simeq^?_{\mathcal{R},\lambda} t\}$ is solvable, the derivation can not end with $\bot$ by soundness of pre-unification. Since any equation in the first component in configurations can be transformed by a rule, and the algorithm terminates, the final configuration should have a form $\emptyset;C;\mu$.

Let V be the set of variables $\mathcal{V}(s) \cup \mathcal{V}(t)$. We show now how to construct the desired Φ step-by-step. Namely, we show the following proposition:

Proposition: Assume $P_1; C_1; \mu_1 \Longrightarrow P_2; C_2; \mu_2$ is a single step rule application in the above mentioned derivation. Let Φ_1 be a name-neighborhood mapping which maps names occurring in μ_1 to singleton neighborhoods such that equations in $\Phi_1(P_1)$ and in $\Phi_1(C_1)$ remain solvable. Assume that there exists a substitution $\sigma_1 \simeq_{\mathcal{R},\lambda} \vartheta$ such that $\Phi_1(\mu_1|_V) \preceq \sigma_1$. Then there exist a substitution $\sigma_2 \simeq_{\mathcal{R},\lambda} \vartheta$ and a name-neighborhood mapping Φ_2 which maps names occurring in μ_2 to singleton neighborhoods such that equations in $\Phi_2(P_2)$ and in $\Phi_2(C_2)$ remain solvable, and $\Phi_2(\mu_2|_V) \preceq \sigma_2$.

Proof of the proposition: We assume that $dom(\vartheta) \subseteq \mathcal{V}(s) \cup \mathcal{V}(t)$. We consider each of the rules separately.

(Tri): In this case we take $\Phi_2 = \Phi_1$ and $\sigma_2 = \sigma_1$.

(Dec): Note that if a name appears in P_1 or C_1, it appears in μ_1 as well. (The (VE) rule guarantees it, since we start from terms, not from X-terms.) Therefore, if the decomposed terms in P_1 had a name in their head, the name is already in μ_1 and, by assumption, in the domain of Φ_1. This step leaves $\mu_2 = \mu_1$ unchanged. Hence, we take $\Phi_2 = \Phi_1$ and $\sigma_2 = \sigma_1$. Since $\Phi_1(P_1)$ and $\Phi_1(C_1)$ were solvable, $\Phi_2(P_2)$ and $\Phi_2(C_2)$ are solvable as well.

(VE): $P_1 = \{x \simeq^?_{\mathcal{R},\lambda} \mathsf{t}\} \uplus P_1'$ where there is no occurrence cycle for x, $\mathsf{t} \notin \mathcal{V}$, $P_2 = \{\mathsf{t}' \simeq^?_{\mathcal{R},\lambda} \mathsf{t}\} \cup P_1'\{x \mapsto \mathsf{t}'\}$, and $\mu_2 = \mu_1\{x \mapsto \mathsf{t}'\}$, where $\mathsf{t}' = \rho(\mathsf{t})$. The new names in μ_2 are those introduced by ρ in t', and there is at least one of them, since $\mathsf{t} \notin \mathcal{V}$. Since ϑ is a solution of the initial configuration, and $x \simeq^?_{\mathcal{R},\lambda} \mathsf{t}$ is an equation between (copies of) the instances of the corresponding subterms of the original terms, $x\vartheta$ and $\mathsf{t}\vartheta$ have a similar structure: if at position p in $\mathsf{t}\vartheta$ there is a name or function symbol, then at position p in $x\vartheta$ we also have a function symbol. Besides, if that position p in $\mathsf{t}\vartheta$ is the position p in t, then we have a new name at position p in t'. Now, let Φ_2 be

$$\Phi_2 := \Phi_1 \cup \{\mathrm{N} \to \{f\} \mid \mathrm{N} \text{ is the name at position } p \text{ in } \mathsf{t}' \text{ and } f \text{ is the function symbol at position } p \text{ in } x\vartheta\}.$$

Then, if N is a renaming of a function symbol g at position p in t (and, hence, in $\mathsf{t}\vartheta$), and f is a function symbol at position p in $x\vartheta$, then f and g must be $(\mathcal{R},\lambda)$-close to each other, because ϑ should unify those positions. Then solvability of $\Phi_1(P_1)$ implies solvability of $\Phi_2(P_2)$.
If $x \in V$, then by pre-unification rules, x does not appear in the range of substitutions in configuration. In particular, it does not appear in the range of μ_1 and, therefore, by substitution composition we have $\mu_2 = \mu_1 \cup \{x \mapsto \mathsf{t}'\}$. Since $\mathsf{t} \notin \mathcal{V}$, x appears in $dom(\vartheta)$. By construction, if we have a name N in position p in $x\mu_2$, in $x\vartheta$ in the same position we have a function symbol f with $\Phi_2(\mathrm{N}) = \{f\}$. In other positions in $x\mu_2$ we have fresh variables, each occurring once. Then there exists a substitution φ such that $x\Phi_2(\mu_2)\varphi = x\vartheta$. Besides, from the assumptions we know that there exists ψ such that for any other variable y in V we have $y\Phi_2(\mu_2)\psi = y\Phi_1(\mu_1)\psi = y\sigma_1$. We

define $\sigma_2 = \{x \mapsto x\vartheta\} \cup \{y \mapsto y\sigma_1 \mid y \in V \setminus \{x\}\}$. Then $\sigma_2 \simeq_{\mathcal{R},\lambda} \vartheta$ and $\Phi_2(\mu_2|_V)(\psi \cup \varphi) = \sigma_2$. Hence, $\Phi_2(\mu_2|_V) \preceq \sigma_2$.
If $x \notin V$, then $\mu_2|_V$ is obtained from $\mu_1|_V$ by replacing x with t'. Let $z \mapsto \mathsf{r}[x] \in \mu_1|_V$ be a substitution pair in where x appears. Then $\mu_2|_V$ will contain $z \mapsto \mathsf{r}[\mathsf{t}']$. By the definition of Φ_2 and by the fact that t' is a fresh copy of t, there exists φ such that $z\Phi_2(\mu_2)\varphi = z\vartheta$. For any y in V such that $x \notin \mathcal{V}(y\mu_2)$, we have $y\Phi_2(\mu_2)\psi = y\sigma_1$ for some ψ. With a similar reasoning as above, we conclude the existence of σ_2 such that $\Phi_2(\mu_2|_V) \preceq \sigma_2$. When $z \mapsto \mathsf{r}[x] \notin \mu_1|_V$, we will not have z in $dom(\mu_2|_V)$ either. Besides, z appears neither in the range of μ_1 nor in the range of μ_2 (since by construction those substitutions are idempotent). Hence, we can reuse σ_1 in the role of σ_2.

(Ori): Straightforward.

(VO): We have $P_1 = \{x \simeq^?_{\mathcal{R},\lambda} y, \overline{x_n \simeq^?_{\mathcal{R},\lambda} y_n}\}$, $P_2 = \{\overline{x_n \simeq^?_{\mathcal{R},\lambda} y_n}\}\{x \mapsto y\}$, $C_1 = C_2$, and $\mu_2 = \mu_1\{x \mapsto y\}$. Then $\Phi_2 = \Phi_1$. Then equations in $\Phi(C_2)$ are solvable. Since ϑ is a solution of the initial configuration, and $x \simeq^?_{\mathcal{R},\lambda} y$ is an equation between (copies of) the instances of the corresponding variable subterms of the original terms, $x\vartheta$ and $y\vartheta$ are $(\mathcal{R},\lambda)$-close to each other. Then equations in $\Phi_2(P_2)$ are also solvable.
Note that $x, y \notin dom(\mu)$, because $x, y \in \mathcal{V}(P_1)$. Therefore, $x\Phi_1(\mu_1) = x$ and $y\Phi_1(\mu_1) = y$. On the other hand, $x\Phi_2(\mu_2) = y\Phi_2(\mu_2) = y$.
Assume $x, y \in V$. From the assumptions we know that there exists ψ such that for any other variable z in V we have $z\Phi_2(\mu_2)\psi = z\Phi_1(\mu_1)\psi = z\sigma_1$. We define $\sigma_2 = \{x \mapsto y\vartheta, y \mapsto y\vartheta\} \cup \{z \mapsto z\sigma_1 \mid z \in V \setminus \{x, y\}\}$. Then $\sigma_2 \simeq_{\mathcal{R},\lambda} \vartheta$ and $\Phi_2(\mu_2|_V)(\psi \cup \{y \mapsto y\vartheta\}) = \sigma_2$.
If $x \notin V$, then we take $\sigma_2 = \{y \mapsto y\vartheta\} \cup \{z \mapsto z\sigma_1 \mid z \in V \setminus \{y\}\}$. If $y \notin V$, then $\sigma_2 = \{x \mapsto y\vartheta\} \cup \{z \mapsto z\sigma_1 \mid z \in V \setminus \{x\}\}$. If $x, y \notin V$, then $\sigma_2 = \sigma_1$. This finishes the proof of the proposition.

In the constructed derivation, for the initial configuration $P_0; C_0; \mu_0 = \{s \simeq^?_{\mathcal{R},\lambda} t\}; \emptyset; Id$ we take $\Phi_0 = \emptyset$. Then $\Phi_0(P_0) = P_0 = \{s \simeq^?_{\mathcal{R},\lambda} t\}$ and $\Phi_0(C_0) = \emptyset$ are solvable, and $\Phi_0(\mu_0|_V) = Id \preceq \vartheta$. By applying the proposition iteratively, we get that for the final configuration $\emptyset; C; \mu$, there exists Φ which maps names to singleton neighborhoods such that $\Phi(C)$ is solvable. By the way how Φ is constructed, its domain consists of all the names that appear in μ, but they are exactly those that appear in C. Hence, $\Phi(C)$ does not contain names and its solvability means that it is already solved (trivially solvable). It implies that Φ is a solution of C. Besides, again by an iterative application of the proposition, we show the existence of a $\sigma \simeq_{\mathcal{R},\lambda} \vartheta$ such that $\Phi(\mu|_V) \preceq \sigma$. □

B Examples

The selected equations are underlined.

Example 4. Let $s = p(x, y, x)$ and $t = q(f(a), g(d), y)$. The following is a pre-unification derivation, which shows how the neighborhood constraint and the substitution shown in Example 1 are computed:

$$\{\underline{p(x,y,x) \simeq^?_{\mathcal{R},\lambda} q(f(a),g(d),y)}\};\ \emptyset;\ Id \Longrightarrow_{\mathsf{Dec}}$$
$$\{\underline{x \simeq^?_{\mathcal{R},\lambda} f(a)},\ y \simeq^?_{\mathcal{R},\lambda} g(d),\ x \simeq^?_{\mathcal{R},\lambda} y\};\ \{p \approx^?_{\mathcal{R},\lambda} q\};\ Id \Longrightarrow_{\mathsf{VE}}$$
$$\{\underline{\mathrm{N}_1(\mathrm{N}_2) \simeq^?_{\mathcal{R},\lambda} f(a)},\ y \simeq^?_{\mathcal{R},\lambda} g(d),\ \mathrm{N}_1(\mathrm{N}_2) \simeq^?_{\mathcal{R},\lambda} y\};\ \{p \approx^?_{\mathcal{R},\lambda} q\};\ \{x \mapsto \mathrm{N}_1(\mathrm{N}_2)\} \Longrightarrow_{\mathsf{Dec}^2}$$
$$\{\underline{y \simeq^?_{\mathcal{R},\lambda} g(d)},\ \mathrm{N}_1(\mathrm{N}_2) \simeq^?_{\mathcal{R},\lambda} y\};\ \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a\};\ \{x \mapsto \mathrm{N}_1(\mathrm{N}_2)\} \Longrightarrow_{\mathsf{VE}}$$
$$\{\underline{\mathrm{N}_3(\mathrm{N}_4) \simeq^?_{\mathcal{R},\lambda} g(d)},\ \mathrm{N}_1(\mathrm{N}_2) \simeq^?_{\mathcal{R},\lambda} \mathrm{N}_3(\mathrm{N}_4)\};\ \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a\};\ \{x \mapsto \mathrm{N}_1(\mathrm{N}_2), y \mapsto \mathrm{N}_3(\mathrm{N}_4)\} \Longrightarrow_{\mathsf{Dec}^2}$$
$$\{\underline{\mathrm{N}_1(\mathrm{N}_2) \simeq^?_{\mathcal{R},\lambda} \mathrm{N}_3(\mathrm{N}_4)}\};\ \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} g,\ \mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d\};\ \{x \mapsto \mathrm{N}_1(\mathrm{N}_2), y \mapsto \mathrm{N}_3(\mathrm{N}_4)\} \Longrightarrow_{\mathsf{Dec}^2}$$
$$\emptyset;\ \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} g,\ \mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\};\ \{x \mapsto \mathrm{N}_1(\mathrm{N}_2), y \mapsto \mathrm{N}_3(\mathrm{N}_4)\}.$$

Note that we could have chosen the second equation in the set $\{y \simeq^?_{\mathcal{R},\lambda} g(d),\ \mathrm{N}_1(\mathrm{N}_2) \simeq^?_{\mathcal{R},\lambda} y\}$ to eliminate y, but the obtained result would differ from the above computed one by the choice of names only.

Together with the output from Example 3, we report the solutions for $\mathcal{R}_\lambda = \{(a,b),(b,c),(c,d),(a,b'),(b',c'),(c',d),(f,g),(p,q)\}$,

$$\Phi_1 = \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{b\},\ \mathrm{N}_3 \mapsto \{f,g\}, \mathrm{N}_4 \mapsto \{c\}\},$$
$$\Phi_2 = \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{b'\}, \mathrm{N}_3 \mapsto \{f,g\}, \mathrm{N}_4 \mapsto \{c'\}\},$$
$$\Phi_3 = \{\mathrm{N}_1 \mapsto \{g\}, \mathrm{N}_2 \mapsto \{b\},\ \mathrm{N}_3 \mapsto \{f,g\}, \mathrm{N}_4 \mapsto \{c\}\},$$
$$\Phi_4 = \{\mathrm{N}_1 \mapsto \{g\}, \mathrm{N}_2 \mapsto \{b'\}, \mathrm{N}_3 \mapsto \{f,g\}, \mathrm{N}_4 \mapsto \{c'\}\},$$
$$\mu = \{x \mapsto \mathrm{N}_1(\mathrm{N}_2), y \mapsto \mathrm{N}_3(\mathrm{N}_4)\}.$$

Example 5. Now we illustrate the steps made for computations in Example 2. Let $s = p(x, x)$ and $t = q(f(y, y), f(a, c))$. Then the pre-unification derivation looks as follows:

$$\{\underline{p(x,x) \simeq^?_{\mathcal{R},\lambda} q(f(y,y),f(a,c))}\};\ \emptyset;\ Id \Longrightarrow_{\mathsf{Dec}}$$
$$\{\underline{x \simeq^?_{\mathcal{R},\lambda} f(y,y)},\ x \simeq^?_{\mathcal{R},\lambda} f(a,c)\};\ \{p \approx^?_{\mathcal{R},\lambda} q\};\ Id \Longrightarrow_{\mathsf{VE}}$$

$\{\underline{\mathrm{N}_1(y_1,y_2) \simeq^?_{\mathcal{R},\lambda} f(y,y)}, \mathrm{N}_1(y_1,y_2) \simeq^?_{\mathcal{R},\lambda} f(a,c)\}; \{p \approx^?_{\mathcal{R},\lambda} q\};$
$\{x \mapsto \mathrm{N}_1(y_1,y_2)\} \Longrightarrow_{\mathsf{Dec}}$
$\{y_1 \simeq^?_{\mathcal{R},\lambda} y, y_2 \simeq^?_{\mathcal{R},\lambda} y, \underline{\mathrm{N}_1(y_1,y_2) \simeq^?_{\mathcal{R},\lambda} f(a,c)}\}; \{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f\};$
$\{x \mapsto \mathrm{N}_1(y_1,y_2)\} \Longrightarrow_{\mathsf{Dec}}$
$\{y_1 \simeq^?_{\mathcal{R},\lambda} y, y_2 \simeq^?_{\mathcal{R},\lambda} y, \underline{y_1 \simeq^?_{\mathcal{R},\lambda} a}, y_2 \simeq^?_{\mathcal{R},\lambda} c\}; \{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f\};$
$\{x \mapsto \mathrm{N}_1(y_1,y_2)\} \Longrightarrow_{\mathsf{VE,Dec}}$
$\{\mathrm{N}_2 \simeq^?_{\mathcal{R},\lambda} y, y_2 \simeq^?_{\mathcal{R},\lambda} y, \underline{y_2 \simeq^?_{\mathcal{R},\lambda} c}\}; \{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f, \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a\};$
$\{x \mapsto \mathrm{N}_1(\mathrm{N}_2,y_2), y_1 \mapsto \mathrm{N}_2\} \Longrightarrow_{\mathsf{VE,Dec}}$
$\{\underline{\mathrm{N}_2 \simeq^?_{\mathcal{R},\lambda} y}, \mathrm{N}_3 \simeq^?_{\mathcal{R},\lambda} y\}; \{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f, \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c\};$
$\{x \mapsto \mathrm{N}_1(\mathrm{N}_2,\mathrm{N}_3), y_1 \mapsto \mathrm{N}_2, y_2 \mapsto \mathrm{N}_3\} \Longrightarrow_{\mathsf{Ori,VE}}$
$\{\underline{\mathrm{N}_3 \simeq^?_{\mathcal{R},\lambda} \mathrm{M}}\}; \{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f, \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c, \mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\};$
$\{x \mapsto \mathrm{N}_1(\mathrm{N}_2,\mathrm{N}_3), y_1 \mapsto \mathrm{N}_2, y_2 \mapsto \mathrm{N}_3, y \mapsto \mathrm{M}\} \Longrightarrow_{\mathsf{Dec}}$
$\emptyset; \{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f, \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c, \mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{M}\};$
$\{x \mapsto \mathrm{N}_1(\mathrm{N}_2,\mathrm{N}_3), y_1 \mapsto \mathrm{N}_2, y_2 \mapsto \mathrm{N}_3, y \mapsto \mathrm{M}\}.$

If $\mathcal{R}_\lambda = \{(a,a_1),(a_1,b),(b,c_1),(c_1,c),(p,q)\}$, then the obtained constraint is satisfied by the assignment $[\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a_1\}, \mathrm{M} \mapsto \{b\}, \mathrm{N}_3 \mapsto \{c_1\}]$, computed by $\mathcal{CS}$ as follows (where NN1 causes branching, we display only the success branch):

$\{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f, \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c, \mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{M}\};$
$\emptyset \Longrightarrow_{\mathsf{FFS}}$
$\{\mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f, \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c, \mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{M}\}; \emptyset \Longrightarrow_{\mathsf{NFS}}$
$\{\mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c, \mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{M}\}; \{\mathrm{N}_1 \mapsto \{f\}\} \Longrightarrow_{\mathsf{NFS}}$
$\{\mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c, \mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{M}\}; \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a,a_1\}\} \Longrightarrow_{\mathsf{NFS}}$
$\{\mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{M}\}; \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a,a_1\}, \mathrm{N}_3 \mapsto \{c,c_1\}\} \Longrightarrow_{\mathsf{NN2}}$
$\{\mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{M}, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{M}\}; \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a,a_1\}, \mathrm{N}_3 \mapsto \{c,c_1\}\} \Longrightarrow_{\mathsf{NN1}}$
$\{\mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{M}\}; \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a_1\}, \mathrm{N}_3 \mapsto \{c,c_1\}, \mathrm{M} \mapsto \{b\}\} \Longrightarrow_{\mathsf{NN1}}$
$\emptyset; \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a_1\}, \mathrm{N}_3 \mapsto \{c_1\}, \mathrm{M} \mapsto \{b\}\}.$

Hence, the computed solution is

$\Phi = \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a_1\}, \mathrm{N}_3 \mapsto \{c_1\}, \mathrm{M} \mapsto \{b\}\},$
$\mu = \{x \mapsto \mathrm{N}_1(\mathrm{N}_2,\mathrm{N}_3), y_1 \mapsto \mathrm{N}_2, y_2 \mapsto \mathrm{N}_3, y \mapsto \mathrm{M}\}.$
$\mu|_{\mathcal{V}(s)\cup\mathcal{V}(t)} = \{x \mapsto \mathrm{N}_1(\mathrm{N}_2,\mathrm{N}_3), y \mapsto \mathrm{M}\}.$

If we used the same new variable, say y', for both occurrences of y in $f(y,y)$ (instead of using y_1 and y_2 as above), we would get the following pre-unification

derivation:

$\{\underline{p(x,x) \simeq^?_{\mathcal{R},\lambda} q(f(y,y),f(a,c))}\};\ \emptyset;\ Id \Longrightarrow_{\mathsf{Dec}}$
$\{\underline{x \simeq^?_{\mathcal{R},\lambda} f(y,y)},\ x \simeq^?_{\mathcal{R},\lambda} f(a,c)\};\ \{p \approx^?_{\mathcal{R},\lambda} q\};\ Id \Longrightarrow_{\mathsf{VE}}$
$\{\underline{\mathrm{N}_1(y',y') \simeq^?_{\mathcal{R},\lambda} f(y,y)},\ \mathrm{N}_1(y',y') \simeq^?_{\mathcal{R},\lambda} f(a,c)\};\ \{p \approx^?_{\mathcal{R},\lambda} q\};$
$\quad \{x \mapsto \mathrm{N}_1(y',y')\} \Longrightarrow_{\mathsf{Dec}}$
$\{y' \simeq^?_{\mathcal{R},\lambda} y,\ \underline{\mathrm{N}(y',y') \simeq^?_{\mathcal{R},\lambda} f(a,c)}\};\ \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f\};$
$\quad \{x \mapsto \mathrm{N}_1(y',y')\} \Longrightarrow_{\mathsf{Dec}}$
$\{y' \simeq^?_{\mathcal{R},\lambda} y,\ \underline{y' \simeq^?_{\mathcal{R},\lambda} a},\ y' \simeq^?_{\mathcal{R},\lambda} c\};\ \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f\};$
$\quad \{x \mapsto \mathrm{N}_1(y',y')\} \Longrightarrow_{\mathsf{VE,Dec}}$
$\{\underline{\mathrm{N}_2 \simeq^?_{\mathcal{R},\lambda} y},\ \mathrm{N}_2 \simeq^?_{\mathcal{R},\lambda} c\};\ \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a\};$
$\quad \{x \mapsto \mathrm{N}_1(\mathrm{N}_2,\mathrm{N}_2),\ y' \mapsto \mathrm{N}_2\} \Longrightarrow_{\mathsf{VE,Dec}}$
$\{\underline{\mathrm{N}_2 \simeq^?_{\mathcal{R},\lambda} \mathrm{N}_3}\};\ \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c\};$
$\quad \{x \mapsto \mathrm{N}_1(\mathrm{N}_2,\mathrm{N}_3),\ y' \mapsto \mathrm{N}_2,\ y \mapsto \mathrm{N}_2\} \Longrightarrow_{\mathsf{Dec}}$
$\emptyset;\ \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\};$
$\quad \{x \mapsto \mathrm{N}_1(\mathrm{N}_2,\mathrm{N}_2),\ y' \mapsto \mathrm{N}_2,\ y \mapsto \mathrm{N}_2\}.$

If $\mathcal{R}_\lambda = \{(a,a_1),(a_1,b),(b,c_1),(c_1,c),(p,q)\}$, the obtained neighborhood constraint does not have a solution:

$\{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\};\ \emptyset \Longrightarrow_{\mathsf{FFS}}$
$\{\mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\};\ \emptyset \Longrightarrow_{\mathsf{NFS}}$
$\{\mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\};\ \{\mathrm{N}_1 \mapsto \{f\}\} \Longrightarrow_{\mathsf{NFS}}$
$\{\mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\};\ \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a,a_1\}\} \Longrightarrow_{\mathsf{NFS}}$
$\{\mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\};\ \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a,a_1\}, \mathrm{N}_3 \mapsto \{c,c_1\}\}$

From here, NN1 generates two branches. First:

$\{\mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\};\ \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a,a_1\}, \mathrm{N}_3 \mapsto \{c,c_1\}\} \Longrightarrow_{\mathsf{NN1}}$
$\emptyset;\ \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a\}, \mathrm{N}_3 \mapsto \emptyset\} \Longrightarrow_{\mathsf{Fail2}} \bot.$

Second:

$\{\mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\};\ \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a,a_1\}, \mathrm{N}_3 \mapsto \{c,c_1\}\} \Longrightarrow_{\mathsf{NN1}}$
$\emptyset;\ \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a_1\}, \mathrm{N}_3 \mapsto \emptyset\} \Longrightarrow_{\mathsf{Fail2}} \bot.$

Example 6. In Example 3, we saw four solutions obtained by the constraint solving algorithm for the the neighborhood constraint $C = \{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} g,\ \mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\}$ and $\mathcal{R}_\lambda = \{(a,b),(b,c),(c,d),(a,b'),(b',c'),(c',d),(f,g),(p,q)\}$.

Here we illustrate the steps how the constraint C can be solved by $\mathcal{CS}$ (the selected equation is always the leftmost):

$$\{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} g,\ \mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\}; \emptyset \Longrightarrow_{\mathsf{FFS}}$$

$$\{\mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} g,\ \mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\}; \emptyset \Longrightarrow_{\mathsf{NFS}}$$

$$\{\mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} g,\ \mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\}; \{\mathrm{N}_1 \mapsto \{f, g\}\} \Longrightarrow_{\mathsf{NFS}}$$

$$\{\mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} g,\ \mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\}; \{\mathrm{N}_1 \mapsto \{f, g\},\ \mathrm{N}_2 \mapsto \{a, b, b'\}\} \Longrightarrow_{\mathsf{NFS}}$$

$$\{\mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\}; \{\mathrm{N}_1 \mapsto \{f, g\},\ \mathrm{N}_2 \mapsto \{a, b, b'\},\ \mathrm{N}_3 \mapsto \{f, g\}\} \Longrightarrow_{\mathsf{NFS}}$$

$$\{\mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\}; \{\mathrm{N}_1 \mapsto \{f, g\},\ \mathrm{N}_2 \mapsto \{a, b, b'\},\ \mathrm{N}_3 \mapsto \{f, g\},\ \mathrm{N}_4 \mapsto \{d, c, c'\}\}.$$

Here the algorithm branches, since NN1 applies. Branch 1 continues with

$$\{\mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\}; \{\mathrm{N}_1 \mapsto \{f, g\}, \mathrm{N}_2 \mapsto \{a, b, b'\}, \mathrm{N}_3 \mapsto \{f, g\}, \mathrm{N}_4 \mapsto \{d, c, c'\}\} \Longrightarrow_{\mathsf{NN1}}$$

$$\{\mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\}; \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a, b, b'\}, \mathrm{N}_3 \mapsto \{f, g\}, \mathrm{N}_4 \mapsto \{d, c, c'\}\}.$$

Branching again by NN1 produces three subbranches. Branch 1.1 fails: $\emptyset; \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a\}, \mathrm{N}_3 \mapsto \{f, g\}, \mathrm{N}_4 \mapsto \emptyset\} \Longrightarrow_{\mathsf{Fail2}} \bot$. Branch 1.2 and branch 1.3 give two solutions, respectively:

$$\Phi_1 = \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{b\},\ \mathrm{N}_3 \mapsto \{f, g\}, \mathrm{N}_4 \mapsto \{c\}\}$$
$$\Phi_2 = \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{b'\}, \mathrm{N}_3 \mapsto \{f, g\}, \mathrm{N}_4 \mapsto \{c'\}\}.$$

Branch 2 also expands into three subbranches, the first of which fails. The other two return two more solutions:

$$\Phi_3 = \{\mathrm{N}_1 \mapsto \{g\}, \mathrm{N}_2 \mapsto \{b\},\ \mathrm{N}_3 \mapsto \{f, g\}, \mathrm{N}_4 \mapsto \{c\}\}$$
$$\Phi_4 = \{\mathrm{N}_1 \mapsto \{g\}, \mathrm{N}_2 \mapsto \{b'\}, \mathrm{N}_3 \mapsto \{f, g\}, \mathrm{N}_4 \mapsto \{c'\}\}.$$

Referring to the name-neighborhood mappings Φ and Φ' and the substitution μ in Example 1, it is easy to observe that $\Phi(\mu) \cup \Phi'(\mu) = \Phi_1(\mu) \cup \Phi_2(\mu) \cup \Phi_3(\mu) \cup \Phi_4(\mu)$.

A Certified Functional Nominal C-Unification Algorithm*

Mauricio Ayala-Rincón[1]**, Maribel Fernández[2], Gabriel Ferreira Silva[1]***, and Daniele Nantes-Sobrinho[1]

[1] Departments of Computer Science and Mathematics, Universidade de Brasília
[2] Department of Computer Science, King's College London

Abstract. The nominal approach allows us to extend first-order syntax and represent smoothly systems with variable bindings using, instead of variables, nominal atoms and dealing with renaming through permutations of atoms. Nominal unification is, therefore, the extension of first-order unification modulo α-equivalence by taking into account this nominal setting. In this work, we present a specification of a nominal C-unification algorithm (nominal unification with commutative operators) in PVS and discuss aspects about the proofs of soundness and completeness. Additionally, the algorithm has been implemented in Python. In relation to the only known specification of nominal C-unification, there are two novel features in this work: first, the formalization of a functional algorithm that can be directly executed (not just a set of non-deterministic inference rules); second, simpler proofs of termination, soundness and completeness, due to the reduction in the number of parameters of the lexicographic measure, from four parameters to only two.

Keywords: Nominal Terms, Nominal C-Unification, Verification of Functional Specifications

1 Introduction

The nominal approach allows us to extend first-order syntax and represent smoothly systems with bindings, which are frequent in computer science and mathematics. However, in order to represent bindings correctly, α-equivalence must be taken into account. For instance, despite their syntactical difference, the formulas $\exists x : x < 0$ and $\exists z : z < 0$ should be considered equivalent. The nominal theory allows us to deal with these bindings in a natural way, using atoms, atom permutations and freshness constraints, instead of using indices as in explicit substitutions *à la de Bruijn* (e.g. [19], [14]).

* Work supported by FAPDF grant 193001369/2016.
** Corresponding author, partially funded by CNPq research grant number 307672/2017-4.
*** Corresponding author, was funded by CNPq scholarship number 139271/2017-1. Work presentation funded by FAPDF.

On the other hand, unification is an important problem in first-order theories, with applications to logic programming systems, type inference algorithms, theorem provers and so on (e.g. [12]). Since unification is essential for equational reasoning, the development of unification techniques for nominal logic has been an attractive area of research since the invention of the nominal approach.

The problem of nominal unification has been solved by Urban et al. ([22]), with further research being developed on algorithm improvements to solve this problem (e.g. [10], [17]). The area now also focuses on elaborating extensions of nominal unification (e.g [20], [16], [9]), among them nominal unification modulo equational theories (e.g. [2], [1], [5], [3]). Here, we consider nominal unification modulo commutative function symbols (nominal C-unification, for short).

Related Work. Nominal unification was first solved by Urban et al. in [22], by proposing a set of transformation rules and showing, using Isabelle/HOL, their correctness and completeness [21]. An alternative specification of nominal unification, as a function that maps (solvable) problems to solutions, was formalised in PVS and proved sound and complete [6]. This work brought two new perspectives to the problem. The first is the specification of a functional algorithm for nominal unification, not a set of inference rules, which made the specification closer to the implementation. The second is the separate treatment of freshness constraints and equational constraints. Both ideas are used in this paper.

Nominal C-unification extends nominal unification to handle commutative function symbols. Using the Coq proof assistant, a set of non-deterministic transformation rules to solve nominal C-unification problems, in the style of Urban et al. [21], was shown correct and complete [2].

Contribution. In this paper, we present the first (to our knowledge) functional nominal C-unification algorithm and formalize its correctness and completeness using the proof assistant PVS. We emphasize the most interesting aspects of its formalization.

Although there is one other specification ([2]) for nominal C-Unification, the approaches taken are different. In [2], a set of rules that gradually transforms the nominal C-unification problem into simpler ones is presented. Here, by contrast, we develop a recursive algorithm, specified and formalized in PVS and implemented in Python, not a set of inference rules. The advantage of this approach is that the algorithm can be executed, while the set of inference rules, specified through inductive definitions cannot, because it is non-deterministic.

As mentioned previously, [6] gave us a nominal unification algorithm and a new insight about the problem: the possibility to handle freshness constraints and equational constraints separately. We adapt a significant portion of the formalization of [6], adding and formalizing the necessary lemmas to obtain a sound and complete algorithm for nominal C-unification and keep the separate treatment of freshness constraints and equational constraints. This insight, along with a trick of separating fixed point equations (see Definition 9) from the unification problem, allowed us to reduce the lexicographic measure found in [2], from 4 parameters to only 2 parameters which made the proofs of termination, soundness and completeness simpler.

Finally, the formalization of soundness and completeness was done in PVS and is available at `http://www.github.com/gabriel951/c-unification`. PVS was chosen partly in order to reuse the definitions and lemmas previously used in [6] and partly because its specification language provides great support for the definition (and formalization) of functional recursive algorithms.

Possible Applications. As remarked before, nominal unification is used in logic programming. Therefore, the nominal C-unification algorithm could be used on a logic programming language that uses the nominal setting, such as α-Prolog [12]. Another application is related to matching (see [8], [7]). Matching two terms t and s can be seen as unification where one of the terms (suppose t, without loss of generality) is not affected by substitutions [2]. This can be accomplished by adding as an additional parameter to the algorithm a set of variables that are forbidden to be instantiated. Then, matching boils down to unifying, using as this additional parameter the set of variables in t [2]. The C-matching algorithm proposed could then, for instance, be used to extend the nominal rewriting relation introduced in [14] modulo commutativity. Also, nominal C-unification and matching are relevant to implement nominal narrowing introduced in [4] allowing commutative symbols.

Organization. The paper is organized as follows. First, in Section 2, we provide the necessary background. The nominal setting is explained and the problem of nominal C-unification is defined. In Section 3, we present and explain the pseudocode for the algorithm specified in PVS and implemented in Python. In Section 4, we discuss the main aspects of the formalization: the principal lemmas, the hardest cases, how introducing commutativity made the problem more complex and so on. Finally, in Section 6, we conclude the paper and offer possible paths of future work. An extended version of this paper is available at `http://ayala.mat.unb.br/publications.html`.

2 Background

In this section, we provide the necessary background in nominal theory.

2.1 Nominal Terms, Permutations and Substitutions

In nominal theory, we consider disjoint countable sets of atoms $\mathcal{A} = \{a, b, c, ...\}$ and of variables $\mathcal{X} = \{X, Y, Z, ...\}$. A permutation π is a bijection of the form $\pi : \mathcal{A} \to \mathcal{A}$ such that the domain of π (i.e., the set of atoms that are modified by π) is finite. Permutations are usually represented as a list of swappings, where the swapping $(a\ b)$ exchanges the atoms a and b and fixes the other atoms. Therefore, a permutation is represented as $\pi = (a_1\ b_1) :: ... :: (a_n\ b_n) :: nil$. π^{-1}, the inverse of this permutation, is computed by simply reversing the list.

Definition 1 (Nominal Terms). *Let Σ be a signature with function symbols and commutative function symbols. The set $\mathcal{T}(\Sigma, \mathcal{A}, \mathcal{X})$ of nominal terms is generated according to the grammar:*

$$s, t \quad ::= \quad \langle\rangle \ \mid\ a \ \mid\ \pi \cdot X \ \mid\ [a]t \ \mid\ \langle s, t\rangle \ \mid\ f\ t \ \mid\ f^C\langle s, t\rangle \tag{1}$$

where $\langle\rangle$ is the unit, a is an atom term, $\pi \cdot X$ is a suspended variable (the permutation π is suspended on the variable X), $[a]t$ is an abstraction (a term with the atom a abstracted), $\langle s, t\rangle$ is a pair, $f\ t$ is a function application and $f^C\langle s, t\rangle$ is a commutative function application over a pair.

Remark 1. Pairs can be used to encode tuples with an arbitrary number of arguments. For instance, the tuple (t_1, t_2, t_3) could be constructed as $\langle t_1, \langle t_2, t_3\rangle\rangle$.

Remark 2. Following the proposal of [2], we impose that commutative functions receive a pair as their argument. No generality is lost with this restriction and the analysis is simplified.

Definition 2 (Permutation Action). *The permutation action on atoms is defined recursively: $nil \cdot c = c$, $(a\ b) :: \pi \cdot a = \pi \cdot b$, $(a\ b) :: \pi \cdot b = \pi \cdot a$ and $(a\ b) :: \pi \cdot c = \pi \cdot c$. The action of permutations on terms is defined recursively:*

$$\pi \cdot \langle\rangle = \langle\rangle \qquad \pi \cdot (\pi' \cdot X) = (\pi :: \pi') \cdot X$$
$$\pi \cdot [a]t = [\pi \cdot a]\pi \cdot t \qquad \pi \cdot \langle s, t\rangle = \langle \pi \cdot s, \pi \cdot t\rangle$$
$$\pi \cdot f\ t = f\ \pi \cdot t \qquad \pi \cdot f^C\langle s, t\rangle = f^C\langle \pi \cdot s, \pi \cdot t\rangle$$

Remark 3. We follow Gabbay's permutative convention, which says that atoms differ in their names. Therefore, if we consider atoms a and b, it is redundant to say $a \neq b$.

Example 1. To illustrate the action of a permutation on a term, consider $\pi = (a\ b) :: (c\ d) :: nil$ and $t = f(a, c)$. Then, the result of the permutation action is $\pi \cdot t = f(b, d)$.

Definition 3 (Nuclear Substitution). *A nuclear substitution is a pair $[X \to t]$, where X is a variable and t is a term. Nuclear substitutions act over terms:*

$$\langle\rangle[X \to t] = \langle\rangle \qquad a[X \to t] = a$$
$$([a]s)[X \to t] = [a](s[X \to t]) \qquad \pi \cdot Y[X \to t] = \begin{cases} \pi \cdot Y & \text{if } X \neq Y \\ \pi \cdot t & \text{otherwise} \end{cases}$$
$$\langle s_1, s_2\rangle[X \to t] = \langle s_1[X \to t], s_2[X \to t]\rangle \qquad (f\ s)[X \to t] = f\ (s[X \to t])$$
$$(f^C\langle s_1, s_2\rangle)[X \to t] = f^C\langle s_1[X \to t], s_2[X \to t]\rangle$$

Definition 4 (Substitution Action on Terms). *A substitution σ is a list of nuclear substitutions, which are applied consecutively to a term:*

$$s\ id = s, \text{ where } id \text{ is the empty list} \qquad s(\sigma :: [X \to t]) = (s[X \to t])\sigma \tag{2}$$

Remark 4. The notion of substitution defined here differs from the more traditional view of a substitution as a simultaneous application of nuclear substitutions, although both are correct [6]. The notion here presented is closer to the concept of triangular substitutions [15].

Example 2. Let $\sigma = [X \to a] :: [Y \to f(X, b)]$ and $t = [a]Y$. Then, $t\sigma = [a]f(a, b)$.

2.2 Freshness and α-equality

Two valuable notions in nominal theory are freshness and α-equality, which are represented, respectively, by the predicates $\#$ and $\approx_\alpha$.

- $a\#t$ means, intuitively, that if a occurs in t then it does so under an abstractor $[a]$.
- $s \approx_\alpha t$ means, intuitively, that s and t are α-equivalent, i.e, they are equal modulo the renaming of bound atoms.

These concepts are formally defined in Definitions 5 and 6.

Definition 5 (Freshness). *A freshness context ∇ is a set of constraints of the form $a\#X$. An atom a is said to be fresh on t under a context ∇, denoted by $\nabla \vdash a\#t$, if it is possible to build a proof using the rules:*

$$\frac{}{\nabla \vdash a\#\langle\rangle}\ (\#\langle\rangle) \qquad \frac{}{\nabla \vdash a\#b}\ (\#atom) \qquad \frac{(\pi^{-1}\cdot a\#X)\in\nabla}{\nabla\vdash a\#\pi\cdot X}\ (\#X)$$

$$\frac{}{\nabla\vdash a\#[a]t}\ (\#[a]a) \qquad \frac{\nabla\vdash a\#t}{\nabla\vdash a\#[b]t}\ (\#[a]b) \qquad \frac{\nabla\vdash a\#s \quad \nabla\vdash a\#t}{\nabla\vdash a\#\langle s,t\rangle}\ (\#pair)$$

$$\frac{\nabla\vdash a\#t}{\nabla\vdash a\#f\ t}\ (\#app) \qquad \frac{\nabla\vdash a\#s \quad \nabla\vdash a\#t}{\nabla\vdash a\#f^C\ \langle s,t\rangle}\ (\#c\text{-}app)$$

Example 3. Notice that $a\#X \vdash a\#\langle[a]\langle X,a\rangle,[b]h\langle X,b\rangle\rangle$, by application of rules $(\#pair)$, $(\#[a]a)$, $(\#[a]b)$, $(\#app)$, $(\#X)$ and $(\#atom)$.

Definition 6 (α-equality with commutative operators). *Two terms t and s are said to be α-equivalent under the freshness context Δ ($\Delta\vdash t\approx_\alpha s$) if it is possible to build a proof using the rules:*

$$\frac{\Delta\vdash s_0\approx_\alpha t_i,\ \ \Delta\vdash s_1\approx_\alpha t_{i+1(mod2)}}{\Delta\vdash f^C\langle s_0,s_1\rangle\approx_\alpha f^C\langle t_0,t_1\rangle}\ (\approx_\alpha C)_{\ i=0,1} \qquad \frac{}{\Delta\vdash a\approx_\alpha a}\ (\approx_\alpha atom)$$

$$\frac{\Delta\vdash s\approx_\alpha t}{\Delta\vdash f\ s\approx_\alpha f\ t}\ (\approx_\alpha app) \qquad \frac{\Delta\vdash s\approx_\alpha t}{\Delta\vdash [a]s\approx_\alpha [a]t}\ (\approx_\alpha [a]a)$$

$$\frac{\Delta\vdash s\approx_\alpha (a\ b)\cdot t,\ \ \Delta\vdash\ a\#t}{\Delta\vdash [a]s\approx_\alpha [b]t}\ (\approx_\alpha [a]b) \qquad \frac{ds(\pi,\pi')\#X\subseteq\Delta}{\Delta\vdash\pi\cdot X\approx_\alpha\pi'\cdot X}\ (\approx_\alpha var)$$

$$\frac{\Delta\vdash s_0\approx_\alpha t_0,\ \ \Delta\vdash s_1\approx_\alpha t_1}{\Delta\vdash\langle s_0,s_1\rangle\approx_\alpha\langle t_0,t_1\rangle}\ (\approx_\alpha pair) \qquad \frac{}{\Delta\vdash\langle\rangle\approx_\alpha\langle\rangle}\ (\approx_\alpha\langle\rangle)$$

Notation: *We define the difference set between two permutations π and π' as $ds(\pi,\pi')=\{a\in\mathcal{A}|\pi\cdot a\neq\pi'\cdot a\}$. By extension, $ds(\pi,\pi')\#X$ is the set containing every constraint of the form $a\#X$ for $a\in ds(\pi,\pi')$.*

Example 4. Notice that $[a]a\approx_\alpha[b]b$:

$$\frac{\dfrac{}{a\approx_\alpha(a\ b)\cdot b}\ (\approx_\alpha atom) \qquad \dfrac{}{a\#b}\ (\#atom)}{[a]a\approx_\alpha[b]b}\ (\approx_\alpha[a]b)$$

2.3 Nominal C-Unification

Definition 7 (Unification Problem). *A unification problem is a pair $\langle \nabla, P \rangle$ where ∇ is a freshness context and P is a finite set of equations and freshness constraints of the form $s \approx_? t$ and $a\#_? t$, respectively.*

Remark 5. Consider ∇ and ∇' freshness contexts and σ and σ' substitutions. We need the following notation to define a solution to a unification problem:

- $\nabla' \vdash \nabla\sigma$ denotes that $\nabla' \vdash a\#X\sigma$ holds for each $(a\#X) \in \nabla$.
- $\nabla \vdash \sigma \approx \sigma'$ denotes that $\nabla \vdash X\sigma \approx_\alpha X\sigma'$ for all X in $dom(\sigma) \cup dom(\sigma')$.

Definition 8 (Solution for a Triple or Problem). *Let δ be a substitution. A solution for a triple $\mathcal{P} = \langle \Delta, \delta, P \rangle$ is a pair $\langle \nabla, \sigma \rangle$ that fulfills the following four conditions:*

1. $\nabla \vdash \Delta\sigma$
2. $\nabla \vdash a\#t\sigma$, *if* $a\#_? t \in P$
3. $\nabla \vdash s\sigma \approx_\alpha t\sigma$, *if* $s \approx_? t \in P$
4. *There exists* λ *such that* $\nabla \vdash \delta\lambda \approx \sigma$

Then, a solution for a unification problem $\langle \Delta, P \rangle$ is a solution for the associated triple $\langle \Delta, id, P \rangle$.

Definition 9 (Fixed Point Equation). *An equation of the form $\pi \cdot X \approx_\alpha \pi' \cdot X$ is called a fixed point equation.*

Remark 6. Fixed point equations are not solved in C-unification because there is an infinite number of solutions to them. Instead, they are carried on, as part of the solution to the unification problem [1].

Remark 7. One of the original features of this work is the separate treatment of fixed point equations from the set of equational and freshness constraints. There is a trivial extension of Definition 8 in order to consider this detachment. Let FP be a set of fixed point equations. $\langle \nabla, \sigma \rangle$ is a solution to the quadruple $\mathcal{P} = \langle \Delta, \delta, P, FP \rangle$ if all conditions of Definition 8 are satisfied and additionally:

- $\nabla \vdash \pi \cdot X\sigma \approx_\alpha \pi' \cdot X\sigma$, if $\pi \cdot X \approx_? \pi' \cdot X \in FP$

Remark 8. The problem of nominal C-unification, as the problem of first-order C-unification is NP-complete (see [7] and [2]).

3 Specification

We developed a functional nominal C-unification algorithm for unifying the terms t and s. The algorithm is recursive and needs to keep track of the current context, the substitutions made so far, the remaining terms to unify and the current fixed point equations. Therefore, the algorithm receives as input a quadruple $(\Delta, \sigma, PrbLst, FPEqLst)$, where Δ is the context we are working with, σ is a list of the substitutions already done, $PrbLst$ is a list of unification problems which we must still unify (each equational constraint $t \approx_? s$ is represented as

a pair (t, s) in Algorithm 1) and $FPEqLst$ is a list of fixed point equations we have already computed.

The first call to the algorithm, in order to unify the terms t and s is simply: $\texttt{UNIFY}(\emptyset, id, [(t, s)], \emptyset)$. The algorithm eventually terminates, returning a list (possibly empty) of solutions, where each solution is of the form $(\Delta, \sigma, FPEqLst)$.

Although extensive, the algorithm is simple. It starts by analysing the list of terms it needs to unify. If $PrbLst$ is an empty list, then it has finished and can return the answer computed so far, which is the list: $[(\Delta, \sigma, FPEqLst)]$. If $PrbLst$ is not empty, then there are terms to unify, and the algorithm starts by trying to unify the terms t and s that are in the head of the list and only after that it goes to the tail of the list. The algorithm is recursive, calling itself on progressively simpler versions of the problem until it finishes.

3.1 Auxiliary Functions

Following the approach of [6], freshness constraints are treated separately from the main function. This has the advantage of making the main function `UNIFY` smaller, handling only equational constraints. To deal with the freshness constraints, the following auxiliary functions, which come from [6], were used:

- $\texttt{fresh_subs?}(\sigma, \Delta)$ returns the minimal context (Δ' in Algorithm 1) in which $a\#_? X\sigma$ holds, for every $a\#X$ in the context Δ.
- $\texttt{fresh?}(a, t)$ computes and returns the minimal context (Δ' in Algorithm 1) in which a is fresh for t.

Both functions also return a boolean (*bool*1 in Algorithm 1), indicating if it was possible to find the mentioned context.

3.2 Main Algorithm

The pseudocode of the algorithm is presented in Algorithm 1.

Remark 9. When trying to unify $f^C\langle t_1, t_2\rangle$ with $f^C\langle s_1, s_2\rangle$ there are two possible paths to take: try to unify t_1 with s_1 and t_2 with s_2, or try to unify t_1 with s_2 and t_2 with s_1. This means that there are two branches that we must consider, and since each branch can generate a solution, we may have more than one solution. This is the reason why the algorithm here presented gives a list of solutions as output. In nominal unification, by contrast, only one most general unifier is given as solution.

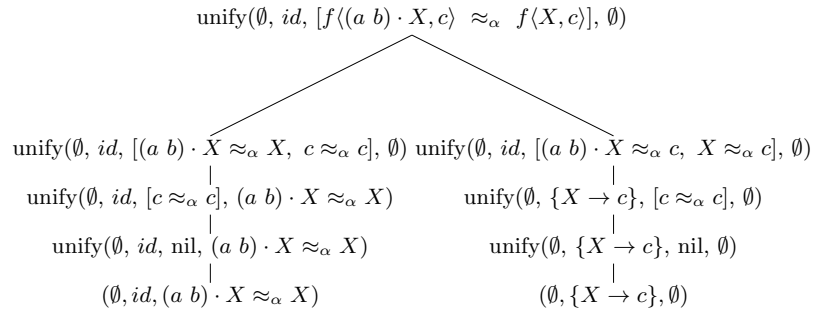
3.3 Examples

A simple example of the algorithm is given in Example 5. In this example, it is possible to see how commutativity introduces branches and how the algorithm calls itself with progressively simpler versions of the problem until it finishes. Example 6 is a slightly more complex example, which uses Example 5.

Algorithm 1 - First Part - Functional Nominal C-Unification

```
1: procedure UNIFY(Δ, σ, PrbLst, FPEqLst)
2:     if nil?(PrbLst) then
3:         return list((Δ, σ, FPEqLst))
4:     else
5:         cons((t, s), PrbLst′) = PrbLst
6:         if (s matches π · X) and (X not in t) then
7:             σ′ = {X → π^{-1} · t}
8:             σ″ = σ′ ∘ σ
9:             (Δ′, bool1) = fresh_subs?(σ′, Δ)
10:             Δ″ = Δ ∪ Δ′
11:             PrbLst″ = append((PrbLst′)σ′, (FPEqLst)σ′)
12:             if bool1 then return UNIFY(Δ″, σ″, PrbLst″, nil)
13:             else return nil
14:             end if
15:         else
16:             if t matches a then
17:                 if s matches a then
18:                     return UNIFY(Δ, σ, PrbLst′, FPEqLst)
19:                 else
20:                     return nil
21:                 end if
22:             else if t matches π · X then
23:                 if (X not in s) then
24:                                                 ▷ Similar to case above where s is a suspension
25:                 else if (s matches π′ · X) then
26:                     FPEqLst′ = FPEqLst ∪ { π · X ≈α π′ · X}
27:                     return UNIFY(Δ, σ, PrbLst′, FPEqLst′)
28:                 else return nil
29:                 end if
30:             else if t matches ⟨⟩ then
31:                 if s matches ⟨⟩ then
32:                     return UNIFY(Δ, σ, PrbLst′, FPEqLst)
33:                 else return nil
34:                 end if
35:             else if t matches ⟨t1, t2⟩ then
36:                 if s matches ⟨s1, s2⟩ then
37:                     PrbLst″ = cons((s1, t1), cons((s2, t2), PrbLst′))
38:                     return UNIFY(Δ, σ, PrbLst″, FPEqLst)
39:                 else return nil
40:                 end if
```

Example 5. Suppose f is a commutative function symbol. This example shows how the algorithm proceeds in order to unify $f\langle(a\ b)\cdot X, c\rangle$ with $f\langle X, c\rangle$.



Example 6. Suppose f and g are commutative function symbols, and h is a non-commutative function symbol. This example shows how the algorithm would

Algorithm 1 - Second Part - Functional Nominal C-Unification

41: **else if** t `matches` $[a]t_1$ **then**
42: **if** s `matches` $[a]s_1$ **then**
43: $PrbLst'' = cons((t_1, s_1), PrbLst')$
44: **return** UNIFY$(\Delta, \sigma, PrbLst'', FPEqLst)$
45: **else if** s `matches` $[b]s_1$ **then**
46: $(\Delta', bool1) = fresh?(a, s_1)$
47: $\Delta'' = \Delta \cup \Delta'$
48: $PrbLst'' = cons((t_1, \ (a\ b)\ s_1), PrbLst')$
49: **if** $bool1$ **then**
50: **return** UNIFY$(\Delta'', \sigma, PrbLst'', FPEqLst)$
51: **else return** nil
52: **end if**
53: **else return** nil
54: **end if**
55: **else if** t `matches` $f\ t_1$ **then** ▷ f is not commutative
56: **if** s `matches` $f\ s_1$ **then**
57: $PrbLst'' = cons((t_1, s_1), PrbLst')$
58: **return** UNIFY$(\Delta, \sigma, PrbLst'', FPEqLst)$
59: **else return** nil
60: **end if**
61: **else** ▷ t is of the form $f^C(t_1, t_2)$
62: **if** s `matches` $f^C(s_1, s_2)$ **then**
63: $PrbLst_1 = cons((s_1, t_1), cons((s_2, t_2), PrbLst'))$
64: $sol_1 =$ UNIFY$(\Delta, \sigma, PrbLst_1, FPEqLst)$
65: $PrbLst_2 = cons((s_1, t_2), cons((s_2, t_1), PrbLst'))$
66: $sol_2 =$ UNIFY$(\Delta, \sigma, PrbLst_2, FPEqLst)$
67: **return** APPEND(sol_1, sol_2)
68: **else return** nil
69: **end if**
70: **end if**
71: **end if**
72: **end if**
73: **end procedure**

unify $g\langle h\ d, f\langle (a\ b) \cdot X, c\rangle\rangle$ with $g\langle f\langle X, c\rangle, h\ d\rangle$. Because g is commutative, the algorithm explores two branches:

- On the first branch, the algorithm tries to unify $h\ d$ with $f\langle X, c\rangle$ and $f\langle (a\ b) \cdot X, c\rangle$ with $h\ d$. However, since it is impossible to unify $h\ d$ with $f\langle X, c\rangle$ (different function symbols), the algorithm returns an empty list, indicating that no solution is possible for this branch.
- On the second branch, the algorithm tries to unify $h\ d$ with $h\ d$ and $f\langle (a\ b) \cdot X, c\rangle$ with $f\langle X, c\rangle$. First, $h\ d$ unifies with $h\ d$ without any alterations on the context Δ, the substitution σ or the list of fixed point equations $FPEqLst$. Finally, the unification of $f\langle (a\ b) \cdot X, c\rangle$ with $f\langle X, c\rangle$ was shown in Example 5, and gives two solutions, which are also the solutions to this example: $(\emptyset, id, (a\ b) \cdot X \approx_\alpha X)$ and $(\emptyset, \{X \to c\}, \emptyset)$.

4 Formalization

4.1 Termination

Termination of the algorithm was proved by proving the type-correctness conditions (TCCs) generated by PVS [18]. In order to do that, a lexicographic measure

was defined:

$$lex2(|Vars(PrbLst) \cup Vars(FPEqLst)|, size(PrbLst)) \tag{3}$$

The first component in the lexicographic measure is the cardinality of the set of variables which occur in $PrbLst$ (the list of remaining unification problems) or in $FPEqLst$ (the list of fixed point equations). To compute the variables in a list, we consider the variables in all terms of the list. Finally, the variables in a term are computed recursively, as can be seen in Definition 10.

Definition 10 (Set of Variables). *The set of variables in a term is recursively defined as:*

$$\begin{array}{cc} Vars(a) = \emptyset & Vars(\langle\rangle) = \emptyset \\ Vars(\pi \cdot X) = \{X\} & Vars([a]t) = Vars(t) \\ Vars(\langle t_0, t_1\rangle) = Vars(t_0) \cup Vars(t_1) & Vars(f\ t) = Vars(t) \\ Vars(f^C\langle t_0, t_1\rangle) = Vars(t_0) \cup Vars(t_1) & \end{array}$$

The second component in the lexicographic measure is the sum of the size of every unification problem. To calculate the size of the unification problem (t, s), we only calculate the size of the first term t. This was an arbitrary choice, as the measure would still work if we had taken the size of s or even the size of t plus the size of s (in each recursive call, both the size of t and the size of s decrease). Finally, the size of t is computed recursively according to Definition 11.

Definition 11 (Size of Terms). *The size of terms is recursively computed as:*

$$\begin{array}{cc} size(a) = 1 & size(\langle\rangle) = 1 \\ size(\pi \cdot X) = 1 & size([a]t) = 1 + size(t) \\ size(\langle t_0, t_1\rangle) = 1 + size(t_0) + size(t_1) & size(f\ t) = 1 + size(t) \\ size(f^C\langle t_0, t_1\rangle) = 1 + size(t_0) + size(t_1) & \end{array}$$

The lexicographic measure decreases in each recursive call. The component that decreases depends on the type of the terms t and s that are in the head of the list of problems to unify. If one of them is a variable X, and we are not dealing with a fixed point equation, then the algorithm will instantiate this variable X, and the first component, $|Vars(PrbLst) \cup Vars(FPEqLst)|$, will decrease. In any other case, the second component, $size(PrbLst)$, will decrease.

Remark 10. It was possible to reduce the lexicographic measure used in [2], from 4 parameters to only 2 parameters. The measure adopted in [2] was:

$$|\mathcal{P}| = \langle |Var(P_{\approx})|, |P_{\approx}|, |P_{nfp}|, |P_{\#}| \rangle \tag{4}$$

where $P_{\approx}$ is the set of equation constraints in P, P_{nfp} is the set of non fixed point equations in P and $P_{\#}$ is the set of freshness constraints in P. Two ideas were used in order to accomplish this reduction. The first was to separate the treatment of freshness constraints from equational constraints, and treat the freshness constraints with the help of the auxiliary functions of 3.1. This idea comes from [6]. The second one is to separate the fixed point equations from the equational constraints. This way, when a fixed point equation is found in $PrbLst$, it is moved to $FPEqLst$, which makes $size(PrbLst)$ diminish.

4.2 Soundness and Completeness

To state the main theorems that allow us to prove soundness and completeness, we must first define the notion of a *valid* quadruple. A valid quadruple is an invariant of the `UNIFY` function in Algorithm 1 with useful properties.

Definition 12 (Valid Quadruple). *Let Δ be a freshness context, σ a substitution, P a list of unification problems and FP a list of fixed point equations. $\mathcal{P} = \langle \Delta, \sigma, P, FP \rangle$ is a valid quadruple if the following two conditions hold:*

- $Vars(im(\sigma)) \cap dom(\sigma) = \emptyset$ - $dom(\sigma) \cap (Vars(P) \cup Vars(FP)) = \emptyset$

where $im(\sigma)$ is the image of σ and $dom(\sigma)$ is the domain of σ.

Remark 11. A valid quadruple has two desirable properties: the substitution is idempotent (condition 1) and applying the substitution to P or FP produces no effect.

Soundness Corollary 1 states that `UNIFY` is sound. It follows directly by application of Theorem 1.

Theorem 1 (Main Theorem for Soundness of `UNIFY` Algorithm). *Suppose $(\Delta_{sol}, \sigma_{sol}, FPEqLst_{sol}) \in \texttt{UNIFY}(\Delta, \sigma, PrbLst,\ FPEqLst)$, (∇, δ) is a solution to $\langle \Delta_{sol}, \sigma_{sol}, \emptyset, FPEqLst_{sol} \rangle$ and $\langle \Delta, \sigma, PrbLst, FPEqLst \rangle$ is a valid quadruple. Then (∇, δ) is a solution to $\langle \Delta, \sigma, PrbLst, FPEqLst \rangle$.*

Proof. The proof is by induction on the lexicographic measure, according to the form of the terms t and s that are in the head of $PrbLst$, the list of remaining unification problems. The hardest cases are the ones of suspended variables and abstractions (see Remark 14). Below we explain the case of commutative functions.

In the case of commutative function symbols $f\langle t_1, t_2 \rangle$ and $f\langle s_1, s_2 \rangle$, there are no changes in the context or the substitution from one recursive call to the next. Therefore, it is trivial to check that we remain with a valid quadruple and it is also trivial to check all but the third condition of Definition 8. For the third condition we have either $\nabla \vdash t_1\delta \approx_\alpha s_1\delta$ and $\nabla \vdash t_2\delta \approx_\alpha s_2\delta$ or $\nabla \vdash t_1\delta \approx_\alpha s_2\delta$ and $\nabla \vdash t_2\delta \approx_\alpha s_1\delta$. In any case, we are able to deduce $\nabla \vdash (f\langle t_1, t_2 \rangle)\delta \approx_\alpha (f\langle s_1, s_2 \rangle)\delta$ by noting that $(f\langle t_1, t_2 \rangle)\delta = f\langle t_1\delta, t_2\delta \rangle$, $(f\langle s_1, s_2 \rangle)\delta = f\langle s_1\delta, s_2\delta \rangle$ and then using rule $(\approx_\alpha\ c - app)$ for alpha equivalence of commutative function symbols.

Corollary 1 (Soundness of `UNIFY` Algorithm). *Suppose (∇, δ) is a solution to $\langle \Delta_{sol}, \sigma_{sol}, \emptyset, FPEqLst_{sol} \rangle$, and $(\Delta_{sol}, \sigma_{sol}, FPEqLst_{sol}) \in \texttt{UNIFY}(\emptyset, id, [(t, s)], \emptyset)$. Then (∇, δ) is a solution to $\langle \emptyset, id, [(t, s)], \emptyset \rangle$.*

Proof. Notice that $\langle \emptyset, id, [(t, s)], \emptyset \rangle$ is a valid quadruple. Then, we apply Theorem 1 and prove the corollary.

Remark 12. An interpretation of Corollary 1 is that if (∇, δ) is a solution to one of the outputs of the algorithm `UNIFY`, then (∇, δ) is a solution to the original problem.

Completeness Corollary 2 states that UNIFY is complete. It follows directly by application of Theorem 2.

Theorem 2 (Main Theorem for Completeness of UNIFY). *Suppose* (∇, δ) *is a solution to* $\langle \Delta, \sigma, PrbLst, FPEqLst \rangle$ *and that* $\langle \Delta, \sigma, PrbLst, FPEqLst \rangle$ *is a valid quadruple. Then, there exists a computed output* $(\Delta_{sol}, \sigma_{sol}, FPEqLst_{sol}) \in$ UNIFY$(\Delta, \sigma, PrbLst, FPEqLst)$ *such that the solution* (∇, δ) *is also a solution to* $\langle \Delta_{sol}, \sigma_{sol}, \emptyset, FPEqLst_{sol} \rangle$.

Proof. The proof is by induction on the lexicographic measure, according to the form of the terms t and s that are in the head of $PrbLst$, the list of remaining unification problems. The hardest cases are again the ones of suspended variables and abstractions (see Remark 14). Below we explain the case of commutative functions.

In the case of commutative function symbols $f\langle t_1, t_2 \rangle$ and $f\langle s_1, s_2 \rangle$, there are no changes in the context or the substitution from one recursive call to the next. Therefore, it is trivial to check that we remain with a valid quadruple and it is also trivial to check all but the third condition of Definition 8. For the third condition we have $\nabla \vdash (f\langle t_1, t_2 \rangle)\delta \approx_\alpha (f\langle s_1, s_2 \rangle)\delta$ and must prove that either ($\nabla \vdash t_1\delta \approx_\alpha s_1\delta$ and $\nabla \vdash t_2\delta \approx_\alpha s_2\delta$) or ($\nabla \vdash t_1\delta \approx_\alpha s_2\delta$ and $\nabla \vdash t_2\delta \approx_\alpha s_1\delta$) happens. This again is solved by noting that $(f\langle t_1, t_2 \rangle)\delta = f\langle t_1\delta, t_2\delta \rangle$, $(f\langle s_1, s_2 \rangle)\delta = f\langle s_1\delta, s_2\delta \rangle$ and then using rule ($\approx_\alpha \ \ c - app$) for alpha equivalence of commutative function symbols.

Corollary 2 (Completeness of UNIFY). *Suppose* (∇, δ) *is a solution to the input quadruple* $\langle \emptyset, id, [(t, s)], \emptyset \rangle$. *Then, there exists* $(\Delta_{sol}, \sigma_{sol}, FPEqLst_{sol}) \in$ UNIFY$(\emptyset, id, [(t, s)], \emptyset)$ *such that* (∇, δ) *is a solution to* $\langle \Delta_{sol}, \sigma_{sol}, \emptyset, FPEqLst_{sol} \rangle$.

Proof. Notice that $\langle \emptyset, id, [(t, s)], \emptyset \rangle$ is a valid quadruple. Then, we apply Theorem 2 and prove the corollary.

Remark 13. An interpretation of Corollary 2 is that if (∇, δ) is a solution to the initial problem, then (∇, δ) is also a solution to one of the outputs of UNIFY.

5 Interesting Points of Formalization and Implementation

We discuss interesting points of the formalization and implementation here.

Remark 14. To prove correctness and completeness of the algorithm, we work with the terms t and s that are in the head of $PrbLst$. We divide the proof by cases. The most interesting case is when t or s is a suspension $\pi \cdot X$ and X does not occur in the other term (see Algorithm 1).

In this case, the algorithm receives as arguments Δ, σ, $PrbLst$ and $FPEqLst$ and the next recursive call is made with four different parameters: Δ'', σ'', $PrbLst''$ and nil (see Algorithm 1). Therefore, all of the four conditions of the Definition 8 are not trivially satisfied. Moreover, since in the next recursive call we will be working with a new substitution σ'' we must prove that the quadruple

we are working with remains valid (this is proved by noting that when a variable X is added to the domain of the substitution, all occurrences of it in $PrbLst$ and in $FPEqLst$ are instantiated, maintaining the validity of the quadruple).

The case of unifying $t = [a]t_1$ with $s = [b]s_1$ is also interesting, since both the context and the list of problems to unify suffer modifications in the recursive call. In all the remaining cases, there are no changes in the context nor in the substitution, making them easier. In these remaining cases, only one condition (the third) of the four in Definition 8 is not trivially satisfied.

Remark 15. Introducing commutative function symbols to the nominal unification algorithm presented in [6] meant we had to:

- Unify terms rooted by commutative function symbols (for instance, $f(t_1, t_2)$ with $f(s_1, s_2)$). First, the algorithm tries to unify t_1 with s_1 and t_2 with s_2, generating a list of solutions sol_1 (see Algorithm 1). Then, the algorithm tries unifying t_1 with s_2 and t_2 with s_1, generating a list of solutions sol_2. The final result is then simply the concatenation of both lists.
- Handle fixed point equations. This was also straightforward. We keep a separate list of fixed point equation ($FPEqLst$), and when the algorithm recognizes a fixed point equation in $PrbLst$ it takes this equation out of $PrbLst$ and puts it on $FPEqLst$.
- Define an appropriate data structure for the problem and the solutions. This was not straightforward. As mentioned before, since commutativity introduced branches, the recursive calls of the algorithm can be seen as a tree (see Example 5). Therefore, initially, an approach using a tree data structure was planned (which would have complicated the analysis). However, since the algorithm simply solves one branch and then the other, we realized all that was needed was to do two recursive calls (one for each branch) and append the two lists of solutions generated. Therefore, we were able to avoid the tree data structure, working instead with lists, which simplified the specification.

Remark 16. Since PVS does not support automatic extraction of Python code, the translation of the PVS specification for the Python implementation was done manually. Indeed, PVS provides extraction to Lisp and Clean (a dialect of Haskell). The code follows strictly the lines of the specification (Algorithm 1) with small adjustments, such as the inclusion of two parameters in the algorithm implementation, in order to support a verbose mode that prints the tree of recursive calls. This, and the representation of atoms, variables and terms in the implementation is discussed in the extended version. An OCaml implementation of a nominal C-unification algorithm was previously developed [2], but in contrast to the current Python implementation, the OCaml implementation does not correspond in a direct way to the formalized non-deterministic inductive specification [2].

6 Conclusion and Future Work

In this paper, we explained the problem of nominal C-unification and presented a functional algorithm for doing this task. We observed how nominal C-unification

has applications on logic programming languages and how the algorithm here presented could be straightforwardly converted to a matching algorithm, which in turn would have applications in nominal rewriting.

Our approach differs from the only other work in nominal C-unification ([2]) in two main points. First, we do not present a set of non-deterministic transformation rules, instead, we opt for a recursive specification, implemented in Python. Second, we follow the approach in [6] and deal with freshness contexts separately. This simplifies the main function and, along with the idea of using a different parameter to represent fixed point equations, allowed us to reduce the lexicographic measure used in [2] from four parameters to only two parameters, thus simplifying the formalizations of termination, soundness and completeness.

Finally, a future study would be extending the formalization to handle the more general case of Mal'cev permutative theories, which include n-ary functions with permutative arguments [13]. Other possible path, as indicated in [11], is expanding the algorithm to handle other equational theories such as unification modulo associative and associative-commutative function symbols (A- and AC-unification).

References

1. Ayala-Rincón, M., de Carvalho-Segundo, W., Fernández, M., Nantes-Sobrinho, D.: On Solving Nominal Fixpoint Equations. In: 11th Int. Symposium on Frontiers of Combining Systems (FroCoS). LNCS, vol. 10483, pp. 209–226. Springer (2017)
2. Ayala-Rincón, M., de Carvalho-Segundo, W., Fernández, M., Nantes-Sobrinho, D.: Nominal C-unification. In: 27th Int. Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2017), Revised Selected Papers. LNCS, vol. 10855, pp. 235–251. Springer (2018)
3. Ayala-Rincón, M., de Carvalho-Segundo, W., Fernández, M., Nantes-Sobrinho, D., Rocha-Oliveira, A.: A Formalisation of Nominal alpha-equivalence with A, C, and AC Function Symbols. Theoret. Comput. Sci. **781**, 3–23 (2019)
4. Ayala-Rincón, M., Fernández, M., Nantes-Sobrinho, D.: Nominal narrowing. In: 1st Int. Conference on Formal Structures for Computation and Deduction (FSCD). LIPIcs, vol. 52, pp. 11:1–11:17 (2016)
5. Ayala-Rincón, M., Fernández, M., Nantes-Sobrinho, D.: Fixed-point constraints for nominal equational unification. In: 3rd Int. Conference on Formal Structures for Computation and Deduction (FSCD). LIPIcs, vol. 108, pp. 7:1–7:16 (2018)
6. Ayala-Rincón, M., Fernández, M., Rocha-Oliveira, A.: Completeness in PVS of a nominal unification algorithm. Elect. Notes Theor. Comp. Sci. **323**, 57–74 (2016)
7. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge U.P. (1999)
8. Baader, F., Snyder, W.: Unification theory. In: Handbook of Automated Reasoning (in 2 volumes), pp. 445–532. Elsevier and MIT Press (2001)
9. Baumgartner, A., Kutsia, T., Levy, J., Villaret, M.: Nominal anti-unification. In: 26th Int. Conference on Rewriting Techniques and Applications (RTA). LIPIcs, vol. 36, pp. 57–73 (2015)
10. Calvès, C., Fernández, M.: A polynomial nominal unification algorithm. Theoret. Comput. Sci. **403**(2-3), 285–306 (2008)

11. de Carvalho Segundo, W.L.R.: Nominal Equational Problems Modulo Associativity, Commutativity and Associativity-Commutativity. Ph.D. thesis, Universidade de Brasilia (2019)
12. Cheney, J., Urban, C.: α-prolog: A logic programming language with names, binding and α-equivalence. In: 20th Int. Conference on Logic Programming (ICLP). LNCS, vol. 3132, pp. 269–283. Springer (2004)
13. Comon, H.: Complete Axiomatizations of Some Quotient Term Algebras. Theoret. Comput. Sci. **118**(2), 167–191 (1993)
14. Fernández, M., Gabbay, M.: Nominal rewriting. Information and Computation **205**(6), 917–965 (2007)
15. Kumar, R., Norrish, M.: (Nominal) Unification by Recursive Descent with Triangular Substitutions. In: First Int. Conference on Interactive Theorem Proving (ITP). LNCS, vol. 6172, pp. 51–66. Springer (2010)
16. Levy, J., Villaret, M.: Nominal unification from a higher-order perspective. In: 19th Int. Conference on Rewriting Techniques and Applications (RTA). LNCS, vol. 5117, pp. 246–260. Springer (2008)
17. Levy, J., Villaret, M.: An efficient nominal unification algorithm. In: Proceedings of the 21st Int. Conference on Rewriting Techniques and Applications (RTA). LIPIcs, vol. 6, pp. 209–226 (2010)
18. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS System Guide - Version 2.4 (2001), `http://pvs.csl.sri.com/documentation.shtml`
19. Pitts, A.: Nominal sets: Names and symmetry in computer science. Cambridge U.P. (2013)
20. Schmidt-Schauß, M., Kutsia, T., Levy, J., Villaret, M.: Nominal unification of higher order expressions with recursive let. In: 26th Int. Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016), Revised Selected Papers. LNCS, vol. 10184, pp. 328–344. Springer (2017)
21. Urban, C.: Nominal techniques in Isabelle/HOL. Journal of Automated Reasoning **40**(4), 327–356 (2008)
22. Urban, C., Pitts, A., Gabbay, M.: Nominal unification. Theoret. Comput. Sci. **323**(1-3), 473–497 (2004)

Modeling and Reasoning in Event Calculus Using Goal-Directed Constraint Answer Set Programming*

Joaquín Arias[1,2], Zhuo Chen[3], Manuel Carro[1,2], and Gopal Gupta[3]

[1] IMDEA Software Institute
[2] Universidad Politcnica de Madrid
joaquin.arias@imdea.org, manuel.carro@{imdea.org,upm.es}
[3] University of Texas at Dallas
{zhuo.chen,gupta}@utdallas.edu

Abstract. Automated commonsense reasoning is essential for building human-like AI systems featuring, for example, explainable AI. Event Calculus (EC) is a family of formalisms that model commonsense reasoning with a sound, logical basis. Previous attempts to mechanize reasoning using EC faced difficulties in the treatment of the continuous change in dense domains (e.g., time and other physical quantities), constraints among variables, default negation, and the uniform application of different inference methods, among others. We propose the use of s(CASP), a query-driven, top-down execution model for Predicate Answer Set Programming with Constraints, to model and reason using EC. We show how EC scenarios can be naturally and directly encoded in s(CASP) and how its expressiveness makes it possible to perform deductive and abductive reasoning tasks in domains featuring, for example, constraints involving dense time and fluents.

Keywords: ASP · goal-directed · Event Calculus · Constraints

1 Introduction

The ability to model continuous characteristics of the world is essential for Commonsense Reasoning (**CR**) in many domains that require dealing with continuous change: time, the height of a falling object, the gas level of a car, the water level in a sink, etc.

Event Calculus (**EC**) is a formalism based on many-sorted predicate logic [12, 22] that can represent continuous change and capture the commonsense law of inertia, whose modeling is a pervasive problem in CR. In EC, time-dependent properties and events are seen as objects and reasoning is performed on the truth values of properties and the occurrences of events at a point in time.

* Work partially supported by EIT Digital (https://eitdigital.eu), MINECO project TIN2015-67522-C3-1-R (TRACES), Comunidad de Madrid project S2018/TCS-4339 BLOQUES-CM co-funded by EIE Funds of the European Union, and US NSF Grant IIS 1718945.

Answer Set Programming (**ASP**) is a logic programming paradigm that was initially proposed to realize non-monotonic reasoning [16]. ASP has also been used to model the Event Calculus [14, 15]. Classical implementations of ASP are limited to variables ranging over discrete and bound domains and use mechanisms such as *grounding* and SAT solving to find out models (called *answer sets*) of ASP programs. However, non-monotonic reasoning often needs variables ranging over dense domains (e.g., those involving time or physical quantities) to faithfully represent the properties of these domains.

This paper presents an approach to modeling Event Calculus using s(CASP) [1] as the underlying reasoning infrastructure. The s(CASP) system is an implementation of Constraint Answer Set Programming over first-order predicates which combines ASP and constraints. It features predicates, constraints among non-ground variables, uninterpreted functions, and, most importantly, a top-down, query-driven execution strategy. These features make it possible to return answers with non-ground variables (possibly including constraints among them) and to compute partial models by returning only the fragment of a stable model that is necessary to answer a given query. Thanks to its interface with constraint solvers, sound non-monotonic reasoning with constraints is possible.

Our approach based on s(CASP) achieves more conciseness and expressiveness than other related approaches because dense domains can be faithfully modeled as continuous quantities, while in other proposals such domains [19, 15] had to be discretized, therefore losing precision or even soundness. Additionally, in our approach the amalgamation of ASP and constraints and its realization in s(CASP) is considerably more natural: under s(CASP), answer set programs are executed in a goal-directed manner so constraints encountered along the way are collected and solved dynamically as execution proceeds — this is very similar to the way in which Prolog has been extended with constraints. Implementations of other ASP systems featuring constraints are considerably more complex.

2 Related Work

Previous work translated *discrete* EC into ASP [14, 15] by reformulating the EC models as first-order stable models and translating the (almost universal) formulas of EC into a logic program that preserves stable models. Given a finite domain, EC2ASP (and its evolution, F2LP) compile (discrete) Event Calculus formulas into ASP programs [14, 15]. This translation scheme relies on two facts: second order circumscription and first order stable model semantics coincide on canonical formulas, and almost-universal formulas can be transformed into a logic program while the stable models are preserved. As a result, computing models of Event Calculus descriptions can be done by computing the stable models of an appropriately generated program under the stable model semantics.

Clearly, approaches featuring discrete domains cannot faithfully handle continuous quantities such as time. In addition, because of their reliance on SAT solvers to find the stable models, they can only handle *safe* programs. In contrast, the s(CASP) system, because of its direct support for predicates with

arbitrary terms, constructive negation, and the novel *forall mechanism* [17, 1], program safety is not a requirement. Thus, s(CASP) can model Event Calculus axioms much more directly and elegantly.

Action languages [9] were developed by Gelfond and Lifschitz to model the elements of natural language that are used to describe the effects of actions. They have been implemented using answer set programming to perform planning and diagnosis [7]. There have been extensions of action languages to accommodate time: for example, the action language C+ has been extended by Lee and Meng to accommodate continuous time [13].

The approaches mentioned above assume discrete quantities and do not support reasoning about continuous time or change. As long as SAT-based ASP systems are used to model Event Calculus, continuous fluents cannot be straightforwardly expressed since they require unbounded, dense domains for the variables. The work closest to incorporating continuous time makes use of SMT solvers. In this approach, constraints are incorporated into ASP and the grounded theory is executed using an SMT solver [13]. However, this approach has not been applied to modeling the Event Calculus.

Other ASP-based approaches to deal with planning in continuous domains include, for example, PDDL+ [5], which was developed to allow reasoning with continuous time-dependent effects. It models temporal behavior in terms of the initiation and termination of processes, which in turn act upon the numeric components of states. Processes are initiated and terminated instantaneously by actions or exogenous events. Continuous changes are made by concurrent processes. In PDDL+, reasoning is monotonic and thus the degree of elaboration tolerance is low. There are implementations of PDDL+ using constraint answer set programming (CASP) [2] though these have not been applied to modeling the Event Calculus.

EC can be written as a logic program, but it cannot be executed directly by Prolog [24], as it lacks some necessary features, such as constructive negation, deduction of negative literals, and (to some extent) detection of infinite failure [21]. A common approach is to write a metainterpreter specific to the EC variant at hand. This can be as complex as writing a (specialized) theorem prover or, more often, a specialized interpreter whose correctness is difficult to ascertain (see the code at [3]). Therefore, some Prolog implementations of EC do not completely formalize the calculus or implement a reduced version. In our case, we leverage on the capabilities of s(CASP) to provide constructive, sound negation, plus negative rule heads, and loop detection [1].

3 Background

Answer Set Programming is a logic programming and modelling language that evaluates normal logic programs under the stable model semantics [8]. s(ASP) [17] is a top-down, goal-driven ASP system that can evaluate ASP programs with function symbols (*functors*) **without** *grounding* them either before or during execution. Grounding is a procedure that substitutes program vari-

ables with the possible values from their domain. For most classical ASP solvers, grounding is a necessary pre-processing phase. Grounding, however, requires program variables to be restricted to take values in a finite domain. As a result, ASP solvers cannot be used to model continuous time or change.

3.1 s(CASP)

s(CASP) [1] extends s(ASP) similarly to how CLP extends Prolog. s(CASP) adds constraints to s(ASP); these constraints are kept and used both during execution and in the answer. Constraints have historically proved to be effective in improving both expressiveness and efficiency in logic programming, as constraints can succinctly express properties of a solution and reduce the search space. As a result, s(CASP) is more expressive and faster than s(ASP), while retaining the capability of executing non-ground predicate answer set programs. A s(CASP) program is a set of clauses of the following form:

$$\texttt{a :- } c_a\texttt{,}b_1\texttt{,...,}b_m\texttt{,not } b_{m+1}\texttt{,...,not } b_n\texttt{.}$$

where a and b_1, ..., b_n are atoms and not corresponds to *default* negation. The difference w.r.t. an ASP program is c_a, a simple constraint or a conjunction of constraints.

In s(CASP), and unlike Prolog's negation as failure, `not p(X)` can return bindings for `X` on success. This is possible thanks to the use of constructive negation [17] and coinduction [10]. This highlights two differences w.r.t. Prolog: first, s(CASP) resolves negated atoms not b_i against *dual rules* of the program [1, 17], which makes it possible for a non-ground call `not p(X)` to return the bindings for `X` for which the positive call `p(X)` would have failed, therefore supporting constructive negation. Second, the dual program is **not** interpreted under SLD semantics in order to handle the different kind of loops that can appear in s(CASP) [1, 17].

The execution of a s(CASP) program starts with a *query* of the form `?-` $c_q\texttt{,}l_1\texttt{,...,}l_m$, where l_i are (negated) literals and c_q is a conjunction of constraint(s). The answers to the query are partial stable models where each partial model is a subset of a stable model [8] including only the literals necessary to support the query (see [1] for details). Additionally, for each partial stable model s(CASP) returns on backtracking the justification tree and the bindings for the free variables of the query that correspond to the most general unifier (*mgu*) of a successful top-down derivation consistent with this stable model.

3.2 Event Calculus

EC is a formalism for reasoning about events and change [22], of which there are several axiomatizations. There are three basic, mutually related, concepts in EC: *events*, *fluents*, and *time points* (see Fig. 1). An event is an action or incident that may occur in the world: for instance, a person dropping a glass is an event. A fluent is a time-varying property of the world, such as the altitude of a glass. A time point is an instant of time. Events may happen at a time point;

Predicate	**Meaning**
$InitiallyN(f)$	fluent f is false at time 0
$InitiallyP(f)$	fluent f is true at time 0
$Happens(e,t)$	event e occurs at time t
$Initiates(e,f,t)$	if e happens at time t, f is true and not released from the commonsense law of inertia after t
$Terminates(e,f,t)$	if e occurs at time t, f is false and not released from the commonsense law of inertia after t
$Releases(e,f,t)$	if e occurs at time t, f is released from the commonsense law of inertia after t
$Trajectory(f_1,t_1,f_2,t_2)$	if f_1 is initiated by an event that occurs at t_1, then f_2 is true at t_2
$StoppedIn(t_1,f,t_2)$	f is stopped between t_1 and t_2
$StartedIn(t_1,f,t_2)$	f is started between t_1 and t_2
$HoldsAt(f,t)$	fluent f is true at time t

Fig. 1. Basic event calculus (BEC) predicates
(e = event, f, f_1, f_2 = fluents, t, t_1, t_2 = timepoints)

fluents have a truth value at any time point or over an interval, and these truth values are subject to change upon an occurrence of an event. In addition, fluents may have (continuous) quantities associated with them when they are true. For example, the event of dropping a glass initiates the fluent that captures that the glass is falling, and perhaps its height above the ground, and the event of holding a glass terminates the fluent that the glass is falling. An EC description consists of a universal theory and a domain narrative. The universal theory is a conjunction of EC axioms and the domain narrative consists of the causal laws of the domain and the known events and fluent properties.

Circumscription [18] is applied to EC domain narratives to minimize the extension of predicates and has two effects: the only events that happen are those defined and the only effects of events are those defined.

The original EC (OEC) was introduced by Kowalski and Sergot in 1986 [12]. OEC has sorts for event occurrences, fluents, and time periods. In this paper we use the Basic Event Calculus (BEC) formulated by Shanahan [21]. Fig. 1 summarizes the predicates in BEC, which are used in the seven BEC axioms (Fig. 2). An explanation of these axioms follows.

Axiom BEC1. A fluent f is stopped between time points t_1 and t_2 iff it is terminated or released by some event e that occurs after t_1 and before t_2.

Axiom BEC2. A fluent f is started between time points t_1 and t_2 iff it is initiated or released by some event e that occurs after t_1 and before t_2.

BEC1. $StoppedIn(t_1, f, t_2) \equiv$
$\exists e, t \ (\ Happens(e, t) \wedge t_1 < t < t_2 \wedge (\ Terminates(e, f, t) \vee Releases(e, f, t) \ ) \ )$

BEC2. $StartedIn(t_1, f, t_2) \equiv$
$\exists e, t \ (\ Happens(e, t) \wedge t_1 < t < t_2 \wedge (\ Initiates(e, f, t) \vee Releases(e, f, t) \ ) \ )$

BEC3. $HoldsAt(f_2, t_2) \leftarrow$
$Happens(e, t_1) \wedge Initiates(e, f_1, t_1) \wedge Trajectory(f_1, t_1, f_2, t_2) \wedge \neg StoppedIn(t_1, f_1, t_2)$

BEC4. $HoldsAt(f, t) \leftarrow InitiallyP(f) \wedge \neg StoppedIn(0, f, t)$

BEC5. $\neg HoldsAt(f, t) \leftarrow InitiallyN(f) \wedge \neg StartedIn(0, f, t)$

BEC6. $HoldsAt(f, t_2) \leftarrow$
$Happens(e, t_1) \wedge Initiates(e, f, t_1) \wedge t_1 < t_2 \wedge \neg StoppedIn(t_1, f, t_2)$

BEC7. $\neg HoldsAt(f, t_2) \leftarrow$
$Happens(e, t_1) \wedge Terminates(e, f, t_1) \wedge t_1 < t_2 \wedge \neg StartedIn(t_1, f, t_2)$

Fig. 2. Formalization of BEC axioms [22].

Axiom BEC3. A fluent f_2 is true at time t_2 if a fluent f_1 initiated at t_1 does not finish before t_2 and it makes fluent f_2 be true.[4]

Axiom BEC4. A fluent f is true at time t if it is true at time 0 and is not stopped on or before t.

Axiom BEC5. A fluent f is false at time t if it is false at time 0 and it is not started on or before t.

Axiom BEC6. A fluent f is true at time t_2 if it is initiated at some earlier time t_1 and it is not stopped before t_2.

Axiom BEC7. A fluent f is false at time t_2 if it is terminated at some earlier time t_1 and it is not started on or before t_2.

4 From Event Calculus to s(CASP)

4.1 Modeling EC with s(CASP)

Two key factors contribute to s(CASP)'s ability to model Event Calculus: the preservation of non-ground variables during the execution and the integration with constraint solvers.

Treatment of variables in s(CASP): Thanks to the usage of non-ground variables, s(CASP) is able to directly model Event Calculus axioms that would otherwise require "unsafe" rules. In classical ASP, a rule is safe when every variable that appears in its head or in a negated literal in its body also appears in a positive literal in the body of the rule, and it is unsafe otherwise. Safety

[4] For implementation convenience, and without loss of expressiveness, we assume that argument t_2 in $Trajectory(f_1, t_1, f_2, t_2)$ is not a time difference w.r.t. t_1, but an absolute time after t_1.

```
%% BEC1
stoppedIn(T1,F,T2) :-
    T1 .<. T, T .<. T2,
    terminates(E,F,T),
    happens(E,T).
stoppedIn(T1,F,T2) :-
    T1 .<. T, T .<. T2,
    releases(E,F,T),
    happens(E,T).
%% BEC2
startedIn(T1,F,T2) :-
    T1 .<. T, T .<. T2,
    initiates(E,F,T),
    happens(E,T).
startedIn(T1,F,T2) :-
    T1 .<. T, T .<. T2,
    releases(E,F,T),
    happens(E,T).
%% BEC3
holdsAt(F2,T2) :-
    initiates(E,F1,T1),
    happens(E,T1),
    trajectory(F1,T1,F2,T2),
    not stoppedIn(T1,F1,T2).
%% BEC4
holdsAt(F,T) :-
    0 .<. T, initiallyP(F),
    not stoppedIn(0,F,T).
%% BEC5
-holdsAt(F,T) :-
    0 .<. T, initiallyN(F),
    not startedIn(0,F,T).
%% BEC6
holdsAt(F,T) :-
    T1 .<. T, initiates(E,F,T1),
    happens(E,T1),
    not stoppedIn(T1,F,T).
%% BEC7
-holdsAt(F,T) :-
    T1 .<. T, terminates(E,F,T1),
    happens(E,T1),
    not startedIn(T1,F,T).
```

Fig. 3. Basic Event Calculus (BCE) modeled in s(CASP)

guarantees that every variable can be grounded. For example, BEC4 is unsafe since parameter t, that appears in the head, does not appear in a positive literal in the body (i.e., it only appears in $\neg StoppedIn(0, f, t)$). A SAT-based ASP solver such as Clingo [6] will not be able to directly process unsafe rules like this. However, the top-down strategy of the execution of s(CASP) makes it possible to keep logical variables both during execution and in answer sets and therefore free (logical) variables can be handled in heads and in negated literals.

Integration with constraint solvers: The s(CASP) system has a generic interface to enable plugging in constraint solvers. s(CASP) currently uses Holzbaur's CLP(Q) linear constraints solver [11], that supports the constraints $<, >, =, \neq, \leq, \geq$. As has been shown, all definitions and axioms in EC involve inequality comparisons over time points. The ability of s(CASP) to make use of constraint solvers makes it ideal to model continuous time in EC.

4.2 Translating BEC into s(CASP)

Our translation of EC description into s(CASP) program is similar to that of the systems EC2ASP and F2LP [14, 15], but we differ in some key aspects that improve performance and are relevant for expressiveness: *the treatment of rules with negated heads, the possibility of generating unsafe rules,* and *the use of constraints over rationals.* We describe below, with the help of a running example, the translation that turns logic statements (as found in BEC) into a s(CASP)

program. The code corresponding to the translations of the axioms of BEC in Fig. 2 can be found in Fig. 3. s(CASP) code follows the syntactical conventions of logic programming: constants (including function names) and predicate symbols start with a lowercase letter and variables start with an uppercase letter. In addition, constraints are written between two dots in s(CASP), (e.g., `.<.`) to make it clear that they do not correspond to Prolog's arithmetic comparisons.

4.3 Translation of the BEC theory

Atoms and Constants: Their names are preserved. *Uniqueness of Names* [23] is assumed by default (and enforced) in logic programming.

Constraints: Predicates that represent constraints (e.g., on time) are directly translated to their counterparts in s(CASP). E.g., $t_1 < t_2$ becomes `T1.<.T2`, where `.<.` is in our examples handled by a linear constraint solver. The translation (and s(CASP) itself) is parametric on the constraint domain.

Definitions: Axiomatizations of EC use definitions of the form $D(x) \equiv \exists y B(x, y)$, where $B(x, y)$ is a conjunction of (negated) atoms, disjunctions of atoms, and constraints (e.g., BEC1). The use of definitions makes it easier to build conceptual blocks out of basic predicates. However, for performance reasons we treat them as if they were written as $\forall x(D(x) \leftarrow \exists y B(x, y))$, following [20]. Intuitively, if we ignore the truth value of D in the (partial) models that s(CASP) generates, the models returned using implication and/or equivalence are the same, and the literal D can be ignored because it would anyway have disappeared had it been expanded. Additionally, s(CASP) internally performs Clark's completion [4] to the s(CASP) program, and therefore, we can assume that s(CASP) rules expresses all possible ways in which heads can be true.

Rules with Positive Heads: A rule (e.g., BEC6)

$$\forall x(H(x) \leftarrow \exists y(A(y) \wedge \neg B(x, y) \wedge x < y))$$

where $x < y$ is a constraint, is translated into

```
1  h(X) :- X .<. Y, a(Y), not b(X,Y).
```

Since s(CASP) performs left-to-right evaluation, placing constraints earlier in the rule is in general better, as constraint solvers are deterministic and constraining variables as soon as possible helps reduce the size of the search tree.

Rules with Negated Heads: BEC rules 5 and 7 infer negated heads ($\neg HoldsAt(f, t)$) while rule 4 and 6 infer positive heads ($HoldsAt(f, t)$), i.e., they have the scheme

$$\forall x(H(x) \leftarrow \exists y A(x, y)) \quad \wedge \quad \forall x(\neg H(x) \leftarrow \exists y B(x, y))$$

The standard approach to translate rules with negated heads is to convert them into global constraints [14]:

```
:- b(X,Y), h(X).
```

Our approach is to define instead a rule for the literal `-h(X)` that captures the explicit evidence that `h(X)` is false:

```
-h(X) :- b(X,Y).
```

which makes it possible to call `-h(X)` in a top-down execution. This construct was termed *classical* negation in [17] and behaves as a regular predicate, except that the s(CASP) compiler automatically adds the global constraint `:- -h(X), h(X)` to ensure that `-h(X)` and `h(X)` cannot be simultaneously true. Therefore, s(CASP) can detect an inconsistency (and return an empty model) if both $HoldsAt(f,t)$ and $\neg HoldsAt(f,t)$ can be simultaneously derived from an EC narrative. Since circumscription is not applied to the BEC theory, not being able to derive $HoldsAt(f,t)$ does not immediately determine that its negation is true. We will see how this is connected with the translation of the narrative.

Rules with Disjunctive Bodies: A rule (e.g., BEC1)

$$\forall x[H(x) \leftarrow \exists y(\ (A(x,y) \vee B(x,y)) \wedge C(x,y)\ )]$$

is translated into two separate clauses:

```
h(X) :- a(X,Y), c(X,Y).
h(X) :- b(X,Y), c(X,Y).
```

4.4 Translation of the narrative

The definition of a given scenario (its *narrative* part) states the basic actions and effects using the predicates in Fig. 1. EC assumes circumscription of the predicates defined in the *narrative*: the events (resp., effects) known to occur are the only events (resp., effects) that occur. Note that this is automatic in s(CASP), since it produces the Clark's completion of s(CASP) programs when generating the dual program. In addition, global constraints can restrict the admissible states of the system.

Every basic BEC predicate $P(x)$ (where P can stand for an event occurrence, an effect of an event on a fluent, etc.) is translated into a s(CASP) rule $P(x) \leftarrow \gamma$, where γ states **all** the cases where $P(x)$ is true. In many cases, these are *facts*, but in other cases γ captures the conditions for $P(x)$ to hold.

Let us consider example 14 in [22], which reasons about turning a light switch on and off.

Events: The description:

$$\begin{aligned} Happens(e,t) \quad \equiv\ & (e = TurnOn \wedge t = 1/2) \ \vee \\ & (e = TurnOff \wedge t = 4) \end{aligned}$$

states that the $TurnOn$ event will happen exclusively at time $t = 1/2$ and that $TurnOff$ will happen exclusively at $t = 4$. This is translated into:

```
happens(turn_on, 1/2).
happens(turn_off, 4).
```

Event effects: When the event $TurnOn$ happens, the light is put in *on* status; similarly, when event $TurnOff$ happens, the status *on* of the light is terminated. In both cases, this can happen at any time t:

```
initiates(turn_on, light_on, T).
terminates(turn_off, light_on, T).
```

Release from Inertia: When turned on, the light emits red light for 2 seconds, and then green light is emitted. $Trajectory$ expresses how this change depends on the time elapsed since an event occurrence. *Releases* states that the color of the light is released from the commonsense law of inertia. After a fluent is released, its truth value is not determined by BEC and is permitted to vary. Thus, there may be models in which the fluent is true, and models in which the fluent may be false. Releasing a fluent frees it up so that other axioms in the domain description can be used to determine its truth value, thus allowing us to represent continuous change using $Trajectory$.

```
trajectory(light_on, T1, light_red, T2) :-
    T2 .>=. T1, T2 .<. T1 + 2.
trajectory(light_on, T1, light_green, T2) :-
    T2 .>=. T1 + 2.
releases(turn_on, light_red, T).
releases(turn_on, light_green, T).
```

The $Trajectory$ formula has the shape $P(x) \leftarrow \gamma$, as we need to state the (time) conditions for the fluent to become activated.

A Note on using $\neg HoldsAt(f,t)$ in γ: The basic BEC predicates may depend on what the BEC theory can deduce, e.g., γ may depend on $HoldsAt(f,t)$ or $\neg HoldsAt(f,t)$ (see Fig. 4). $HoldsAt(f,t)$ would be invoked directly, but $\neg HoldsAt(f,t)$ would be called using classical negation, e.g., `-holdsAt(F,T)`. The reason is that since EC does not apply circumscription to its axioms, we can deduce only the truth (or falsehood) of a predicate when we have direct evidence of either of them — i.e., what the positive (`holdsAt(F,T)`) and negative (`-holdsAt(F,T)`) heads provide.

State Constraint: State constraints usually contain $HoldsAt(f,t)$ or $\neg HoldsAt(f,t)$ and represent restrictions on the models. In our running example, a light cannot be red and green at the same time: $\forall t.\neg(HoldsAt(Red,t) \wedge HoldsAt(Green,t))$. This is translated as:

```
:- holdsAt(light_red,T), holdsAt(light_green,T).
```

4.5 Continuous Change: A Complete Encoding

We consider now an example from [23]: a water tap fills a vessel, whose water level is subject to continuous change. When the level reaches the bucket rim, it starts spilling. We will present the main ideas behind its encoding (Fig. 4) and will show some queries we can ask about its state and behavior.

```
1  #include bec_theory.
2
3  max_level(10):- not max_level(16).
4  max_level(16):- not max_level(10).
5
6  initiallyP(level(0)).
7  happens(overflow,T).
8  happens(tapOn,5).
9
10 initiates(tapOn,filling,T).
11 terminates(tapOff,filling,T).
12 initiates(overflow,spilling,T):-
13    max_level(Max),
14    holdsAt(level(Max), T).
15 releases(tapOn,level(0),T):-
16    happens(tapOn,T).
17
18 trajectory(filling,T1,level(X2),T2):-
19    T1 .<. T2, X2 .=. X + T2-T1,
20    max_level(Max), X2 .=<. Max,
21    holdsAt(level(X),T1).
22 trajectory(filling,T1,level(overflow),T2):-
23    T1 .<. T2, X2 .=. X + T2-T1,
24    max_level(Max), X2 .>. Max,
25    holdsAt(level(X),T1).
26 trajectory(spilling,T1,leak(X),T2):-
27    holdsAt(filling, T2),
28    T1 .<. T2, X .=. T2-T1.
```

Fig. 4. Encoding of an Event Calculus narrative with continuous change

Continuous Change: The fluent $Level(x)$ represents that the water is at level x in the vessel. The first $Trajectory$ formula (lines 18-21) determines the time-dependent value of the $Level(x)$ fluent,[5] which is active as long as the $Filling$ fluent is true and the rim of the vessel is not reached. Additionally, the second $Trajectory$ formula (lines 22-25) allows us to capture the fact that the water reached the rim of the vessel and overflowed.

Triggered Fluent: The fluent *Spilling* is triggered (lines 12-14) when the water level reaches the rim of the vessel. As a consequence, the $Trajectory$ formula in lines 26-28 starts the fluent $Leak(x)$ and captures the amount of water leaked while the fluent *Spilling* holds.

Different Worlds: The clauses in lines 3-4 force the vessel capacity to be either 10 or 16 — i.e., they create two possible worlds/models: {`max_level(10), not max_level(16), ...`} and {`max_level(16), not max_level(10), ...`}. The same mechanism can be used to state whether an event happens or not. For this, a keyword `#abducible` is provided as a shortcut in s(CASP). We will use it in the *Abduction* subsection later on.

5 Examples and Evaluation

The benchmarks used in this section are available at `https://goo.gl/jzUw46`. They were run on a MacOS 10.14.3 laptop with an Intel Core i5 at 2GHz.

[5] For simplicity the amount of water filled/leaked correspond directly to how long the water has been pouring in / spilling from the vessel.

Deduction: Deduction determines whether a state of the world is possible given a theory (in our case, BEC) and an initial narrative. We can perform deduction in BEC for the previous examples through queries to the corresponding s(CASP) program. For the lights scenario:

- `?- holdsAt(light_on,2)` succeeds: it deduces that the light is on at time 2.
- `?- -holdsAt(light_on,5)` succeeds: the light is not on at time 5.
- `?- holdsAt(F,2)` is true in one stable model containing `holdsAt(light_red,2)` and `holdsAt(light_on,2)`, meaning that at time 2, the light is on and its color is red.

In the water level scenario (Fig. 4) we can make queries involving time and the water level:

- `?- holdsAt(level(H),15/2)` is true when `H=5/2`.
- `?- holdsAt(level(5/2),T)` is true when `T=15/2`.

Note that s(CASP) can operate and answer correctly queries involving rationals without having to modify the original program to introduce domains for the relevant variables or to *scale* the constants to convert rationals into integers.

Abduction: Abductive reasoning tries to determine a sequence of events/actions that reaches a final state. In the case of ASP, actions are naturally captured as the set of atoms that are true in a model which includes the initial and final states and are consistent with BEC. For the water scenario, (Fig. 4), let us assume we want to reach water level 14 at time 19. The query `?- holdsAt(level(14),19)` will return a single model with a vessel size of 16 and the rest of the atoms in the model capture what must (not) happen to reach this state.

More interesting abductive tasks can be performed: adding the line `#abducible happens(tapOff,U)` to the program, we specify that it is possible (but not necessary) for the tap to close at some time `U`. As we mentioned in Section 4.5, this directive is translated into code that creates different worlds/-models. The query `?- holdsAt(spilling,T)` determines if the water may overspill and under which conditions. s(CASP) returns two models:

- One containing `T>15, holdsAt(spilling,T), happens(tapOn,5), 5<U<15, not happens(TapOff,U), max_level(10)` meaning that the water will spill at `T=15` if the vessel has a capacity of 10, the tap is open at `T=5`, and it is **not** closed between times 5 and 15.
- Another similar model, with the water spilling at `T=21` in a vessel with capacity of 16 and where the tap was not closed before `U=21`.

Note that s(CASP) determined the truth value of *Happens* and, more importantly, performed constraint solving to infer the time ranges during which some events ought (and ought not) to take place, represented by the negated atoms in the models inferred by constructive negation. Since all relevant atoms have a time parameter, they actually represent a *timed plan*. Due to the expressiveness of constraints, this plan contains information on time points when

events must (not) happen and also on time *windows* (sometimes in relation with other events) during which events must (not) take place. Note that it would be impossible to (finitely) represent this interval with grounded atoms, as it would correspond to an infinite number of points in time.

Evaluation: Comparing directly implementations of EC in s(CASP) with implementations in other ASP systems is not easy: most ASP systems do not support continuous quantities and the technological gap is too wide: grounding-based ASP have been in production for years. We will instead evaluate the advantage of providing constraints over continuous domains by comparing s(CASP) w.r.t. s(ASP) [17], an implementation of equivalent technological maturity that does not provide constraints. In order to execute comparable programs, we produced versions of programs that check conditions when variables are instantiated enough and simulate grounding by making the domains of some variables (such as those expressing time) explicit, emulating ASP declarations such as `time(1..5)`. Variables can then be instantiated on backtracking. Queries involving rational numbers need a specific version of the programs that use smaller steps to enumerate the variables. We want to emphasize that these adjustments are unnecessary in s(CASP).

The results for several queries (Table 1) show that s(CASP) is significantly faster than s(ASP). Note that using rationals instead of integers in s(CASP) does not incur any noticeable overhead. In the s(ASP) case, discretization requires fractional steps (e.g., a step of 0.25 is required for the last query), with a considerable negative impact on execution time. Note that selecting the right step size to discretize domains depends on both the query and the program, adding additional complications to the problem of approximating continuous domains with discrete variables.

Table 1. Run time (ms) comparison for the light scenario.

Queries	s(CASP)	s(ASP)
`holdsAt(light_on,2)`	**233**	8,320
`-holdsAt(light_on,5)`	**314**	7,952
`holdsAt(light_red,2)`	**231**	8,214
`holdsAt(light_green,3)`	**229**	8,577
`holdsAt(light_red,9/4)`	**234**	51,548

6 Conclusions

We showed how Event Calculus can be modeled in s(CASP), a goal-directed implementation of constraint answer set programming with predicates, with

much fewer limitations than other approaches. s(CASP) can capture the notion of continuous time (and, in general, fluents) in Event Calculus thanks to its grounding-free top-down evaluation strategy. It can also represent complex models and answer queries in a flexible manner thanks to the use of constraints.

The main contribution of the paper is to show how Event Calculus can be directly modeled using s(CASP), and ASP system that seamlessly supports constraints. The modeling of the Event Calculus using s(CASP) is more elegant and faithful to the original axioms compared to other approaches such as F2LP, where time has to be discretized. While other approaches such as ASPMT do support continuous domains, their reliance on SMT solvers makes their implementation really complex as associations among variables are lost during grounding. The use of s(CASP) brings other advantages: for example, the justification for the answers to a query is obtained for free, since in a query-driven system, the justification is merely the trace of the proof. Likewise, explanations for observations via abduction are also generated for free, thanks to the goal-directed, top-down execution of s(CASP).

To the best of the authors' knowledge, our approach is the only one that faithfully models continuous-time Event Calculus under the stable model semantics. All other approaches discretize time and thus do not model EC in a sound manner. Our approach supports both deduction and abduction with little or no additional effort.

The work reported in this paper can be seen as the first serious application of s(CASP) [1]. It illustrates the advantages that goal-directed ASP systems have over grounding and SAT solver-based ones for certain applications. Our future work includes applying the s(CASP) system to solving planning problems where a generated plan must obey real-time constraints.

References

1. Arias, J., Carro, M., Salazar, E., Marple, K., Gupta, G.: Constraint Answer Set Programming without Grounding. Theory and Practice of Logic Programming **18**(3-4), 337–354 (2018). https://doi.org/10.1017/S1471068418000285
2. Balduccini, M., Magazzeni, D., Maratea, M.: PDDL+ planning via constraint answer set programming. In: 9th Workshop on Answer Set Programming and Other Computing Paradigms (October 2016), http://arxiv.org/abs/1609.00030
3. Chittaro, L., Montanari, A.: Efficient Temporal Reasoning in the Cached Event Calculus. Computational Intelligence **12**, 359–382 (1996). https://doi.org/10.1111/j.1467-8640.1996.tb00267.x
4. Clark, K.L.: Negation as Failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Springer (1978)
5. Fox, M., Long, D.: PDDL+: Modeling continuous time dependent effects. In: Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space. vol. 4, p. 34 (2002)
6. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Clingo = ASP + Control: Preliminary Report. arXiv preprint arXiv:1405.3694 (2014)
7. Gelfond, M., Kahl, Y.: Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach. Cambridge University Press (2014)

8. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: 5th International Conference on Logic Programming. pp. 1070–1080 (1988)
9. Gelfond, M., Lifschitz, V.: Representing Action and Change by Logic Programs. The Journal of Logic Programming **17**(2-4), 301–321 (1993)
10. Gupta, G., Bansal, A., Min, R., Simon, L., Mallya, A.: Coinductive Logic Programming and its Applications. In: 23rd Int'l. Conference on Logic Programming. pp. 27–44. Springer (2007)
11. Holzbaur, C.: OFAI CLP(Q,R) Manual, Edition 1.3.3. Tech. Rep. TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna (1995)
12. Kowalski, R., Sergot, M.: A Logic-Based Calculus of Events. In: Foundations of knowledge base management, pp. 23–55. Springer (1989)
13. Lee, J., Meng, Y.: Answer set programming modulo theories and reasoning about continuous changes. In: IJCAI 2013. pp. 990–996 (2013)
14. Lee, J., Palla, R.: Reformulating the Situation Calculus and the Event Calculus in the General Theory of Stable Models and in Answer Set Programming. J. of Artif. Intell. Research **43**, 571–620 (2012)
15. Lee, J., Palla, R.: F2LP: Computing Answer Sets of First Order Formulas. http://reasoning.eas.asu.edu/f2lp/ (February 2019), last accessed: Feb. 22, 2019
16. Lifschitz, V.: What Is Answer Set Programming? In: 23rd National Conference on Artificial Intelligence. vol. 3, pp. 1594–1597. AAAI Press (2008)
17. Marple, K., Salazar, E., Gupta, G.: Computing Stable Models of Normal Logic Programs Without Grounding. CoRR **eprint arXiv:1709.00501** (2017), http://arxiv.org/abs/1709.00501
18. McCarthy, J.: Circumscription - A Form of Non-Monotonic Reasoning. Artificial Intelligence **13**(1-2), 27–39 (1980). https://doi.org/10.1016/0004-3702(80)90011-9
19. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating Answer Set Programming and Constraint Logic Programming. Annals of Mathematics and Artificial Intelligence **53**(1-4), 251–287 (2008)
20. Mueller, E.T.: Event Calculus Answer Set Programming. http://decreasoner.sourceforge.net/csr/ecasp/index.html (August 2014), accessed on Feb. 3, 2019
21. Mueller, E.T.: Chapter 17: Event calculus. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of Knowledge Representation, Foundations of AI, vol. 3, pp. 671 – 708. Elsevier (2008). https://doi.org/10.1016/S1574-6526(07)03017-9
22. Mueller, E.T.: Commonsense reasoning: an event calculus based approach. Morgan Kaufmann (2014)
23. Shanahan, M.: The Event Calculus Explained. In: Artificial Intelligence Today, pp. 409–430. Springer (1999). https://doi.org/10.1007/3-540-48317-9_17
24. Shanahan, M.: An Abductive Event Calculus Planner. The Journal of Logic Programming **44**(1-3), 207–240 (2000)

Blockchain Superoptimizer*

Julian Nagele[1][0000−0002−4727−4637] and Maria A Schett[2][0000−0003−2919−5983]

[1] Queen Mary University of London, UK mail@jnagele.net
[2] University College London, UK mail@maria-a-schett.net

Abstract. In the blockchain-based, distributed computing platform Ethereum, programs called smart contracts are compiled to bytecode and executed on the Ethereum Virtual Machine (EVM). Executing EVM bytecode is subject to monetary fees—a clear optimization target. Our aim is to superoptimize EVM bytecode by encoding the operational semantics of EVM instructions as SMT formulas and leveraging a constraint solver to automatically find cheaper bytecode. We implement this approach in our EVM Bytecode SuperOptimizer ebso and perform two large scale evaluations on real-world data sets.

Keywords: Superoptimization, Ethereum, Smart Contracts, SMT

1 Introduction

Ethereum is a blockchain-based, distributed computing platform featuring a quasi-Turing complete programming language. In Ethereum, programs are called smart contracts, compiled to bytecode and executed on the Ethereum Virtual Machine (EVM). In order to avoid network spam and to ensure termination, execution is subject to monetary fees. These fees are specified in units of *gas*, *i.e.*, any instruction executed on the EVM has a cost in terms of gas, possibly depending on its input and the execution state.

Example 1. Consider the expression $3+(0-x)$, which corresponds to the program `PUSH 0 SUB PUSH 3 ADD`. The EVM is a stack-based machine, so this program takes an argument x from the stack to compute the expression above. However, clearly one can save the `ADD` instruction and instead compute $3 - x$, *i.e.*, optimize the program to `PUSH 3 SUB`. The first program costs 12 g to execute on the EVM, while the second costs only 6 g.

We build a tool that automatically finds this optimization and similar others that are missed by state-of-the-art smart contract compilers: the **E**VM **b**ytecode super**o**ptimizer ebso. The use of ebso for Example 1 is sketched in Figure 1. To find these optimizations, ebso implements *superoptimization*. Superoptimization is often considered too slow to use during software development except for special circumstances. We argue that compiling smart contracts is such a circumstance.

* This research is supported by the UK Research Institute in Verified Trustworthy Software Systems and partially supported by funding from Google.

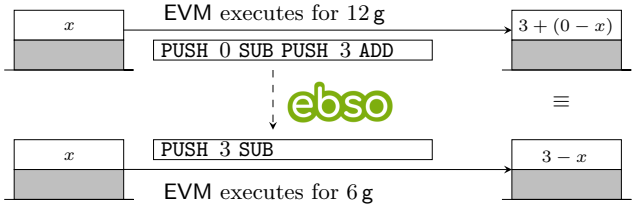


Fig. 1: Overview over ebso.

Since bytecode, once it has been deployed to the blockchain, cannot change again, spending extra time optimizing a program that may be called many times, might well be worth it. Especially, since it is very clear what "worth it" means: the clear cost model of gas makes it easy to define optimality.[3]

Our main contributions are: *(i)* an SMT encoding of a subset of EVM bytecode semantics (Section 4), *(ii)* an implementation of two flavors of superoptimization: *basic*, where the constraint solver is used to check equivalence of enumerated candidate instruction sequences, and *unbounded*, where also the enumeration itself is shifted to the constraint solver (Section 5), and *(iii)* two large scale evaluations (Section 6). First, we run ebso on a collection of smart contracts from a programming competition aimed at producing the cheapest EVM bytecode for given programming challenges. Even in this already highly optimized data set ebso still finds 19 optimizations. In the second evaluation we compare the performance of basic and unbounded superoptimization on the 2500 most called smart contracts from the Ethereum blockchain and find that, in our setting, unbounded superoptimization outperforms basic superoptimization.

2 Ethereum and the EVM

Smart contracts in Ethereum are usually written in a specialized high-level language such as Solidity or Vyper and then compiled into *bytecode*, which is executed on the EVM. The EVM is a virtual machine formally defined in the Ethereum yellow paper [14]. It is based on a *stack*, which holds *words*, *i.e.*, bit vectors, of size 256.[4] The maximal *stack size* is set to 2^{10}. Pushing words onto a full stack leads to a *stack overflow*, while removing words from the empty stack leads to a *stack underflow*. Both lead the EVM to enter an *exceptional halting* state. The EVM also features a volatile *memory*, a word-addressed byte array, and a persistent key-value *storage*, a word-addressed word array, whose contents are stored on the Ethereum blockchain. The bytecode directly corresponds to

[3] Of course setting the gas price of individual instructions, such that it accurately reflects the computational cost is hard, and has been a problem in the past see *e.g.* news.ycombinator.com/item?id=12557372.

[4] This word size was chosen to facilitate the cryptographic computations such as hashing that are often performed in the EVM.

more human-friendly *instructions*. For example, the EVM bytecode `6029600101` encodes the following sequence of instructions: `PUSH` 41 `PUSH` 1 `ADD`. Instructions can be classified into different categories, such as *arithmetic* operations, *e.g.* `ADD` and `SUB` for addition and subtraction, *comparisons*, *e.g.* `SLT` for signed less-than, and *bitwise operations*, like `AND` and `NOT`. The instruction `PUSH` pushes a word onto the stack, while `POP` removes the top word.[5] Words on the stack can be duplicated using `DUP`i and swapped using `SWAP`i for $1 \leqslant i \leqslant 16$, where i refers to the ith word below the top. Some instructions are specific to the blockchain domain, like `BLOCKHASH`, which returns the hash of a recently mined block, or `ADDRESS`, which returns the address of the currently executing account. Instructions for control flow include *e.g.* `JUMP`, `JUMPDEST`, and `STOP`.

We write $\delta(\iota)$ for the number of words that instruction ι takes from the stack, and $\alpha(\iota)$ for the number of words ι adds onto the stack. A *program* p is a finite sequence of instructions. We define the *size* $|p|$ of a program as the number of its instructions. To execute a program on the Ethereum blockchain, the caller has to pay *gas*. The amount to be paid depends on both the instructions of the program and the input: every instruction comes with a *gas cost*. For example, `PUSH` and `ADD` currently cost 3 g, and therefore executing the program above costs 9 g. Most instructions have a fixed cost, but some take the current state of the execution into account. A prominent example of this behavior is storage. Writing to a zero-valued key conceptually allocates new storage and thus is more expensive than writing to a key that is already in use, *i.e.*, holds a non-zero value. The gas prices of all instructions are specified in the yellow paper [14].

3 Superoptimization

Given a *source* program p superoptimization tries to generate a *target* program p' such that *(i)* p' is equivalent to p, and *(ii)* the cost of p' is minimal with respect to a given cost function C. This problem arises in several contexts with different source and target languages. In our case, *i.e.*, for a binary recompiler, both source and target are EVM bytecode.

A standard approach to superoptimization and synthesis [4, 9, 12, 13] is to search through the space of *candidate instruction* sequences of increasing cost and use a constraint solver to check whether a candidate correctly implements the source program. The solver of choice is usually a Satisfiability Modulo Theories (SMT) solver, which operates on first-order formulas in combination with background theories, such as the theory of bit vectors or arrays. Modern SMT solvers are highly optimized and implement techniques to handle arbitrary first-order formulas, such as E-matching. With increasing cost of the candidate sequence, the search space dramatically increases. To deal with this explosion one idea is to hand some of the search to the solver, by using templates [4, 13]. Templates leave holes in the target program, *e.g.* for immediate arguments of instructions, that the solver must then fill. A candidate program is correct if the

[5] We gloss over the 32 different `PUSH` instructions depending on the size of the word to be pushed.

```
1: function BasicSo(p_s, C)
2:   n ← 0
3:   while true do
4:     for all p_t ∈ {p | C(p) = n} do
5:       χ ← EncodeBso(p_s, p_t)
6:       if Satisfiable(χ) then
7:         m ← GetModel(χ)
8:         p_t ← DecodeBso(m)
9:         return p_t
10:    n ← n + 1
```

(a) Basic Superoptimization.

```
1: function UnboundedSo(p_s, C)
2:   p_t ← p_s
3:   χ ← EncodeUso(p_t) ∧ Bound(p_t, C)
4:   while Satisfiable(χ) do
5:     m ← GetModel(χ)
6:     p_t ← DecodeUso(m)
7:     χ ← χ ∧ Bound(p_t, C)
8:   return p_t
```

(b) Unbounded Superoptimization.

Alg. 2: Superoptimization.

encoding is satisfiable, *i.e.*, if the solver finds a model. Constructing the target program then amounts to obtaining the values for the templates from the model. This approach is shown in Algorithm 2(a).

Unbounded superoptimization [5, 6] pushes this idea further. Instead of searching through candidate programs and calling the SMT solver on them, it shifts the search into the solver, *i.e.*, the encoding expresses all candidate instruction sequences of any length that correctly implement the source program. This approach is shown in Algorithm 2(b): if the solver returns satisfiable then there is an instruction sequence that correctly implements the source program. Again, this target program is reconstructed from the model. If successful, a constraint asking for a cheaper program is added and the solver is called again. Note that this also means that unbounded superoptimization can stop with a correct, but possibly non-optimal solution. In contrast, basic superoptimization cannot return a correct solution until it has finished.

The main ingredients of superoptimization in Algorithm 2 are EncodeBso/Uso producing the SMT encoding, and DecodeBso/Uso reconstructing the target program from a model. We present our encodings for the semantics of EVM bytecode in the following section.

4 Encoding

We start by encoding three parts of the EVM execution state: *(i)* the stack, *(ii)* gas consumption, and *(iii)* whether the execution is in an exceptional halting state. We model the stack as an uninterpreted function together with a counter, which points to the next free position on the stack.

Definition 1. *A state* $\sigma = \langle \mathsf{st}, \mathsf{c}, \mathsf{hlt}, \mathsf{g} \rangle$ *consists of*

(i) *a function* $\mathsf{st}(\mathcal{V}, j, n)$ *that, after the program has executed* j *instructions on input variables from* $\mathcal{V}$ *returns the word from position* n *in the stack,*
(ii) *a function* $\mathsf{c}(j)$ *that returns the number of words on the stack after executing* j *instructions. Hence* $\mathsf{st}(\mathcal{V}, j, \mathsf{c}(j) - 1)$ *returns the top of the stack.*

(iii) *a function* $\mathsf{hlt}(j)$ *that returns true* ($\top$) *if exceptional halting has occurred after executing* j *instructions, and false* ($\bot$) *otherwise.*
(iv) *a function* $\mathsf{g}(\mathcal{V}, j)$ *that returns the amount of gas consumed after executing* j *instructions.*

Here the functions in σ represent *all* execution states of a program, indexed by variable j.

Example 2. Symbolically executing the program `PUSH` 41 `PUSH` 1 `ADD` using our representation above we have

$$\begin{array}{cccc} \mathsf{g}(0) = 0 & \mathsf{g}(1) = 3 & \mathsf{g}(2) = 6 & \mathsf{g}(3) = 9 \\ \mathsf{c}(0) = 0 & \mathsf{c}(1) = 1 & \mathsf{c}(2) = 2 & \mathsf{c}(3) = 1 \\ \mathsf{st}(1,0) = 41 & \mathsf{st}(2,0) = 41 & \mathsf{st}(2,1) = 1 & \mathsf{st}(3,0) = 42 \end{array}$$

and $\mathsf{hlt}(0) = \mathsf{hlt}(1) = \mathsf{hlt}(2) = \mathsf{hlt}(3) = \bot$.

Note that this program does not consume any words that were already on the stack. This is not the case in general. For instance we might be dealing with the body of a function, which takes its arguments from the stack. Hence we need to ensure that at the beginning of the execution sufficiently many words are on the stack. To this end we first compute the *depth* $\hat{\delta}(p)$ of the program p, *i.e.*, the number of words a program p consumes. Then we take variables $x_0, \ldots, x_{\hat{\delta}(p)-1}$ that represent the input to the program and initialize our functions accordingly.

Definition 2. *For a program with* $\hat{\delta}(p) = d$ *we initialize the state* σ *using*

$$\mathsf{g}_\sigma(0) = 0 \wedge \mathsf{hlt}_\sigma(0) = \bot \wedge \mathsf{c}_\sigma(0) = d \wedge \bigwedge_{0 \leqslant \ell < d} \mathsf{st}_\sigma(\mathcal{V}, 0, \ell) = x_\ell$$

For instance, for the program consisting of the single instruction `ADD` we set $\mathsf{c}(0) = 2$, and $\mathsf{st}(\{x_0, x_1\}, 0, 0) = x_0$ and $\mathsf{st}(\{x_0, x_1\}, 0, 1) = x_1$. We then have $\mathsf{st}(\{x_0, x_1\}, 1, 0) = x_1 + x_2$.

To encode the effect of EVM instructions we build SMT formulas to capture their operational semantics. That is, for an instruction ι and a state σ we give a formula $\tau(\iota, \sigma, j)$ that defines the effect on state σ if ι is the j-th instruction that is executed. Since large parts of these formulas are similar for every instruction and only depend on δ and α we build them from smaller building blocks.

Definition 3. *For an instruction* ι *and state* σ *we define:*

$$\begin{aligned} \tau_{\mathsf{g}}(\iota, \sigma, j) &\equiv \mathsf{g}_\sigma(\mathcal{V}, j+1) = \mathsf{g}_\sigma(\mathcal{V}, j) + C(\sigma, j, \iota) \\ \tau_{\mathsf{c}}(\iota, \sigma, j) &\equiv \mathsf{c}_\sigma(j+1) = \mathsf{c}_\sigma(j) + \alpha(\iota) - \delta(\iota) \\ \tau_{\mathsf{pres}}(\iota, \sigma, j) &\equiv \forall n.n < \mathsf{c}_\sigma(j) - \delta(\iota) \rightarrow \mathsf{st}_\sigma(\mathcal{V}, j+1, n) = \mathsf{st}_\sigma(\mathcal{V}, j, n) \\ \tau_{\mathsf{hlt}}(\iota, \sigma, j) &\equiv \mathsf{hlt}_\sigma(j+1) = \mathsf{hlt}_\sigma(j) \vee \mathsf{c}_\sigma(j) - \delta(\iota) < 0 \vee \mathsf{c}_\sigma(j) - \delta(\iota) + \alpha(\iota) > 2^{10} \end{aligned}$$

Here $C(\sigma, j, \iota)$ *is the gas cost of executing instruction* ι *on state* σ *after* j *steps.*

The formula τ_{g} adds the cost of ι to the gas cost incurred so far. The formula τ_{c} updates the counter for the number of words on the stack according to δ and α. The formula τ_{pres} expresses that all words on the stack below $\mathsf{c}_\sigma(j) - \delta(\iota)$ are preserved. Finally, τ_{hlt} captures that exceptions relevant to the stack can occur through either an underflow or an overflow, and that once it has occurred an exceptional halt state persists. For now the only other component we need is how the instructions affect the stack st, *i.e.*, a formula $\tau_{\mathsf{st}}(\iota, \sigma, j)$. Here we only give an example and refer to our implementation or the yellow paper [14] for details. We have

$$\begin{aligned}\tau_{\mathsf{st}}(\texttt{ADD}, \sigma, j) &\equiv \mathsf{st}_\sigma(\mathcal{V}, j+1, \mathsf{c}_\sigma(j+1) - 1) \\ &= \mathsf{st}_\sigma(\mathcal{V}, j, \mathsf{c}_\sigma(j) - 1) + \mathsf{st}_\sigma(\mathcal{V}, j, \mathsf{c}_\sigma(j) - 2)\end{aligned}$$

Finally these formulas yield an encoding for the semantics of an instruction.

Definition 4. *For an instruction ι and state σ we define*

$$\tau(\iota, \sigma, j) \equiv \tau_{\mathsf{st}}(\iota, \sigma, j) \wedge \tau_{\mathsf{c}}(\iota, \sigma, j) \wedge \tau_{\mathsf{g}}(\iota, \sigma, j) \wedge \tau_{\mathsf{hlt}}(\iota, \sigma, j) \wedge \tau_{\mathsf{pres}}(\iota, \sigma, j)$$

Then to encode the semantics of a program p all we need to do is to apply τ to the instructions of p.

Definition 5. *For a program $p = \iota_0 \cdots \iota_n$ we set $\tau(p, \sigma) \equiv \bigwedge_{0 \leqslant j \leqslant n} \tau(\iota_j, \sigma, j)$.*

Before building an encoding for superoptimization we consider another aspect of the EVM for our state representation: storage and memory. The gas cost for storing words depends on the words that are currently stored. Similarly, the cost for using memory depends on the number of bytes currently used. This is why the cost of an instruction $C(\sigma, j, \iota)$ depends on the state and the function g_σ accumulating gas cost depends on $\mathcal{V}$.

To add support for storage and memory to our encoding there are two natural choices: the theory of arrays or an Ackermann encoding. However, since we have not used arrays so far, they would require the solver to deal with an additional theory. For an Ackermann encoding we only need uninterpreted functions, which we have used already. Hence, to represent storage in our encoding we extend states with an uninterpreted function $\mathsf{str}(\mathcal{V}, j, k)$, which returns the word at key k after the program has executed j instructions. Similarly to how we set up the initial stack we need to deal with the values held by the storage before the program is executed. Thus, to initialize str we introduce fresh variables to represent the initial contents of the storage. More precisely, for all `SLOAD` and `SSTORE` instructions occurring at positions $j_1, \ldots, j_\ell$ in the source program, we introduce fresh variables $s_1, \ldots, s_\ell$ and add them to $\mathcal{V}$. Then for a state σ we initialize str_σ by adding the following conjunct to the initialization constraint from Definition 2:

$$\forall w.\ \mathsf{str}_\sigma(\mathcal{V}, 0, w) = \mathit{ite}(w = a_{j_1}, s_1, \mathit{ite}(w = a_{j_2}, s_2, \ldots, \mathit{ite}(w = a_{j_\ell}, s_\ell, w_\perp)))$$

where $a_j = \mathsf{st}_\sigma(\mathcal{V}, j, \mathsf{c}(j) - 1)$ and $w_\perp$ is the default value for words in the storage.

The effect of the two storage instructions SLOAD and SSTORE can then be encoded as follows:

$$\tau_{\mathsf{st}}(\texttt{SLOAD}, \sigma, j) \equiv \mathsf{st}_\sigma(\mathcal{V}, j+1, \mathsf{c}_\sigma(j+1)-1) = \mathsf{str}(\mathcal{V}, j, \mathsf{st}_\sigma(\mathcal{V}, j, \mathsf{c}_\sigma(j)-1))$$
$$\tau_{\mathsf{str}}(\texttt{SSTORE}, \sigma, j) \equiv \forall w.\ \mathsf{str}_\sigma(\mathcal{V}, j+1, w) = ite(w = \mathsf{st}_\sigma(\mathcal{V}, j, \mathsf{c}_\sigma(j)-1), \mathsf{st}_\sigma(\mathcal{V}, j, \mathsf{c}_\sigma(j)-2), \mathsf{str}_\sigma(\mathcal{V}, j, w))$$

Moreover all instructions except SSTORE preserve the storage, that is, for $\iota \neq$ SSTORE we add the following conjunct to $\tau_{\mathsf{pres}}(\iota, \sigma, j)$:

$$\forall w.\ \mathsf{str}_\sigma(\mathcal{V}, j+1, w) = \mathsf{str}_\sigma(\mathcal{V}, j, w)$$

To encode memory a similar strategy is an obvious way to go. However, we first want to evaluate the solver's performance on the encodings obtained when using stack and storage. Since the solver already struggled, due to the size of the programs and the number of universally quantified variables, see Section 6, we have not yet added an encoding of memory.

Finally, to use our encoding for superoptimization we need an encoding of equality for two states after a certain number of instructions. Either to ensure that two programs are equivalent (they start and end in equal states) or different (they start in equal states, but end in different ones). The following formula captures this constraint.

Definition 6. *For states σ_1 and σ_2 and program locations j_1 and j_2 we define*

$$\begin{aligned}\epsilon(\sigma_1, \sigma_2, j_1, j_2) \equiv\ & \mathsf{c}_{\sigma_1}(j_1) = \mathsf{c}_{\sigma_2}(j_2) \wedge \mathsf{hlt}_{\sigma_1}(j_1) = \mathsf{hlt}_{\sigma_2}(j_2) \\ & \wedge \forall n. n < \mathsf{c}_{\sigma_1}(j_1) \rightarrow \mathsf{st}_{\sigma_1}(\mathcal{V}, j_1, n) = \mathsf{st}_{\sigma_2}(\mathcal{V}, j_2, n) \\ & \wedge \forall w. \mathsf{str}_{\sigma_1}(\mathcal{V}, j_1, w) = \mathsf{str}_{\sigma_2}(\mathcal{V}, j_2, w)\end{aligned}$$

Since we aim to improve gas consumption, we do not demand equality for g.

We now have all ingredients needed to implement basic superoptimization: simply enumerate all possible programs ordered by gas cost and use the encodings to check equivalence. However, since already for one PUSH there are 2^{256} possible arguments, this will not produce results in a reasonable amount of time. Hence we use templates as described in Section 3. We introduce an uninterpreted function $\mathsf{a}(j)$ that maps a program location j to a word, which will be the argument of PUSH. The solver then fills these templates and we can get the values from the model. This is a step forward, but since we have 80 encoded instructions, enumerating all permutations still yields too large a search space. Hence we use an encoding similar to the CEGIS algorithm [4]. Given a collection of instructions, we formulate a constraint representing all possible permutations of these instructions. It is satisfiable if there is a way to connect the instructions into a target program that is equivalent to the source program. The order of the instructions can again be reconstructed from the model provided by the solver. More precisely given a source program p and a list of candidate instructions $\iota_1, \ldots, \iota_n$, EncodeBso from Algorithm 2(a) takes variables $j_1, \ldots, j_n$ and two states σ and σ' and builds

the following formula

$$\forall \mathcal{V}.\, \epsilon(\sigma, \sigma', 0, 0) \wedge \epsilon(\sigma, \sigma', |p|, n) \wedge \tau(p, \sigma) \\ \wedge \bigwedge_{1 \leqslant \ell \leqslant n} \tau(\iota_\ell, \sigma', j_\ell) \wedge \bigwedge_{1 \leqslant \ell < k \leqslant n} j_\ell \neq j_k \wedge \bigwedge_{1 \leqslant \ell \leqslant n} j_\ell \geqslant 0 \wedge j_\ell < n$$

Here the first line encodes the source program, and says that the start and final states of the two programs are equivalent. The second line encodes the effect of the candidate instructions and enforces that they are all used in some order. If this formula is satisfiable we can simply get the j_i from the model and reorder the candidate instructions accordingly to obtain the target program.

Unbounded superoptimization shifts even more of the search into the solver, encoding the search space of all possible programs. To this end we take a variable n, which represents the number of instructions in the target program and an uninterpreted function $\mathsf{instr}(j)$, which acts as a template, returning the instruction to be used at location j. Then, given a set of candidate instructions the formula to encode the search can be built as follows:

Definition 7. *Given a set of instructions* CI *we define the formula* $\rho(\sigma, n)$ *as*

$$\forall j.\, j \geqslant 0 \wedge j < n \to \bigwedge_{\iota \in \mathsf{CI}} \mathsf{instr}(j) = \iota \to \tau(\iota, \sigma, j) \wedge \bigvee_{\iota \in \mathsf{CI}} \mathsf{instr}(j) = \iota$$

Finally, the constraint produced by EncodeUso *from Algorithm 2(b) is*

$$\forall \mathcal{V}.\, \tau(p, \sigma) \wedge \rho(\sigma', n) \wedge \epsilon(\sigma, \sigma', 0, 0) \wedge \epsilon(\sigma, \sigma', |p|, n) \wedge \mathsf{g}_\sigma(\mathcal{V}, |p|) > \mathsf{g}_{\sigma'}(\mathcal{V}, n)$$

During our experiments we observed that the solver struggles to show that the formula is unsatisfiable when p is already optimal. To help in these cases we additionally add a bound on n: since the cheapest EVM instruction has gas cost 1, the target program cannot use more instructions than the gas cost of p, *i.e.*, we add $n \leqslant \mathsf{g}_\sigma(\mathcal{V}, |p|)$.

In our application domain there are many instructions that fetch information from the outside world. For instance, `ADDRESS` gets the Ethereum address of the account currently executing the bytecode of this smart contract. Since it is not possible to know these values at compile time we cannot encode their full semantics. However, we would still like to take advantage of structural optimizations where these instructions are involved, *e.g.*, via `DUP` and `SWAP`.

Example 3. Consider the program `ADDRESS DUP1`. The same effect can be achieved by simply calling `ADDRESS ADDRESS`. Duplicating words on the stack, if they are used multiple times, is an intuitive approach. However, because executing `ADDRESS` costs 2 g and `DUP1` costs 3 g, perhaps unexpectedly, the second program is cheaper.

To find such optimizations we need a way to encode `ADDRESS` and similar instructions. For our purposes, these instructions have in common that they put arbitrary but fixed words onto the stack. Analogous to uninterpreted functions, we call them *uninterpreted instructions* and collect them in the set UI. To represent

their output we use universally quantified variables—similar to input variables. To encode the effect uninterpreted instructions have on the stack, *i.e.*, τ_{st}, we distinguish between *constant* and *non-constant* uninterpreted instructions.

Let $\mathsf{ui}_c(p)$ be the set of *constant uninterpreted instructions* in p, *i.e.* $\mathsf{ui}_c(p) = \{\iota \in p \mid \iota \in \mathsf{UI} \wedge \delta(\iota) = 0\}$. Then for $\mathsf{ui}_c(p) = \{\iota_1, \ldots, \iota_k\}$ we take variables $u_{\iota_1}, \ldots, u_{\iota_k}$ and add them to $\mathcal{V}$, and thus to the arguments of the state function st. The formula τ_{st} can then use these variables to represent the unknown word produced by the uninterpreted instruction, *i.e.*, for $\iota \in \mathsf{ui}_c(p)$ with the corresponding variable u_ι in $\mathcal{V}$, we set $\tau_{\mathsf{st}}(\iota, \sigma, j) \equiv \mathsf{st}_\sigma(\mathcal{V}, j+1, \mathsf{c}_\sigma(j)) = u_\iota$.

For a *non-constant instruction* ι, such as `BLOCKHASH` or `BALANCE`, the word put onto the stack by ι depends on the top $\delta(\iota)$ words of the stack. We again model this dependency using an uninterpreted function. That is, for every non-constant uninterpreted instruction ι in the source program p, $\mathsf{ui}_n(p) = \{\iota \in p \mid \iota \in \mathsf{UI} \wedge \delta(\iota) > 0\}$, we use an uninterpreted function f_ι. Conceptually, we can think of f_ι as a read-only memory initialized with the values that the calls to ι produce.

Example 4. The instruction `BLOCKHASH` gets the hash of a given block b. Thus optimizing the program `PUSH` b_1 `BLOCKHASH PUSH` b_2 `BLOCKHASH` depends on the values b_1 and b_2. If $b_1 = b_2$ then the cheaper program `PUSH` b_1 `BLOCKHASH DUP1` yields the same state as the original program.

To capture this behaviour, we need to associate the arguments b_1 and b_2 of `BLOCKHASH` with the two different results they may produce. As with constant uninterpreted instructions, to model arbitrary but fixed results, we add fresh variables to $\mathcal{V}$. However, to account for different results produced by ℓ invocations of ι in p we have to add ℓ variables. Let p be a program and $\iota \in \mathsf{ui}_n(p)$ a unary instruction which appears ℓ times at positions $j_1, \ldots, j_\ell$ in p. For variables $u_1, \ldots, u_\ell$, we initialize f_ι as follows:

$$\forall w.\, f_\iota(\mathcal{V}, w) = \mathit{ite}(w = a_{j_1}, u_1, \mathit{ite}(w = a_{j_2}, u_2, \ldots, \mathit{ite}(w = a_{j_\ell}, u_\ell, w_\perp)))$$

where a_j is the word on the stack after j instructions in p, that is $a_j = \mathsf{st}_\sigma(\mathcal{V}, j, \mathsf{c}(j) - 1)$, and $w_\perp$ is a default word.

This approach straightforwardly extends to instructions with more than one argument. Here we assume that uninterpreted instructions put exactly one word onto the stack, *i.e.*, $\alpha(\iota) = 1$ for all $\iota \in \mathsf{UI}$. This assumption is easily verified for the EVM: the only instructions with $\alpha(\iota) > 1$ are `DUP` and `SWAP`. Finally we set the effect a non-constant uninterpreted instruction ι with associated function f_ι has on the stack:

$$\tau_{\mathsf{st}}(\iota, \sigma, j) \equiv \mathsf{st}_\sigma(\mathcal{V}, j+1, \mathsf{c}_\sigma(j+1) - 1) = f_\iota(\mathcal{V}, \mathsf{st}_\sigma(\mathcal{V}, j, \mathsf{c}_\sigma(j) - 1))$$

For some uninterpreted instructions there might a be way to partially encode their semantics. The instruction `BLOCKHASH` returns 0 if it is called for a block number greater than the current block number. While the current block number is not known at compile time, the instruction `NUMBER` does return it. Encoding this interplay between `BLOCKHASH` and `NUMBER` could potentially be exploited for finding optimizations.

5 Implementation

We implemented basic and unbounded superoptimization in our tool ebso, which is available under the Apache-2.0 license: github.com/juliannagele/ebso. The encoding employed by ebso uses several background theories: *(i)* uninterpreted functions (UF) for encoding the state of the EVM, for templates, and for encoding uninterpreted instructions, *(ii)* bit vector arithmetic (BV) for operations on words, *(iii)* quantifiers for initial words on the stack and in the storage, and the results of uninterpreted instructions, and *(iv)* linear integer arithmetic (LIA) for the instruction counter. Hence following the SMT-LIB classification[6] ebso's constraints fall under the logic UFBVLIA. As SMT solver we chose Z3 [3], version 4.7.1 which we call with default configurations. In particular, Z3 performed well for the theory of quantified bit vectors and uninterpreted functions in the last SMT competition (albeit non-competing).[7]

The aim of our implementation is to provide a prototype without relying on heavy engineering and optimizations such as exploiting parallelism or tweaking Z3 strategies. But without any optimization, for the full word size of the EVM—256 bit—ebso did not handle the simple program `PUSH 0 ADD POP` within a reasonable amount of time. Thus we need techniques to make ebso viable. By investigating the models generated by Z3 run with the default configuration, we believe that the problem lies with the leading universally quantified variables. And we have plenty of them: for the input on the stack, for the storage, and for uninterpreted instructions. By reducing the word size to a small k, we can reduce the search space for universally quantified variables from 2^{256} to some significantly smaller 2^k. But then we need to check any target program found with a smaller word size.

Example 5. The program `PUSH 0 SUB PUSH 3 ADD` from Example 1 optimizes to `NOT` for word size 2 bit, because then the binary representation of 3 is all ones. When using word size 256 bit this optimization is not correct.

To ensure that the target program has the same semantics for word size 256 bit, we use *translation validation*: we ask the solver to find inputs, which distinguish the source and target programs, *i.e.*, where both programs start in equivalent states, but their final state is different. Using our existing machinery this formula is easy to build:[8]

Definition 8. *Two programs p and p' are equivalent if*

$$\nu(p, p', \sigma, \sigma') \equiv \exists \mathcal{V}, \tau(p, \sigma) \wedge \tau(p', \sigma') \wedge \epsilon(\sigma, \sigma', 0, 0) \wedge \neg\epsilon(\sigma, \sigma', |p|, |p'|)$$

is unsatisfiable. Otherwise, p and p' are different, and the values for the variables in $\mathcal{V}$ from the model are a corresponding witness.

[6] smtlib.cs.uiowa.edu/logics.shtml

[7] smt-comp.github.io/2019/results/ufbv-single-query

[8] This approach also allows for other over-approximations. For instance, we tried using integers instead of bit vectors, which performed worse.

A subtle problem remains: how can we represent the program `PUSH` 224981 with only k bit? Our solution is to replace arguments $a_1, \ldots, a_m$ of `PUSH` where $a_i \geqslant 2^k$ with fresh, universally quantified variables $c_1, \ldots, c_m$. If a target program is found, we replace c_i by the original value a_i, and check with translation validation whether this target program is correct. A drawback of this approach is that we might lose potential optimizations.

Example 6. The program `PUSH` 0b111...111 `AND` optimizes to the empty program. But, abstracting the argument of `PUSH` translates the program to `PUSH` c_i `AND`, which does not allow the same optimization.

Like many compiler optimizations, ebso optimizes basic blocks. Therefore we split EVM bytecode along instructions that change the control flow, *e.g.* `JUMPI`, or `SELFDESTRUCT`. Similarly we further split basic blocks into (ebso) blocks so that they contain only encoded instructions. Instructions, which are not encoded, or encodable, include instructions that write to memory, *e.g.* `MSTORE`, or the log instructions `LOG`.

Lemma 1. *If program p superoptimizes to program t then in any program we can replace p by t.*

Proof. We show the statement by induction on the program context (c_1, c_2) of the program $c_1 p c_2$. By assumption, the statement holds for the base case $([\,], [\,])$. For the step case $(\iota c_1, c_2)$, we observe that every instruction ι is deterministic, *i.e.* executing ι starting from a state σ leads to a deterministic state σ'. By induction hypothesis, executing $c_1 p c_2$ and $c_1 t c_2$ from a state σ' leads to the same state σ'', and therefore we can replace $\iota c_1 p c_2$ by $\iota c_1 t c_2$. We can reason analogously for $(c_1, c_2 \iota)$.

6 Evaluation

We evaluated ebso on two real-word data sets: *(i)* optimizing an already highly optimized data set in Section 6.1, and *(ii)* a large-scale data set from the Ethereum blockchain to compare basic and unboundend superoptimization in Section 6.2. We use ebso to extract ebso blocks from our data sets. From the extracted blocks *(i)* we remove duplicate blocks, and *(ii)* we remove blocks which are only different in the arguments of `PUSH` by abstracting to word size 4 bit. We run both evaluations on a cluster [7] consisting of nodes running Intel Xeon E5645 processors at 2.40 GHz, with one core and 1 GiB of memory per instance.

We successfully validated all optimizations found by ebso by running a reference implementation of the EVM on pseudo-random input. Therefore, we run the bytecode of the original input block and the optimized bytecode to observe that both produce the same final state. The EVM implementation we use is `go-ethereum`[9] version `1.8.23`.

[9] github.com/ethereum/go-ethereum

	#	%
optimized (optimal)	19 (10)	0.69 % (0.36 %)
proved optimal	481	17.54 %
time-out (trans. val. failed)	2243 (196)	81.77 % (7.15 %)

Table 1: Aggregated results of running ebso on GG.

6.1 Optimize the Optimized

This evaluation tests ebso against human intelligence. Underlying our data set are 200 Solidity contracts (GG_{raw}) we collected from the *1st Gas Golfing Contest*.[10] In that contest competitors had to write the most gas-efficient Solidity code for five given challenges: *(i)* integer sorting, *(ii)* implementing an interpreter, *(iii)* hex decoding, *(iv)* string searching, and *(v)* removing duplicate elements. Every challenge had two categories: *standard* and *wild*. For wild, any Solidity feature is allowed—even inlining EVM bytecode. The winner of each track received 1 Ether. The Gas Golfing Contest provides a very high-quality data set: the EVM bytecode was not only optimized by the solc compiler, but also by humans leveraging these compiler optimizations and writing inline code themselves. To collect our data set GG, we first compiled the Solidity contracts in GG_{raw} with the same set-up as in the contest.[11] One contract in the wild category failed to compile and was thus excluded from GG_{raw}. From the generated `.bin-runtime` files, we extracted our final data set GG of 2743 distinct blocks.

For this evaluation, we run ebso in its default mode: unbounded superoptimization. We run unbounded superoptimization because, as can be seen in Section 6.2, in our context unbounded superoptimization outperformed basic superoptimization. As time-out for this challenging data set, we estimated 1 h as reasonable.

Table 1 shows the aggregated results of running ebso on GG. In total, ebso optimizes 19 blocks out of 2743, 10 of which are shown to be optimal. Moreover, ebso can prove for more than 17 % of blocks in GG that they are already optimal. It is encouraging that ebso even finds optimizations in this already highly optimized data set. The quality of the data set is supported by the high percentage of blocks being proved as optimal by ebso. Next we examine three found optimizations more closely. Our favorite optimization `POP PUSH 1 SWAP1 POP PUSH 0` to `SLT DUP1 EQ PUSH 0` witnesses that superoptimization can find unexpected results, and that unbounded superoptimization can stop with non-optimal results: `SLT DUP1 EQ` is, in fact, a round-about and optimizable way to pop two words from the stack and push 1 on the stack. Some optimizations follow clear patterns. The optimizations `CALLVALUE DUP1 ISZERO PUSH 81` to `CALLVALUE CALLVALUE`

[10] g.solidity.cc

[11] Namely, `$ solc --optimize --bin-runtime --optimize-runs 200` with solc compiler version `0.4.24` available at github.com/ethereum/solidity/tree/v0.4.24.

	uso		bso	
	#	%	#	%
optimized (optimal)	943 (393)	1.54 % (0.64 %)	184	0.3 %
proved optimal	3882	6.34 %	348	0.57 %
time-out (trans. val. failed)	56 392 (1467)	92.12 % (2.4 %)	60 685	99.13 %

Table 2: Aggregated results of running ebso with uso and bso on EthBC.

`ISZERO PUSH` 81 and `CALLVALUE DUP1 ISZERO PUSH` 364 to `CALLVALUE CALLVALUE ISZERO PUSH` 364 are both based on the fact that `CALLVALUE` is cheaper than `DUP1`. Finding such patterns and generalizing them into peephole optimization rules could be interesting future work.

Unfortunately, ebso hit a time-out in nearly 82 % of all cases, where we count a failed translation validation as part of the time-outs, since in that case ebso continues to search for optimizations after increasing the word size.

6.2 Unbounded *vs.* Basic Superoptimization

In this evaluation we compare unbounded and basic superoptimization, which we will abbreviate with uso and bso, respectively. To compare uso and bso, we want a considerably larger data set. Fortunately, there is a rich source of EVM bytecode accessible: contracts deployed on the Ethereum blockchain. Assuming that contracts that are called more often are well constructed, we queried the 2500 most called contracts[12] using Google BigQuery.[13] From them we extract our data set EthBC of 61 217 distinct blocks. For this considerably larger data set, we estimated a cut-off point of 15 min as reasonable. One limitation is that, due to the high volume, we only run the full evaluation once.

Table 2 shows the aggregated results of running ebso on EthBC. Out of 61 217 blocks in EthBC, ebso finds 943 optimizations using uso out of which it proves 393 to be optimal. Using bso 184 optimizations are found. Some blocks were shown to be optimal by both approaches. Also, both approaches time out in a majority of the cases: uso in more than 92 %, and bso in more than 99 %. Over all 61 217 blocks the total amount of gas saved for uso is 17 871 and 6903 for bso. For all blocks where an optimization is found, the average gas saving per block in uso is 29.63 %, and 46.1 % for bso. The higher average for bso can be explained by *(i)* bso's bias for smaller blocks, where relative savings are naturally higher, and *(ii)* bso only providing optimal results, whereas uso may find intermediate, non-optimal results. The optimization with the largest gain, is one which we did not necessarily expect to find in a deployed contract: a redundant storage access. Storage is expensive, hence optimized for in deployed contracts, but uso and

[12] up to block number 7 300 000 deployed on Mar-04-2019 01:22:15 AM +UTC

[13] cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics

bso both found PUSH 0 PUSH 4 SLOAD SUB PUSH 4 DUP2 SWAP1 SSTORE POP which optimizes to the empty program—because the program basically loads the value from key 4 only to store it back to that same key. This optimization saves at least 5220 g, but up to 20 220 g.

From Table 2 we see that on EthBC, uso outperforms bso by roughly a factor of five on found optimizations; more than ten times as many blocks are proved optimal by uso than by bso. As we expected, most optimizations found by bso were also found by uso, but surprisingly, bso found 21 optimizations, on which uso failed. We found that nearly all of the 21 source programs are fairly complicated, but have a short optimization of two or three instructions. To pick an example, the block PUSH 0 PUSH 12 SLOAD LT ISZERO ISZERO ISZERO PUSH 12250 is optimized to the relatively simple PUSH 1 PUSH 12250—a candidate block, which will be tried early on in bso. Additionally, all 21 blocks are cheap: all cost less than 10 g. We also would have expected at least some of these optimizations to have been found by uso. We believe internal unfortunate, non-deterministic choice within the solver to be the reason that it did not.

7 Conclusion

Summary. We develop ebso, a superoptimizer for EVM bytecode, implementing two different superoptimization approaches and compare them on a large set of real-world smart contracts. Our experiments show that, relying on the heavily optimized search heuristics of a modern SMT solver is a feasible approach to superoptimizing EVM bytecode.

Related Work. Superoptimization [9] has been explored for a variety of different contexts [5, 6, 10, 12], including binary translation [1] and synthesizing compiler optimizations [11]. To our knowledge ebso is the first application of superoptimization to smart contracts.

Chen et al. [2] also aim to save gas by optimizing EVM bytecode. They identified 24 anti patterns by manual inspection. Building on their work we run ebso on their identified patterns. For 19 instances, ebso too found the same optimizations. For 2 patterns, ebso lacks encoding of the instructions (STOP, JUMP), and for 2 patterns ebso times out on a local machine.

Due to the repeated exploitation of flaws in smart contracts, various formal approaches for analyzing EVM bytecode have been proposed. For instance Oyente [8] performs control flow analysis in order to detect security defects such as reentrancy bugs.

Outlook. There is ample opportunity for future work. We do not yet support the EVM's memory. While conceptually this would be a straightforward extension, the number of universally quantified variables and size of blocks are already posing challenges for performance, as we identified by analyzing the optimizations found by ebso.

Thus, it would be interesting to use SMT benchmarks obtained by ebso's superoptimization encoding to evaluate different solvers, *e.g.* CVC4[14] or Vampire[15]. The basis for this is already in place: ebso can export the generated constraints in SMT-LIB format. Accordingly, we plan to generate new SMT benchmarks and submit them to one of the suitable categories of SMT-LIB.

In order to ease the burden on developers ebso could benefit from caching common optimization patterns [11] to speed up optimization times. Another fruitful approach could be to extract the optimization patterns and generalize them into peephole optimizations and rewrite rules.

References

1. Bansal, S., Aiken, A.: Binary translation using peephole superoptimizers. In: Proc. 8th OSDI. pp. 177–192. USENIX (2008)
2. Chen, T., Li, Z., Zhou, H., Chen, J., Luo, X., Li, X., Zhang, X.: Towards saving money in using smart contracts. In: Proc. 40th ICSE-NIER. pp. 81–84. ACM (2018). https://doi.org/10.1145/3183399.3183420
3. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proc. 14th TACAS. LNCS, vol. 9206, pp. 337–340. Springer (2008)
4. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: Proc. 32nd PLDI. pp. 62–73. ACM (2011). https://doi.org/10.1145/1993498.1993506
5. Jangda, A., Yorsh, G.: Unbounded superoptimization. In: Proc. Onward! 2017. pp. 78–88. ACM (2017). https://doi.org/10.1145/3133850.3133856
6. Joshi, R., Nelson, G., Randall, K.H.: Denali: A Goal-directed Superoptimizer. In: Proc. 23rd PLDI. pp. 304–314. ACM (2002). https://doi.org/10.1145/512529.512566
7. King, T., Butcher, S., Zalewski, L.: Apocrita - High Performance Computing Cluster for Queen Mary University of London (Mar 2017). https://doi.org/10.5281/zenodo.438045
8. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proc. 23rd CCS. pp. 254–269. ACM (2016). https://doi.org/10.1145/2976749.2978309
9. Massalin, H.: Superoptimizer: A look at the smallest program. In: Proc. 2nd ASPLOS. pp. 122–126. IEEE (1987). https://doi.org/10.1145/36206.36194
10. Phothilimthana, P.M., Thakur, A., Bodík, R., Dhurjati, D.: Scaling up superoptimization. In: Proc. 21st ASPLOS. pp. 297–310. ACM (2016). https://doi.org/10.1145/2872362.2872387
11. Sasnauskas, R., Chen, Y., Collingbourne, P., Ketema, J., Taneja, J., Regehr, J.: Souper: A synthesizing superoptimizer. CoRR **abs/1711.04422** (2017), http://arxiv.org/abs/1711.04422
12. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. In: Proc. 18th ASPLOS. pp. 305–316. ACM (2013). https://doi.org/10.1145/2451116.2451150
13. Srinivasan, V., Reps, T.: Synthesis of machine code from semantics. In: Proc. 36th PLDI. pp. 596–607. ACM (2015). https://doi.org/10.1145/2737924.2737960
14. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Tech. Rep. Byzantium Version e94ebda (2018), https://ethereum.github.io/yellowpaper/paper.pdf

[14] cvc4.cs.stanford.edu/web/

[15] www.vprover.org

An Integrated Approach to Assertion-Based Random Testing in Prolog*

Ignacio Casso[1], José F. Morales[1],
P. López-García[1,3], and Manuel V. Hermenegildo[1,2]

[1] IMDEA Software Institute, Madrid, Spain
[2] ETSI Informática, Universidad Politécnica de Madrid (UPM), Madrid, Spain
[3] Spanish Council for Scientific Research (CSIC), Spain
{ignacio.casso,josef.morales,pedro.lopez,manuel.hermenegildo}@imdea.org

Abstract. We present an approach for assertion-based random testing of Prolog programs that is integrated within an overall assertion-based development model. Our starting point is the `Ciao` assertion model, a framework that unifies unit testing and run-time verification, as well as static verification and static debugging, using a common assertion language. Properties which cannot be verified statically are checked dynamically. In this context, the idea of generating random test values from assertion preconditions emerges naturally since these preconditions are conjunctions of literals, and the corresponding predicates can in principle be used as generators. Our tool generates valid inputs from the properties that appear in the assertions shared with other parts of the model, and the run time-check instrumentation of the `Ciao` framework is used to perform a wide variety of checks. This integration also facilitates the combination with static analysis. The generation process is based on running standard predicates under non-standard (random) search rules. Generation can be fully automatic but can also be guided or defined specifically by the user. We propose methods for supporting (C)LP-specific properties, including combinations of shape-based (regular) types and variable sharing and instantiation, and we also provide some ideas for shrinking for these properties. We also provide a case study applying the tool to the verification and checking of the implementations of some of `Ciao`'s abstract domains.

1 Introduction and motivation

Code validation is a vital task in any software development cycle. Traditionally, two of the main approaches used to this end are *verification* and *testing*. The former uses formal methods to prove automatically or interactively some specification of the code, while the latter mainly consists in executing the code for concrete inputs or test cases and checking that the program input-output relations (and behaviour, in general) are the expected ones.

`Ciao` [11] introduced a novel development workflow [12, 13, 21] that integrates the two approaches above. In this model, program assertions are fully integrated

* Research partially funded by MINECO TIN2015-67522-C3-1-R *TRACES* project, and the Madrid P2018/TCS-4339 *BLOQUES-CM* program. We are also grateful to the anonymous reviewers for their useful comments.

in the language, and serve both as specifications for static analysis and as run-time check generators, unifying run-time verification and unit testing with static verification and static debugging. This model represents an alternative approach for writing safe programs without relying on full static typing, which is specially useful for dynamic languages like Prolog, and can be considered an antecedent of the popular *gradual-* and *hybrid-typing* approaches [5, 24, 22].

The `Ciao` model: Assertions in the `Ciao` model can be seen as a shorthand for defining instrumentation to be added to programs, in order to check dynamically preconditions and postconditions, including conditional postconditions, properties at arbitrary program points, and certain computational (non-functional) properties. The run-time semantics implemented by the translation of the assertion language ensures that execution paths that violate the assertions are captured during execution, thus detecting errors. Optionally, (abstract interpretation-based [4]) compile-time analysis is used to detect assertion violations, or to prove (parts of) assertions true, verifying the program or reducing run-time checking overhead.

As an example, consider the following `Ciao` code (with the standard definition of quick-sort):

```
:- pred qs(Xs,Ys) : (list(Xs), var(Ys)) => (list(Ys), sorted(Ys)) + not_fails.

:- prop list/1.
list([]).
list([_|T]) :- list(T).

:- prop sorted/1.
...
```

The assertion has a *calls* field (the conjunction after '`:`'), a *success* field (the conjunction after '`=>`'), and a computational properties field (after '`+`'), where all these fields are optional. It states that a valid calling mode for `qs/2` is to invoke it with its first argument instantiated to a list, and that it will then return a list in `Ys`, that this list will be sorted, and that the predicate will not fail. Properties such as `list/1` or `sorted/1` are normal predicates, but which meet certain properties (e.g., termination) [13] and are marked as such via `prop/1` declarations. Other properties like `var/1` or `not_fails` are builtins.

Compile-time analysis with a types/*shapes* domain can easily detect that, if the predicate is called as stated in the assertion, the `list(Ys)` check on success will always succeed, and that the predicate itself will also succeed. If this predicate appears within a larger program, analysis can also typically infer whether or not `qs/2` is called with a list and a free variable. However, perhaps, e.g., `sorted(Ys)` cannot be checked statically (this is in fact often possible, but let us assume that, e.g., a suitable abstract domain is not at hand). The assertion would then be simplified to:

```
:- pred qs(Xs,Ys) => sorted(Ys).
```

And then `sorted(Ys)` will be called at run-time within the assertion checking-harness, right after calls to `qs/2`. This harness ensures that the variable bindings (or constraints) and the whole checking process are kept isolated from the normal execution of the program (this can be seen conceptually as including a Prolog `copy_term`, or calling within a double negation, `\+\+`, executing in a separate process, etc.).

Testing vs. run-time checking: The checking of `sorted/1` in the example above will occur in principle during execution of the program, i.e., in *deployment*. However, in many cases it is not desirable to wait until then to detect errors. This is the case for example if errors can be catastrophic or perhaps if there is interest in testing, perhaps for debugging purposes, more general properties that have not been formally proved and whose main statements are not directly part of the program (and thus, will never be executed), such as, e.g.:

```
:- pred revrev(X) : list(X) + not_fails.
revrev(X) :- reverse(X,Y),reverse(Y,X).
```

This implies performing a *testing* process prior to deployment. The `Ciao` model includes a mechanism, integrated with the assertion language, that allows defining *test assertions*, which will run (parts of) the program for a given input and check the corresponding output, as well as *driving* the run-time checks independently of concrete application data [16]. For example, if the following (unit) tests are added to `qs/2`:

```
:- test qs(Xs,Ys) : (Xs = []) => (Ys = []).
:- test qs(Xs,Ys) : (Xs = [3,2,4,1]) => (Ys = [1,2,3,4]).
```

`qs/2` will be *run* with, e.g., `[3,2,4,1]` as input in `Xs`, and the output generated in `Ys` will be checked to be instantiated to `[1,2,3,4]`. This is done by extracting the *test drivers* [16]:

```
:- texec qs([],_).
:- texec qs([3,2,4,1],_).
```

and the rest of the work (checking the assertion fields) is done by the standard run-time checking machinery. In our case, this includes checking at run-time the simplified assertion "`:- pred qs(Xs,Ys) => sorted(Ys).`", so that the output in `Ys` will be checked by calling the implementation of `sorted/1`.

Towards automatic generation: Hand-written test cases such as those above are quite useful in practice, but they are also tedious to write and even when they are present they may not cover some interesting cases. An aspect that is specific to (Constraint-)Logic Programming (CLP) and is quite relevant in this context is that predicates in general (and properties in particular) can be used as both checkers and generators. For example, calling `list(X)` from the `revrev/1` example above with `X` uninstantiated generates lazily, through backtracking, an infinite set of lists, `Xs = []; Xs = [_]; Xs = [_,_,_]` ..., which can be used to catch cases in which an error in the coding of `reverse/2` makes `revrev/1` fail. This leads naturally to the idea of generating systematically and automatically test cases by running in generation mode (i.e., "backwards") the properties in the calls fields of assertions.

While this idea of using properties as test case generators has always been present in the descriptions of the `Ciao` model [12, 21], it has not really been exploited significantly to date. Our purpose in this paper is to close this gap. We report on the development of `LPtest`, an implementation of random testing [8] with a more natural connection with Prolog semantics, as well as with the `Ciao` framework. Due to this connection and the use of assertions, this *assertion-based testing* allows supporting complex properties like combinations of shape-based (regular) types, variable sharing, and instantiation, and also non-functional properties.

Our contributions can be summarized as follows:

- We have developed an approach and a tool for assertion-based random test generation for Prolog and related languages. It has a number of characteristics in common with property-based testing from functional languages, as exemplified by `QuickCheck` [3], but provides the assertions and properties required in order to cover (C)LP features such as logical variables and non-ground data structures or non-determinism, with related properties such as modes, variables sharing, non-failure, determinacy and number of solutions, etc. In this, `LPtest` is most similar to `PrologTest` [1], but we argue that our framework is more general and we support richer properties.
- Our approach offers a number of advantages that stem directly from framing it within the `Ciao` model. This includes the integration with compile-time checking (static analysis) and the combination with the run-time checking framework, etc. using a single assertion language. This for example greatly simplifies error reporting and diagnosis, which can all be inherited from these parts of the framework. It can also be combined with other test-case generation schemes. To the extent of our knowledge, this has only been attempted partially by `PropEr` [20]. Also, since `Erlang` is in many ways closer to a functional language, `PropEr` does not support Prolog-relevant properties and it is not integrated with static analysis. In comparison to `PrologTest`, we provide combination with static analysis, through an integrated assertion language, whereas the assertions of `PrologTest` are specific to the tool, and we also support a larger set of properties.
- In our approach the automatic generation of inputs is performed by running in generation mode the properties (predicates) in those preconditions, taking advantage of the specialized SLD *search rules* of the language (e.g., breadth first, iterative deepening, and, in particular, random search) or implementations specialized for such generation. In particular, we perform automatic generation of instances of Prolog regular types, instantiation modes, sharing relations among variables and grounding, arithmetic constraints, etc. To the extent of our knowledge all previous tools only supported generation for types, while we also consider the latter.
- We have enhanced assertion and property-based test generation by combining it with static analysis and abstract domains. To the extent of our knowledge previous work had at most discarded properties that could be proved statically (which in `LPtest` comes free from the overall setting, as mentioned before), but not used static analysis information to guide or improve the testing process.
- We have implemented automatic shrinking for our tool, and in particular we have developed an automatic shrinking algorithm for Prolog regular types.

The rest of this paper is organized as follows. In Sec. 2 we overview our approach to assertion-based testing in the context of Prolog and `Ciao`. In Sec. 3 we introduce our test input generation schema. In Sec. 4, we show how assertion-based testing can be combined with and enhanced by static analysis. Sec. 5 is dedicated to shrinking of test cases in `LPtest`. We show some preliminary results of a case of study in which our tool is applied to prove the correctness of `CiaoPP` (the `Ciao` "preprocessor") static analysis domain operations in Sec. 6. Finally, we review the related work in Sec. 7 and provide our conclusions in Sec. 8.

2 Using LPtest within the Ciao model

As mentioned before, the goal of LPtest is to integrate random testing of assertions within Ciao's assertion-based verification and debugging framework. Given an assertion for a predicate, we want to generate goals for that predicate satisfying the assertion precondition (i.e., valid call patterns for the predicate) and execute them to check that the assertion holds for those cases or find errors. As mentioned briefly in the introduction, the current framework already provides most of the pieces needed for this task: the run-time checking framework allows us to check at runtime that the assertions for a predicate are not violated, and the unit-test framework allows us to specify and run the concrete goals to check those assertions. We only need to be able to generate terms satisfying assertion preconditions and to integrate everything. Generation of test cases is discussed in Sec. 3, and the following example shows how everything is integrated step by step.

Consider again a similar assertion for the qs/2 predicate, and assume that the program has a bug and *fails* for lists with repeated elements:

```
:- module(qs,[qs/2],[assertions, nativeprops]).
...
:- pred qs(Xs,Ys) : (list(Xs,int), var(Ys))
                 => (list(Ys,int), sorted(Ys)) + not_fails.
...
partition([],_,[],[]).
partition([X|Xs],Pv,[X|L],R) :- X < Pv, !, partition(Xs,Pv,L,R). % should be =<
partition([X|Xs],Pv,L,[X|R]) :- X > Pv, partition(Xs,Pv,L,R).
```

When we invoke LPtest to test the assertions of the qs module, first it will use CiaoPP to analyze the module and try to verify the assertions statically [13]. As a result each assertion may be proved true or false, and then it can be ignored in the testing phase, or it is left for run-time checking and testing, possibly simplified. CiaoPP generates a new source file which includes the original assertions marked with *status* checked, false, or, for the ones that remain for run-time checking, check. LPtest starts by reporting a simple adaptation of CiaoPP's output. E.g., for our example, LPtest will output:

```
Testing assertion:
:- pred qs(Xs,Ys) : (list(Xs,int), var(Ys))
                 => (list(Ys,int), sorted(Ys)) +  not_fails.

Assertion was partially verified statically:
  :- checked pred qs(Xs,Ys) (list(Xs,int), var(Ys)) => list(Ys,int).
Left to check::
  :- check   pred qs(Xs,Ys) => sorted(Ys) + not_fails.
```

LPtest will then try test dynamically the remaining assertion. For that, it will first collect the Ciao properties that the test case must fulfill (i.e., those in the precondition of the assertion, which is taken from the *original* assertion, which is also output by CiaoPP), and generate a number (100 by default) of *test case drivers* (texec's) satisfying those properties. Those test cases will be pipelined to the *unit-test framework*, which, relying on the standard run-time checking instrumentation, will manage their execution, capture any error reported during run-time checking, and return them to LPtest, which will output:

```
Assertion
   :- check   pred qs(Xs,Ys) => sorted(Ys) + not_fails.
proven false for test case:
   :- texec qs([5,9,-3,8,9,-6,2],_).
```

```
because:
   call to qs(Xs,Ys) fails for
   Xs = [9,-3,8,9,-6,2]
```

Finally, `LPtest` will try to shrink the test cases, enumerating test cases that are progressively smaller and repeating the steps above in a loop to find the smallest test case which violates the assertion. `LPtest` will output:

```
Test case shrinked to:
   :- texec qs([0,0],_).
```

The testing algorithm for a module can thus be summarized as follows:

1. (Call `CiaoPP`) Use *static analysis* to check the assertions. Remove proved assertions, simplify partially proved assertions.
2. (`LPtest`) For each assertion, *generate N test cases* from the properties in the precondition, following the guidelines in Sec. 3. For each test case, go to **3**. Then go to **4**.
3. (`RTchecks`) Use the unit-test framework to execute the test case and capture any run-time checking error (i.e., assertion violation).
4. (`LPtest`) Collect all failed test cases from `RTchecks`. For each of them, go to **5** to shrink them, and then report them (using `RTchecks`).
5. (`LPtest`) Generate a simpler test case not generated yet.
   - If not possible, finalize and return current test case as shrinked test case. If possible, go to **3** to run the test.
     * If the new test case fails, go to **5** with the new test case.
     * If it succeeds, repeat this step.

The use of the `Ciao` run-time checking framework in this (pseudo-)algorithm, together with the rich set of native properties in `Ciao`, allows us to specify and check a wide range of properties for our programs. We provide a few examples of the expressive power of the approach:

(Conditional) Postconditions. We can write postconditions using the *success* (`=>`) field of the assertions. Those postconditions can range from user-defined predicates to properties native to `CiaoPP`, for which there are built-in checkers in the run-time checking framework. These properties include types, which can be partially instantiated, i.e., contain variables, and additional features particular to logic programming such as modes and sharing between variables. As an example, one can test with `LPtest` the following assertions, where `covered(X,Y)` means that all variables occurring in `X` also occur in `Y`:

```
:- pred rev(Xs,Ys) : list(Xs) => list(Ys).
:- pred sort(Xs,Ys) : list(Xs,int) => (list(Ys,int), sorted(Ys)).
:- pred numbervars(Term,N,M) => ground(Term).
:- pred varset(Term,Xs) => (list(Xs,var), covered(Term,Xs)).
```

For this kind of properties, `LPtest` tries to ensure that at least some of the test cases do not succeed trivially (by the predicate just failing), and warns otherwise.

Computational Properties. `LPtest` can also be used to check properties regarding the computation of a predicate. These properties are all native and talk about features that range from determinism and multiplicity of solutions to resource usage (cost). They can be checked with `LPtest`, as long as the run-time checking

framework supports it (e.g., some properties, like termination, are not decidable). Examples of this would be:

```
:- pred rev(X,Y) : list(X) + (not_fails, is_det, no_choicepoints).
:- pred append(X,Y,Z) : list(X) => steps_o(length(X)).
```

Rich generation. The properties supported for generation include not only types, but also modes and sharing between variables, and arithmetic constraints, as well as a restricted set of user-defined properties. As an example, `LPtest` can test the following assertion:

```
:- prop sorted_int_list(X).

sorted_int_list([]).
sorted_int_list([N]) :- int(N).
sorted_int_list([N,M|Ms]) :- N =< M, sorted_int_list([M|Ms]).

:- pred insert_ord(X,Xs,XsWithX)
    : (int(X), sorted_int_list(X))
   => sorted_int_list(XsWithX).
```

3 Test Case Generation

The previous section illustrated specially the parts that `LPtest` inherits from the `Ciao` framework, but a crucial step was skipped: the generation of test cases from the *calls* field of the assertions, i.e., the generation of Prolog terms satisfying a conjunction of `Ciao` properties. This was obviously one of the main challenges we faced when designing and implementing `LPtest`. In order for the tool to be integrated naturally within the `Ciao` verification and debugging framework, this generation had to be as automatic as possible. However, full automation is not always possible in the presence of arbitrary properties potentially using the whole Prolog language (e.g., cuts, dynamic predicates, etc.). The solution we arrived at is to support fully automatic and efficient generation for reasonable subsets of the Prolog language, and provide means for the user to guide the generation in more complex scenarios.

Pure Prolog. The simplest and essential subset of Prolog is pure Prolog. In pure Prolog every predicate, and, in particular, every `Ciao` property, is itself a generator: if it succeeds with some terms as arguments, those terms will be (possibly instances of) answers to the predicate when called with free variables as arguments. The problem is that the classic depth-first search strategy used in Prolog resolution, with which those answers will be computed, is not well suited for test-case generation. One of `Ciao`'s features comes here to the rescue. `Ciao` has a concept of *packages*, syntactic and/or semantic extensions to the language that can be loaded module-locally. This mechanism is used to implement language extensions such as functional syntax, constraints, higher order, etc., and, in particular *alternative search rules*. These include for example (several versions of) breadth first, iterative deepening, Andorra-style execution, etc. These rules can be activates on a per-module basis. For example, the predicates in a module that starts with the following header:

```
:- module( myprops, _, [bf]).
```

(which loads the `bf` package) will run in breadth-first mode. While breadth-first is useful mostly for teaching, other alternative search rules are quite useful in practice. Motivated by the `LPtest` context, i.e., with the idea of running properties in

generation mode, we have developed also a *randomized alternative search strategy* package, rnd, which can be described by the following simplified meta-interpreter:

```
solve_goal(G) :- random_clause(G,Body), solve_body(Body).

random_clause(Head,Body) :-
    findall(cl(Head,B),meta_clause(Head,B),ClauseList),
    once(shuffle(ClauseList,ShuffleList)),
    member(cl(Head,Body),ShuffleList). % Body=[] for facts

solve_body([]).
solve_body([G|Gs]) :- solve_goal(G), solve_body(Gs).
```

The actual algorithm used for generation is of course more involved. Among other details, it only does backtracking on failure (on success it starts all over again to produce the next answer, without repeating traces), and it has a growth control mechanism to avoid getting stuck in traces that lead to non-terminating generations.

Using this search strategy, a set of terms satisfying a conjunction of pure Prolog properties can be generated just by running all those properties sequentially with unbounded variables. This is implemented using different versions of each property (generation, run-time check) which are generated automatically from the declarative definition of the property using instrumentation. In particular, this simplest subset of the language allows us to deal directly with regular types (e.g., `list/1`).

Mode, sharing, and arithmetic constraints. We extend the subset of the language for which generation is supported with arithmetic (e.g., `integer/1`, `float/1`, `</2`), mode-related extralogical predicates and properties (e.g., `var/1`, `ground/1`), and sharing-related native properties (e.g., `mshare/1`, which describes the sharing –aliasing– relations of a set of variables using *sharing sets* [15], and `indep/2`, that states that two variable do not share). When a goal or a property of this kind appears during generation, the variables occurring in it are constrained using a constraints domain. The domain ensures that those constraints are satisfiable during all steps of generation, failing and backtracking otherwise. There is a last step in generation in which all free variables are randomly further intantiated in a way that those constraints are satisfied.

This can be seen conceptually as choosing first a trace at random for each property and collecting constraints in the trace, and then randomly sampling (enumerating) the constrains. However, since the constraints introduced by unification are terms, it is equivalent to solving a predicate with the random search strategy and treating each builtin or native property as a constraint. In practice, we support more builtins for generation in properties (e.g., `==/2` just unifies two variables, we have shape constraints that handle `=../2`, and support negation to some extent), but the approach has only been tested significantly for the subset of Prolog presented so far.

In the last phase of constraints (random sampling), unconstrained free variables can be further instantiated with some probability, using random shape and sharing constraints, chosen among native properties and properties defined by the users on modules that are loaded at the time. This way, random terms are still generated for an assertion without precondition, or the generated term for `list(X)` is not always a list of free variables. This is also the technique used to further instantiate a free variable constrained as *ground* but for which no shape information is available.

Generation for other properties. For the remaining properties which use Prolog features not covered so far (e.g., dynamic predicates), there is a last step in the

generation algorithm in which they are simply checked for the terms generated so far. User-defined generators are encouraged for assertions with preconditions that are complex enough to reach this step. There is a limit to how many times generation can reach this step and fail, to avoid getting stuck in an inefficient or non-terminating generate-and-check loop. To recognize these properties without inspecting the code (left as future work), users are trusted to mark the properties suitable for generation with special syntax, and only the native properties discussed and the regular types are considered suitable by default.

4 Integration with static analysis

The use of a unified assertion framework for testing and analysis allows us to enhance `LPtest` random testing by combining it with static analysis.

First of all, as illustrated in Section 2, by performing static analysis first, some of the assertions can be (partially) verified statically, so they can be simplified or ignored for the testing phase. This process is performed automatically without any effort on `LPtest`'s part: `CiaoPP` is already capable of doing static assertion checking and simplification [13], and generates a new source file with the assertions left to be checked, which are the ones we test with our tool.

Beyond this, and perhaps more interestingly in our context, statically inferred information can also help while testing the remaining assertions. In particular, it is used to generate more relevant test cases in the generation phase. Consider for example the following assertion:

```
:- pred qs(Xs,Ys) => sorted(Ys).
```

Without the usual precondition, `LPtest` would have to generate arbitrary terms to test the assertion, most of which would not be relevant test cases since the predicate would fail for them, and therefore the assertion would be satisfied trivially. However, static analysis typically infers the output type for this predicate:

```
:- pred qs(Xs,_) => list(Xs,int).
```

I.e., analysis infers that on success `Xs` must be a list, and so on call it must be *compatible* with a list if it is to succeed. Therefore the assertion can also be checked as follows:

```
:- pred qs(Xs,_) : compat(Xs,list(int)) => sorted_int_list(Xs).
```

where `compat(Xs,list(int))` means that `Xs` is either a list of integers or can be further instantiated to one. Now we would only generate relevant inputs (generation for `compat/2` is implemented by randomly uninstantiating a term), and `LPtest` is able to prove the assertion false. The same can be done for modes and sharing to some extent: variables that are inferred to be free on success must also be free on call, and variabless inferred to be independent must be independent on call too. Also, when a predicate is not exported, the *calls* assertions inferred for it can be used for generation. In general, the idea here is to perform some backwards analysis. However, this can also be done without explicit backwards analysis by treating testing and (forward) static analysis independently and one after the other, which makes the integration conceptually simple and easy to implement.

A finer-grain integration. We now propose a finer-grained integration of assertion-based testing and analysis, which still treats analysis as a black box, although not as an independent step. So far our approach has been to try to check an assertion with static analysis, and if this fails we perform random testing. However, the analysis often fails to prove the assertion because its precondition (i.e., the entry abstract substitution to the analysis) is too general, but it can prove it for refinements of that entry, i.e., refinements of the precondition. In that case, all test cases satisfying that refined precondition are guaranteed to succeed, and therefore useless in practice. We propose to work with different refined versions of an assertion, by adding further, exhaustive constraints in a native domain to the precondition, and performing testing only on the versions which the analysis cannot verify statically, thus pruning the test case input space. For example, for an assertion of a predicate of arity one, without mode properties, three different assertions equivalent to the first would result by adding to the precondition (`ground(X)`, `var(X)`), or `(nonground(X), nonvar(X))`. The idea is to generalize this to arbitrary, maybe infinite abstract domains, for which an abstract value is not so easily partitioned as in the example above. It is still in development, but the core of the algorithm would be the following: to test an assertion to a given entry $A \in D_\alpha$, the assertion is proved by the analysis or tested recursively for a set of abstract values $S \subseteq \{B | B \in D_\alpha, B \sqsubseteq A\}$ lower than that entry, and random test cases are generated in the "space" between the entry and those lower values $\gamma(A) \setminus \bigcup \gamma(B)$, where γ is the concretization function in the domain. For this it is only necessary to provide a suitable sampling function in the domain, and a rich generation algorithm for that domain. But note that, e.g., for the *sharing-freeness* domain, we already have the latter: we already have generation for mode and sharing constraints, and a transformation scheme between abstract values and mode/sharing properties. Note also that all this can still be done while treating the static analysis as a black box, and that if the enumeration of values is fine-grained enough, this algorithm also ensures coverage of the test input space during generation.

5 Shrinking

One flaw of random testing is that often the failed test cases reported are unecessary complex, and thus not very useful for debugging. Many property-based tools introduce shrinking to solve this problem: after one counter-example is found, they try to reduce it to a simpler counter-example that still fails the test in the same way. `LPtest` supports shrinking too, both user-guided and automatic. We present the latter.

The shrinking algorithm mirrors that of generation, and in fact reuses most of the generation framework. It can be seen as a new generation with further constraints: bounds on the shape and size of the generated goal. The traces followed to generate the new term from a property must be "subtraces" of the ones followed to generate the original one. The random sampling of the constraints for the new terms must be "simpler" than for the original ones. The final step in which the remaining properties are checked is kept.

We present the algorithm for the first step. Generation for the shrinked value follows the path that leads to the to-be-shrinked value, but at any moment it can non-deterministically stop following that trace and generate a new subterm using size parameter 0. Applying this method to shrink lists of Peano numbers is equiva-

lent to the following predicate, where the first argument is the term to be shrinked and the second a free variable to be the shrinked value on success:

```
shrink_peano_list([X|Xs],[Y|Ys]) :-
  shrink_peano_number(X,Y),
  shrink_peano_list(Xs,Ys).
shrink_peano_list(_,Ys) :-
  gen_peano_list(0,Ys). % X=[]

shrink_peano_number(s(X),s(Y)) :-
  shrink_peano_number(X,Y).
shrink_peano_number(_,Y) :-
  gen_peano_number(0,Y). % Y=0.
```

This method can shrink the list `[s(0),0,s(s(s(0)))]` to `[s(0),0]` or `[s(0),0,s(s(0))]`, but never to `[s(0),s(s(s(0)))]`. To solve that, we allow the trace of the to-be-shrinked term to advance non-deterministically at any moment to an equivalent point, so that the trace of the generated term does not have to follow it completely in parallel. It would be as if the following clauses were added to the the previous predicate (the one which sketches the actual workings of the method during meta-interpretation):

```
shrink_peano_list([_|Xs],Ys) :-
  shrink_peano_list(Xs,Ys).

shrink_peano_number(s(X),Y) :-
  shrink_peano_number(X,Y).
```

With this method, `[s(0),s(s(s(0)))]` would now be a valid shrinked value.

This is implemented building shrinking versions of the properties, similarly to the examples presented, and running them in generation mode. However, since we want shrinking to be an enumeration of simpler values, and not random, the search strategy used is the usual depth-first strategy and not the randomized one presented in Sec. 3. The usual sampling of constraints is used too, instead of the random one.

The number of potential shrinked values grows exponentially with the size of the traces. To mitigate this problem, `LPtest` commits to a shrinked value once it checks that it violates the assertion too, and continues to shrink that value, but never starts from another one on backtracking. Also, the enumeration of shrinked values returns first the values closer to the original term, i.e., if `X` is returned before `Y`, then shrinking `Y` could never produce `X`. Therefore we never repeat a shrinked value [4] in our greedy search for the simplest counterexample.

6 A Case Study

In order to better illustrate our ideas, we present now a case study which consists in testing the correctness of the implementation of some of `CiaoPP`'s abstract domains. In particular, we focus herein on the *sharing-freeness* domain [19] and the correctness of its structure as a lattice and its handling of builtins. Tested predicates include `leq/2`, which checks if an abstract value is below another in the lattice, `lub/3` and `glb/3`, which compute the *least upper bound* and *greatest lower bound* of two abstract values, `builtin_success/3`, which computes the *success substitution of a builtin* from a *call substitution*, and `abstract/2`, which computes the abstraction for a list of substitutions.

[4] Actually, we do not repeat subtraces, but two different subtraces can represent the same value (e.g., there are two ways to obtain s(0) from s(s(0))).

Generation. Testing these predicates required generating random values for abstract values and builtins. The latter is simple: a simple declaration of the regular type `builtin(F,A)`, which simply enumerates the builtins together with their arity, is itself a generator, and using the generation scheme proposed in Sec. 3 it becomes a random generator, while it can still be used as a checker in the run-time checking framework. The same happens for a simple declarative definition of the property `shfr(ShFr,Vs)`, which checks that `ShFr` is a valid *sharing-freeness* value for a list of variables `Vs`. This is however not that trivial and proves that our generation scheme works and is useful in practice, since that property is not a regular type, and among others it includes sharing constraints between free variables. These two properties allowed us to test assertions like the following:

```
:- pred leq_reflexive(X) : shfr(X,_) + not_fails.
leq_reflexive(X) :- leq(X,X).

:- pred lub(X,Y,Z) : (shfr(X,Vs), shfr(Y,Vs)) => (leq(X,Z), leq(Y,Z)).

:- pred builtin_sucess(Blt,Call,Succ)
    : (builtin(F,A), Blt=F/A, length(Vs,A), shfr(Call,Vs))
    + (not_fails, is_det, not_further_inst([X,Y]))}
```

To check some assertions we needed to generate related pairs of abstract values. That is encoded in the precondition as a final literal `leq(ShFr1,ShFr2)`, as in the next assertion:

```
:- pred builtins_monotonic(Blt, X, Y)
    : (builtin(Blt), Blt=F/A, length(Vs,A), shfr(X,Vs), shfr(Y,Vs), leq(X,Y))
    + not_fails.

    builtins_monotonic(Blt,X,Y) :-
    ....
```

In our framework the generation is performed by producing first the two values independently, and checking the literal. This became inefficient, so we decided to write our own generator for this particular case. Finally, we tested the generation for arbitrary terms with the following assertion, which checks that the abstract value resulting from executing a builtin and abstracting the arguments on success is lower than the one resulting of abstracting the arguments on call and calling `builtin_success/3`:

```
:- pred builtin_soundness(Blt, Args)
    : (builtin(Blt), Blt=F/A, length(Args,A), list(Args, term))
    + not_fails.

builtin_soundess(Blt,Args) :- ...
```

Analysis. Many properties used in our assertions were user-defined and not native to `CiaoPP`, so the analysis could not abstract them precisely. However, the analysis did manage to simplify or prove some of the remaining ones, particularly those dealing with determinism (`+ is_det`) and efficiency (`no_choicepoints`).

Additionally, we successfully did the experiment of not defining the regular type `builtin/2`, and letting the analysis infer it on its own and use it for generation. We also tested by hand the finer integration between testing and analysis proposed in Sec. 4: some assertions involving builtins could not be proven for the general case, but this could be done for some of the simpler builtins, and thus testing could be avoided for those particular cases.

Bugs found. We did not find any bugs for the predicates `leq/2`, `lub/2`, and `glb/2`. This was not surprising: they are relatively simple and commonly used in `CiaoPP`. However, we found several bugs in `builtin_success/2`. Some of them were minor and thus had never been found or reported before: some builtin handlers left unnecessary choicepoints, or failed for the abstract value $\bot$ (with which they are never called in `CiaoPP`). Others were more serious: we found bugs for less commonly-used builtins, and even two larger bugs for the builtins `=/2` and `==/2`. The handler failed for the literal `X=X` and for literals like `f(X)==g(Y)`, both of which do not normally appear in realistic programs and thus were not detected before.

7 Related Work

Random testing has been used for a long time in Software Engineering [8]. `QuickCheck` [3] provided the first implementation of a property-based random test generation system. It was first developed for Haskell and functional programming languages in general and then extended to other languages, and has seen significant practical use [14]. It uses a domain-specific language of testable specifications and generates test data based on Haskell types.

`ErlangQuickCheck` and `PropEr` [20] are closely related systems for Erlang, where types are dynamically checked and the value generation is guided by means of functions, using quantified types defined by these generating functions. We use a number of ideas from `QuickCheck` and the related systems, such as applying shrinking to reduce the test cases. However, `LPtest` is based on the ideas of the (earlier) `Ciao` model and we do not propose a new assertion language, but rather use and extend that of the `Ciao` system. This allows supporting Prolog-relevant properties, which deal with non-ground data, logical variables, variable sharing, etc., while `QuickCheck` is limited to ground data. Also, while `QuickCheck` offers quite flexible control of the random generation, we argue that using random search strategies over predicates defining properties is an interesting and more natural approach for Prolog.

The closest related work is `PrologTest` [1], which adapts `QuickCheck` and random property-based testing to the Prolog context. We share many objectives with `PrologTest` but we argue that our framework is more general, with richer properties (e.g., variable sharing), and is combined with static analysis. Also, as in `QuickCheck`, `PrologTest` uses a specific assertion language, while, as mentioned before, we share the `Ciao` assertions with the other parts of the `Ciao` system. `PrologTest` also uses Prolog predicates as random *generators*. This can also be done in `LPtest`, but we also propose an approach which we argue is more elegant, based on separating the code of the generator from the random generation strategy, using the facilities present in the `Ciao` system for running code under different SLD *search rules*, such as breadth first, iterative deepening, or randomized search.

Other directly related systems are `EasyCheck` [2] and `CurryCheck` [9] for the `Curry` language. In these systems test cases are generated from the (strong) types present in the language, as in `QuickCheck`. However, they also deal with determinism and modes. To the extent of our knowledge test case minimization has not been implemented in these systems.

There has also been work on generating test cases using CLP and partial evaluation techniques, both for Prolog and imperative languages (see, e.g., [7, 6] and its references). This work differs from (and is complementary to) ours in that the test

cases are generated via a symbolic execution of the program, with the traditional aims of path coverage, etc., rather than from assertions and with the objective of randomized testing.

Other related work includes *fuzz testing* [18], where "nonsensical" (i.e., fully random) inputs are passed to programs to trigger program crashes, and grammar-based testing, where inputs generation is based on a grammatical definition of inputs (similar to generating with regular types) [10]. Schrijvers proposed `Tor` [23] as a mechanism for supporting the execution of predicates using alternative search rules. Midtgaard and Moller [17] have also applied property-based testing to checking the correctness of static analysis implementations.

8 Conclusions and future work

We have presented an approach and a tool, `LPtest`, for assertion-based random testing of Prolog programs that is integrated with the `Ciao` assertion model. In this context, the idea of generating random test values from assertion preconditions emerges naturally since preconditions are conjunctions of literals, and the corresponding predicates can conceptually be used as generators. `LPtest` generates valid inputs from the properties that appear in the assertions shared with other parts of the model. We have shown how this generation process can be based on running the property predicates under non-standard (random) search rules and how the run time-check instrumentation of the `Ciao` framework can be used to perform a wide variety of checks. We have proposed methods for supporting (C)LP-specific properties, including combinations of shape-based (regular) types and variable sharing and instantiation. We have also proposed some integrations of the test generation system with static analysis and provided a number of ideas for shrinking in our context. Finally, we have shown some results on the applicability of the approach and tool to the verification and checking of the implementations of some of `Ciao`'s abstract domains. The tool is rather new and we are gathering experience with it (which we expect to report on at the conference). However, it has already proven itself quite useful in finding bugs in code that had survived years of use and debugging.

References

1. C. Amaral, M. Florido, and V. Santos Costa. PrologCheck - Property-Based Testing in Prolog. In *Functional and Logic Programming - 12th Int'l. Symp., FLOPS*, volume 8475 of *LNCS*, pages 1–17. Springer, 2014.
2. Jan Christiansen and Sebastian Fischer. EasyCheck - Test Data for Free. In *Functional and Logic Programming, 9th Int'l. Symp., FLOPS*, pages 322–336, April 2008.
3. Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Fifth ACM SIGPLAN Int'l. Conf. on Functional Programming*, ICFP'00, pages 268–279. ACM, 2000.
4. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252. ACM Press, 1977.
5. Cormac Flanagan. Hybrid Type Checking. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*, pages 245–256, January 2006.

6. M. Gómez-Zamalloa, E. Albert, and G. Puebla. On the Generation of Test Data for Prolog by Partial Evaluation. In *Proc. of WLPE'08*, pages 26–43, 2008.
7. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test Case Generation for Object-Oriented Imperative Languages in CLP. *Theory and Practice of Logic Programming, ICLP'10 Special Issue*, 10 (4–6), 2010.
8. Dick Hamlet. Random Testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, page 970–978. Wiley, 1994.
9. Michael Hanus. CurryCheck: Checking Properties of Curry Programs. In *Logic-Based Program Synthesis and Transformation - 26th Int'l. Symp. LOPSTR 2016, Revised Selected Papers*, pages 222–239, September 2016.
10. Mark Hennessy and James F. Power. An Analysis of Rule Coverage as a Criterion in Generating Minimal Test Suites for Grammar-based Software. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 104–113, November 2005.
11. M. V. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *TPLP*, 12(1–2):219–252, 2012. http://arxiv.org/abs/1102.5497.
12. M. V. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, 1999.
13. M. V. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
14. John Hughes. QuickCheck Testing for Fun and Profit. In Michael Hanus, editor, *Practical Aspects of Declarative Languages*, pages 1–32, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
15. D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *North American Conference on Logic Programming*, 1989.
16. E. Mera, P. Lopez-Garcia, and M. V. Hermenegildo. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *25th Int'l. Conference on Logic Programming (ICLP'09)*, volume 5649 of *LNCS*, pages 281–295. Springer-Verlag, July 2009.
17. Jan Midtgaard and Anders Møller. QuickChecking Static Analysis Properties. *Softw. Test., Verif. Reliab.*, 27(6), 2017.
18. Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.
19. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *ICLP'91*, pages 49–63. MIT Press, June 1991.
20. Manolis Papadakis and Konstantinos Sagonas. A PropEr Integration of Types and Function Specifications with Property-Based Testing. In *10th ACM SIGPLAN workshop on Erlang*, pages 39–50, September 2011.
21. G. Puebla, F. Bueno, and M. V. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Proc. of LOPSTR'99*, LNCS 1817, pages 273–292. Springer-Verlag, March 2000.
22. A. Rastogi, N. Swamy, C. Fournet, G.M. Bierman, and P. Vekris. Safe & Efficient Gradual Typing for TypeScript. In *42nd POPL*, pages 167–180. ACM, January 2015.
23. Tom Schrijvers, Bart Demoen, Markus Triska, and Benoit Desouter. Tor: Modular search with hookable disjunction. *Sci. Comput. Program.*, 84:101–120, 2014.
24. Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

Trace analysis using an Event-driven Interval Temporal Logic *

María-del-Mar Gallardo[1] and
Laura Panizo[1]

Universidad de Málaga, Andalucía Tech,
Departamento de Lenguajes y Ciencias de la Computación,
Campus de Teatinos s/n, 29071, Málaga, Spain
{gallardo, laurapanizo}@lcc.uma.es

Abstract. Nowadays, many critical systems can be characterized as hybrid ones, combining continuous and discrete behaviours that are closely related. Changes in the continuous dynamics are usually fired by internal or external discrete events. Due to their inherent complexity, it is a crucial but not trivial task to ensure that these systems satisfy some desirable properties. An approach to analyze them consists of the combination of model-based testing and run-time verification techniques. In this paper, we present an interval logic to specify properties of event-driven hybrid systems and an automatic transformation of the logic formulae into networks of finite-state machines. Currently, we use PROMELA/SPIN to implement the network of finite-state machines, and analyze non-functional properties of mobile applications. We use the TRIANGLE testbed, which implements a controllable network environment for testing, to obtain the application traces and monitor network parameters.

1 Introduction

In the last years, the improvement of sensor technology has led to the development of different software systems that monitor some physical magnitudes to control many everyday tasks. Water resource management systems [8], or aeronautics [9] are some examples of this type of systems. As it is well known, hybrid systems are composed of the so-called discrete and continuous components, which are strongly interrelated. Usually, the role of the discrete part is to control the continuous one, modifying its behaviour when necessary according to some system conditions. The continuous component may follow complex dynamics, which are usually represented by differential equations. The verification of critical properties on these systems is crucial since they may carry out critical tasks that affect the health of people or with a great economic impact. In the literature, hybrid automata constitute the best known mechanism to model

* This work has been supported by the Spanish Ministry of Science, Innovation and Universities project RTI2018-099777-B-I00 and the European Union's Horizon 2020 research and innovation programme under grant agreement No 777517 (EuWireless)

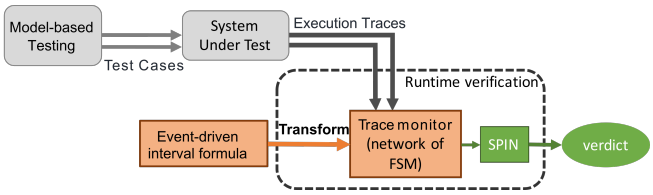


Fig. 1: Approach for testing event-driven hybrid systems

hybrid systems. For example, tools like Uppaal [2] focus on the verification by model checking of some hybrid automata subclasses (timed automata). However, not all hybrid systems can be easily represented as hybrid automata, not only because of their complex dynamics but also because of their interaction with an unpredictable environment. For this reason, in the last decades, other computational hybrid models have appeared such as *extended hybrid systems* [4] in which the hybrid systems are parameterized to incorporate the influence of the environment, or sampled-data control systems [12] in which the continuous and discrete components alternate their execution using a fixed time duration.

In a previous work [6], we proposed a framework to test event-driven hybrid systems using a combination of model-based testing (to automatically generate test cases) and runtime verification (to check the traces obtained against the desirable properties). The framework, shown in Figure 1, was implemented in the context of the TRIANGLE project to analyze non-functional properties on traces produced by the execution of mobile applications. In this work, we implemented an ad-hoc trace monitoring system that was able to analyze some non-functional properties of interest.

In this paper, we concentrate on the trace analysis using runtime verification. In particular, we propose an event-driven linear temporal logic (eLTL) that allows us to extend the set of non-functional properties that can be specified and analyzed in the framework described above. The motivation for the definition of the new logic is twofold. On the one hand, we need a logic in which properties on monitored magnitudes are evaluated on time intervals determined by internal or external events that have occurred during the execution trace. For instance, in the context of mobile applications (apps), in a video streaming app, the video resolution can vary depending on network parameters (e.g. radio technology, signal strength, etc.). The exact moment when the video starts playing is a priori unknown, but during video playback, determined, for instance, by events `vstart` and `vstop`, different network and device parameters must be monitored to determine the suitable video resolution. On the other hand, we also need a logic whose formulae can be transformed into monitors that act as listeners of the trace events to dynamically evaluate the specified property. Thus, the contributions of the paper are both the definition of the event-driven linear time logic eLTL and the transformation of the logic formulae into finite-state machines (FSM) that act as observers of the execution traces. A preliminary version of

the logic was presented in a Spanish workshop [7]. With respect to this former paper, the current version has been extended with a more formal presentation of the logic, and with the implementation section which is completely new.

The paper is organized as follows. Section 2 summarizes some work related to interval logics. Section 3 presents the syntax and semantics of the event-driven interval logic. We also show its expressiveness with some examples and briefly compare eLTL and LTL. Section 4 describes the transformation of each eLTL formula into a network of FSM and proves the correctness of the transformation. Finally, Section 5 gives the conclusions and future work. Appendices contain the proof of all the results presented in the paper, along with the current PROMELA implementation of the network of FSM which allows us to check the satisfaction of eLTL formulae on traces using SPIN [11].

2 Related Work

In Linear Temporal Logic (LTL) is not easy to express requirements to be held in a bounded future. Thus, the extension of LTL with *intervals* seems a natural idea to easily express these other type of properties. This is the approach followed in [20], where the authors use events to determine the intervals on which formulae must be evaluated, although they do not deal with real-time. The temporal logic FIL is also defined with similar purposes but the formulae are written using a graphical representation. Real-time FIL [19] is an extension of FIL that incorporates a new predicate $len(d_1, d_2]$ that bounds the length of the intervals on which properties have to be evaluated. In other context, the duration calculus [3] (DC) was defined to verify real-time systems. In DC system states have a *duration* in each time interval that can be measured taking into account the presence of the state in the interval. DC includes modalities (temporal operators) able to express relations between intervals and states, which constitute the basis of the logic.

The Metric Interval Logic (MITL) [1] is a real-time temporal logic that extends LTL by using modal operators of the form $\Box_I$, $\Diamond_I$ where I is an open/close, bounded/unbounded interval of $\mathbb{R}$. $\text{MITL}_{[a,b]}$ [13] is a bounded version of MITL with all temporal modalities restricted to bounded intervals of the form $[a, b]$. $\text{MITL}_{[a,b]}$ formulae can be translated into deterministic timed automata. More recently, $\text{MITL}_{[a,b]}$ was extended to Signal Temporal Logic STL [14] including numerical predicates that allow analogue and mixed-signal properties to be specified. Lately, the MITL logic has been extended to xSTL [16] by adding *timed regular expressions* to express behaviour patterns to be met by signals.

Finally, the differential dynamic logic (dL) [18] is a specification language to describe safety and liveness properties of hybrid systems. In dL, formulae are of the form $[\alpha]\phi$ or $\langle\alpha\rangle\phi$ meaning that the behaviour of hybrid system α always (eventually) is inside the region defined by ϕ.

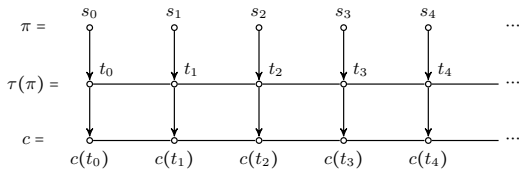


Fig. 2: Synchronization of trace π and continuous variable c using $\tau(\pi)$

3 Event-driven Systems and Logic eLTL

In this section, we introduce a general model of event-driven hybrid systems, which is characterized by containing continuous variables whose values can be monitored. From a very abstract perspective, the behaviour of such a system may be given by a transition system $P = \langle \Sigma, \overset{-}{\longmapsto}, L, s_0 \rangle$ where Σ is a non-enumerable set of observable states, L is a finite set of labels, $\overset{-}{\longmapsto} \subseteq \Sigma \times L \times \Sigma$ is the transition relation, and $s_0 \in \Sigma$ is the initial state. Transitions labels represent the external/internal system events or system instructions that make the system evolve. In addition, we assume that $\iota \in L$ is an special label that represents the time passing between two successive states during which no event or instruction is executed. Thus, transitions may take place when an event arrives, when a system discrete instruction is carried out, or when a continuous transition occur in which the only change in the state is the passing of time.

We denote with $\mathcal{O}_f(P)$ the set of execution traces of finite length determined by P. The elements of $\mathcal{O}_f(P)$ are traces of the form $\pi = s_0 \overset{l_0}{\longmapsto} s_1 \overset{l_1}{\longmapsto} \cdots \overset{l_{n-2}}{\longmapsto} s_{n-1}$ where each $l_i \in L$ is the event/instruction/ι that fired the transition. The length of a trace $\pi = s_0 \overset{l_0}{\longmapsto} s_1 \overset{l_1}{\longmapsto} \cdots \overset{l_{n-2}}{\longmapsto} s_{n-1}$ is the number of its states n. Given a trace π of length n, we define the set $\mathcal{O}bs(\pi)$ of observable states of π; that is, $\mathcal{O}bs(\pi) = \{s_0, \cdots, s_{n-1}\}$. It is worth noting that although event-driven hybrid systems have continuous variables, we assume that their values are only visible at observable states. In addition, we assume that the time instant in which each state occurs is given by function $\tau : \Sigma \longrightarrow \mathbb{R}^{\geq 0}$ which relates each state s with the moment it happens $\tau(s) \in \mathbb{R}^{\geq 0}$.

In the following, given a trace π of length n and $t \in \{\tau(s_0), \cdots \tau(s_{n-1})\}$, we denote with $\langle \pi, t \rangle$ the observable state s_i of the trace at time instant t. In addition, we use function $\sigma : \{\tau(s_0), \cdots \tau(s_{n-1})\} \rightarrow \mathcal{O}bs(\pi)$ as the inverse function of τ, i.e., $\forall 0 \geq i < n . \tau(\sigma(t_i)) = t_i$ and $\sigma(\tau(s_i)) = s_i$.

Each continuous variable c of the system is a function $c : \mathbb{R}^{\geq 0} \longrightarrow \mathbb{R}$ that gives the value of c, $c(t)$, at each time instant t. Figure 2 shows the relation between the states in a trace, the time instants where they occur and the corresponding values of continuous variable c at these instants. By abuse of notation, in the figure and in the rest of the section, we use $\tau(\pi)$ to denote set $\{\tau(s_0), \cdots \tau(s_{n-1})\}$.

We have decided to define the behaviour of event-driven hybrid systems by means of the simple notion of transition systems on purpose. The definition is

highly general in the sense that it is able to capture the behaviour of hybrid event-driven systems described by hybrid automata or other formalisms. Transitions correspond to changes of the system variables producing *observable states* in the traces that can be the result of the system that accepts an event or executes an instruction, or the result of an internal evolution ι where time passing is the only change in the trace. Anyway, the number of observable states in each trace is finite. In practice, in our current implementation, the time instants and the value of continuous variables in traces is recorded in log files, although other time models could also be managed by the logic presented below.

3.1 Syntax and Semantics of eLTL

We consider two types of state formulae to be analyzed on states of Σ. On the one hand, we have those that can be evaluated on single states as used in propositional linear temporal logic LTL, for instance. On the other hand, we assume that events of L are also state formulae that can be checked on states. Thus, let $\mathcal{F}$ be the set of all state formulae to be evaluated on elements of Σ. As usual, we suppose that state formulae may be constructed by combining state formulae and Boolean operators. Relation $\vdash \subseteq \Sigma \times \mathcal{F}$ associates each state with the state formulae it satisfies, that is, given $s \in \Sigma$, and $p \in \mathcal{F}$, $s \vdash p$ iff the state s satisfies the state formula p. In the following, given $\pi \in \mathcal{O}_f(P)$, $t_i \in \tau(\pi)$ and $p \in \mathcal{F}$, we write $\langle \pi, t_i \rangle \vdash p$ iff $\sigma(t_i) \vdash p$. When $l_i \in L$ is an event occurred at state s_i that evolves to s_{i+1} in trace $\pi = s_0 \xmapsto{l_0} s_1 \xmapsto{l_1} \cdots \xmapsto{l_{n-2}} s_{n-1}$, we assume that state s_{i+1} records the fact that l_i has just occurred and, in consequence, we have that $s_{i+1} \vdash l_i$, or equivalently, $\langle \pi, t_{i+1} \rangle \vdash l_i$. Other logics such as HML [10] or ACTL [5] focus on actions versus state formulae. We have decided to keep them at the same level to allow the use of both in the logic.

In order to analyze the behaviour of continuous variables, it is useful to observe them not only in a given time instant, but also during *time intervals* to know, for example, whether their values hold inside some expected limits or whether they never exceed a given threshold. To this end, we use *intervals of states* (inside the traces) to determine the periods of time during which continuous variables should be observed. Our proposal is inspired in the interval calculus introduced by [3], where the domain of interval logic is the set of time intervals $\mathbb{I}$ defined as $\{[t_1, t_2] | t_1, t_2 \in \mathbb{R}^{\geq 0}, t_1 \leq t_2\}$. Considering this, we define the so-called *interval formulae* as functions of the type $\phi : \mathbb{I} \to \{true, false\}$ to represent the formulae that describe the expected behaviour of continuous variables on time intervals. For instance, assume that $c : \mathbb{R}^{\geq 0} \to \mathbb{R}$ is a continuous variable of our system. Given a constant $K \in \mathbb{R}^{\geq 0}$, function $\phi_c : \mathbb{I} \to \{true, false\}$ given as $\phi_c([t_1, t_2]) = |c(t_2) - c(t_1)| < K$ defines an interval formula that is *true* on an interval $[t_1, t_2]$ iff the absolute value of difference between c in the interval endpoints t_1 and t_2 is less than K. Let us denote with Φ the set of interval formulae. We assume that Φ contains the special interval formula $True : \mathbb{I} \to \{true, false\}$ that returns *true* for all positive real intervals, that is, $\forall I \in \mathbb{I}. True(I) = true$.

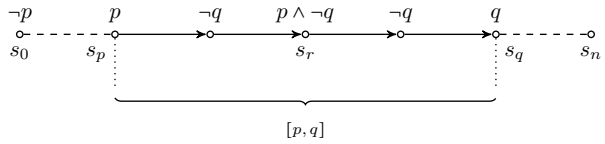
In the following, given two state formulae $p, q \in \mathcal{F}$, we use expressions of the form $[p, q]$, that we call *event intervals*, to delimit intervals of states in traces.

Intuitively, given a trace $\pi = s_0 \overset{l_0}{\longmapsto} s_1 \overset{l_1}{\longmapsto} \cdots \overset{l_{n-2}}{\longmapsto} s_{n-1}$, $[p,q]$ represents time intervals $[t_i,t_j]$ with $t_i,t_j \in \tau(\pi)$ such that $\langle \pi,t_i\rangle \vdash p$ and $\langle \pi,t_j\rangle \vdash q$; that is, $s_i \vdash p$ and $s_j \vdash q$. We also consider simple state formulae p to denote states in π satisfying p. Now, we formally define relation $\Vdash$ that relates event intervals with intervals of states in traces.

Definition 1. *Given a trace $\pi \in \mathcal{O}_f(P)$, two state formulae $p,q \in \mathcal{F}$ and two time instants $t_p,t_q \in \tau(\pi)$ such as $t_p < t_q$, we say that the time interval $[t_p,t_q]$ satisfies the event interval $[p,q]$, and we denote it as $\pi \downarrow [t_p,t_q] \Vdash [p,q]$, iff the following four conditions hold: (1) $\langle \pi,t_p\rangle \vdash p$; (2) $\forall t_j \in (t_p,t_q) \cap \tau(\pi), \langle \pi,t_j\rangle \nvdash q$; (3) $\langle \pi,t_q\rangle \vdash q$; and (4) there exists no interval $[t'_p,t_q] \neq [t_p,t_q]$, verifying conditions 1–3 of this definition, such that $[t_p,t_q] \subset [t'_p,t_q]$.*

That is, $\pi \downarrow [t_p,t_q] \Vdash [p,q]$ iff $\sigma(t_p) = s_p$ satisfies p and $\sigma(t_q) = s_q$ is the first state following s_p that satisfies q. In addition, the fourth condition ensures that the interval of states from s_p until s_q is maximal in the sense that it is not possible to find a larger interval ending at s_q satisfying the previous conditions. This notion of maximality guarantees that the evaluation of interval formulae starts at the state when event p first occurs, although it could continue being true in some following states. In the previous definition, the time instants t_p and t_q must be different elements of $\tau(\pi)$, that is, $[t_p,t_q]$ cannot be a point.

Example 1. The following trace (π) tries to clarify Definition 1. Given $p,q \in \mathcal{F}$, and assuming that $\tau(s_i) = t_i$ for all states, we have that $\pi \downarrow [t_p,t_q] \Vdash [p,q]$, but $\pi \downarrow [t_r,t_q] \nVdash [p,q]$, since condition (4) does not hold.



Definition 2. *[eLTL formulae] Given $p,q \in \mathcal{F}$, and $\phi \in \Phi$, the formulae of eLTL logic are recursively constructed as follows:*

$$\psi ::= \phi \mid \neg\psi \mid \psi_1 \vee \psi_2 \mid \psi_1 \mathcal{U}_{[p,q]} \psi_2 \mid \psi_1 \mathcal{U}_p \psi_2$$

The rest of the temporal operators are accordingly defined as:
$\Diamond_{[p,q]}\psi \equiv True\ \mathcal{U}_{[p,q]}\psi$, $\Box_{[p,q]}\psi \equiv \neg(\Diamond_{[p,q]}\neg\psi)$,
$\Diamond_p\psi \equiv True\ \mathcal{U}_p\psi$, $\Box_p\psi \equiv \neg(\Diamond_p\neg\psi)$

The following definition gives the semantics of eLTL formulae given above. Given a trace $\pi \in \mathcal{O}_f(P)$, and $t_i,t_f \in \tau(\pi)$ with $t_i \leq t_f$, we use $\langle \pi,t_i,t_f\rangle$ to represent the subtrace of π from state $s_i = \sigma(t_i)$ to state $s_f = \sigma(t_f)$.

Definition 3 (Semantics of eLTL formulae).
Given $p,q \in \mathcal{F}$, $\phi \in \Phi$, and the eLTL formulae ψ,ψ_1,ψ_2, the satisfaction relation $\vDash$ is defined as follows:

$$\langle \pi,t_i,t_f\rangle \vDash \phi \qquad \textit{iff} \quad \phi([t_i,t_f]) \tag{3.1}$$

$$\langle \pi, t_i, t_f \rangle \vDash \neg\psi \quad \textit{iff} \quad \langle \pi, t_i, t_f \rangle \nvDash \psi \tag{3.2}$$

$$\langle \pi, t_i, t_f \rangle \vDash \psi_1 \vee \psi_2 \quad \textit{iff} \quad \langle \pi, t_i, t_f \rangle \vDash \psi_1 \textit{ or } \langle \pi, t_i, t_f \rangle \vDash \psi_2 \tag{3.3}$$

$$\langle \pi, t_i, t_f \rangle \vDash \psi_1 \mathcal{U}_{[p,q]} \psi_2 \quad \textit{iff} \quad \exists I = [t_p, t_q] \subseteq [t_i, t_f] \textit{ such that } \pi \downarrow [t_p, t_q] \Vdash [p, q] \textit{ and } \langle \pi, t_i, t_p \rangle \vDash \psi_1, \langle \pi, t_p, t_q \rangle \vDash \psi_2 \tag{3.4}$$

$$\langle \pi, t_i, t_f \rangle \vDash \psi_1 \mathcal{U}_{p} \psi_2 \quad \textit{iff} \quad \exists t_p.\ t_i \leq t_p \leq t_f \textit{ and } \langle \pi, t_i, t_p \rangle \vDash \psi_1, \langle \pi, t_p, t_p \rangle \vDash \psi_2 \tag{3.5}$$

The semantics given by $\vDash$ is similar to that of LTL, except that $\vDash$ manages interval formulae instead of state formulae. For instance, case 3.1 states that the subtrace $\langle \pi, t_i, t_f \rangle$ of π satisfies an interval formula ϕ iff $\phi([t_i, t_f])$ holds. Case 3.4 establishes that $\mathcal{U}_{[p,q]}$ holds on the subtrace $\langle \pi, t_i, t_f \rangle$ iff there exists an interval $[t_p, t_q] \subset [t_i, t_f]$ such that ψ_1 and ψ_2 hold on $[t_i, t_p]$ and $[t_p, t_q]$, respectively. Case 3.5 is similar except for the interval in which ψ_2 has to be true is $[t_p, t_p]$, which represents the time instant t_p.

Proposition 1. *The semantics of operators $\Box_{[p,q]}, \Diamond_{[p,q]}, \Box_p$ and $\Diamond_p$, given in Definition 2, is the following:*

$$\langle \pi, t_i, t_f \rangle \vDash \Diamond_{[p,q]} \psi \quad \textit{iff} \quad \exists I = [t_p, t_q] \subseteq [t_i, t_f],\ \textit{such that } \pi \downarrow [t_p, t_q] \Vdash [p, q] \textit{ and } \langle \pi, t_p, t_q \rangle \vDash \psi \tag{3.6}$$

$$\langle \pi, t_i, t_f \rangle \vDash \Box_{[p,q]} \psi \quad \textit{iff} \quad \forall I = [t_p, t_q] \subseteq [t_i, t_f],\ \textit{if } \pi \downarrow [t_p, t_q] \Vdash [p, q] \textit{ then } \langle \pi, t_p, t_q \rangle \vDash \psi \tag{3.7}$$

$$\langle \pi, t_i, t_f \rangle \vDash \Diamond_{p} \psi \quad \textit{iff} \quad \exists t_p \in [t_i, t_f] \textit{ such that } \langle \pi, t_p \rangle \vdash p \textit{ and } \langle \pi, t_p, t_p \rangle \vDash \psi \tag{3.8}$$

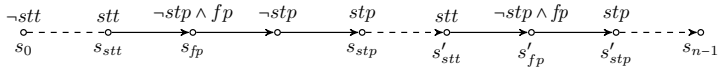
$$\langle \pi, t_i, t_f \rangle \vDash \Box_{p} \psi \quad \textit{iff} \quad \forall t_p \in [t_i, t_f] \textit{ if } \langle \pi, t_p \rangle \vdash p \textit{ then } \langle \pi, t_p, t_p \rangle \vDash \psi \tag{3.9}$$

3.2 Examples

We now give some examples to show the use of the logic. In [6, 17], we proposed a model-based testing approach to test mobile applications (apps) under different network scenarios. We automatically generated app user flows, that is, different interactions of the user with the app, using model-based testing techniques. Then, we executed these app user flows in the TRIANGLE testbed, which provides a controlled mobile network environment, to obtain measurements and execution traces in order to evaluate the performance of the apps.

In this section, we make use of eLTL to describe desirable properties regarding to the values of continuous variables of the ExoPlayer app, a video streaming mobile app that implements different adaptive video streaming protocols. Using the current implementation of the eLTL operators, and with the execution traces provided by the evaluation presented in [17], we can determine if the execution traces of ExoPlayer satisfy the properties. The execution traces of the app contain the following events: the start of video playback (*stt*), the load of the first complete picture (*fp*), the end of the video playback (*stp*), and the changes in the video resolution (*low*, *high*). In addition, the TRIANGLE testbed measures every second (approximately) the amount and rate of transmitted and received data, as well as different parameters of the network (e.g. signal strength and signal quality) and the device (e.g. RAM, CPU and radio technology).

Property 1: We can write the property "*during video playback, the first picture must be loaded at least once in all network conditions*" which may be specified using the formula $\Box_{[stt,stp]} \Diamond_{fp} \mathit{True}$. The following trace satisfies this property, where the expressions over each state represent the state formulae it holds.



Property 2: We can also specify the property "*during video playback, if the video resolution is high, the average received data rate is greater than 5 Mbps, and if the video resolution is low the average data rate is below 1Mbps.*" The video resolution is high in the time interval between h and l events. Similarly, the video resolution is low between the events l and h. The eLTL formula is

$$\Box_{[stt,stp]}(\Box_{[high,low]}\phi_1 \wedge \Box_{[low,high]}\phi_2)$$

where ϕ_1 and ϕ_2 are defined as: $\phi_1([t_i,t_f]) = RxRate(t_i,t_f) \geq 5Mbps$ and $\phi_2([t_i,t_f]) = RxRate(t_i,t_f) \leq 1Mbps$.

This formula uses function $RxRate(t_i,t_f)$ that accesses to the file of the trace and workouts the average in the corresponding time interval. In the current implementation on SPIN, it is calculated using PROMELA embedded C code.

Property 3: The eLTL formula for property "*during video playback, if the video resolution changes from* High *to* Low*, the peak signal strength (rssi) is less than -45 dBm*" can be written as:

$$\Box_{[stt,stp]}\,(\Box_{[high,low]}\phi), \quad where \;\; \phi([t_i,t_f]) = \begin{cases} true & if\ \exists t \in [t_i,t_f], \\ & maxRSSI(t) \leq -45dBm \\ false & otherwise \end{cases}$$

Using this formula with different thresholds for the peak rssi, we can determine whether the adaptive protocols take into consideration the signal strength in the terminal to make a decision and change the video resolution.

Other examples In the health field, eLTL can also be useful. For instance, patients with type 1 diabetics should be monitored to assure that their glucose levels are always inside safe limits. Related to this problem, we could describe different properties of interest. Given the interval formula $\psi_K([t_1,t_2]) = t_2 - t_1 \geq K$ with $K \in \mathbb{R}$, and events *sleep*, *awake*, *run*, *end*, *break*, *endBreak*, *drink* and *over*70 that denote when the patient goes to sleep, awakes, starts running, stops running, drinks and his/her glucose level is over $70mg/l$:

- Property "*while sleeping, the glucose level is never below* $70mg/l$" can be expressed as $\Box_{[sleep,awake]}\Box_{true}\, over70$. Observe that in this property $over70$ acts as a simple interval formulae that holds on each state inside $[sleep, awake]$ iff the glucose level is over 70.
- Property "*if the patient is running more than 60 minutes, he/she has to make a stop of more than 5 minutes to drink*" can be written as

$$\Box_{[run,end]}(\psi_{60} \rightarrow (\Diamond_{[break,endBreak]}(\psi_5 \wedge \Diamond_{drink}\, True)))$$

3.3 Comparison with LTL

In this section, we briefly compare the expressiveness of logics LTL and eLTL. One important difference between both logics is that LTL is evaluated on infinite traces while, on the contrary, eLTL deals with finite traces. This makes some LTL properties hard to specify in eLTL. In addition, eLTL is thought to analyze extra-functional properties on traces, that is, properties that refer to the behaviour of certain magnitudes in subtraces (as in the examples presented above), which cannot easily be expressed in LTL. The context where eLTL formulae are checked is determined by the event intervals $[p, q]$ associated to the modal operators. However, this context is implicit in LTL since formulae are evaluated on the whole infinite trace. In conclusion, we can say that although both logic have similarities, they are different regarding expressiveness. The following table shows some usual patterns of LTL formulae with its corresponding eLTL versions. The inverse transformation is not so easy. For instance, eLTL formula $\Box_{[a,b]} \Diamond_p \mathit{True}$, which forces that p occurs between each pair of a and b events, is hard to write in LTL. In the table, we use interval formulae ϕ_p ($p \in \mathcal{F}$) defined as $\phi_p([t_i, t_f]) = \sigma(t_i) \vdash p$. In addition, $a, b, q \in \mathcal{F}$ are events used to delimit finite subtraces.

LTL	eLTL	Comments
$\Diamond p$	$\Diamond_p \mathit{True}$	In both cases, p has to be *true* eventually, but in eLTL, p must be *true* inside of the finite trace
$\Box p$	$\Box_{true} \phi_p$	In both cases, p has to be always *true*, but in eLTL it is limited to the states of the finite trace
$\Box \Diamond p$	$\Box_{[a,b]} \Diamond_p \mathit{True}$	In LTL, p has to be *true* infinitely often. In eLTL, p has to occur always inside the subtraces determined by $[a, b]$
$\Diamond \Box p$	$\Diamond_{[a,b]} \Box_{true} \phi_p$	The LTL formula says that p has to be always *true* from some unspecified state. The eLTL says the same, but limited by the extreme states of the finite trace $[a, b]$
$p \mathcal{U} q$	$(\Box_{true} \phi_p) \mathcal{U}_q \mathit{True}$	In this case, the LTL formula is clearly easier to write, since eLTL is thought to evaluate magnitudes on subtraces.
$(\Diamond p) \mathcal{U} q$	$(\Diamond_p \mathit{True}) \mathcal{U}_q \mathit{True}$	the LTL formula could be *true* even if p occur after p in the trace. However, in the eLTL version, p has to occur *before* p.

4 Implementation

In this section, we describe the translation of eLTL formulae into a network of state machines $\mathcal{M}$ that check the satisfiability of the property on execution traces. As described in Section 3, formulae are evaluated against time bounded traces π that execute in time intervals of the form $[t_i, t_f]$. Formulae can include nested temporal operators whose evaluation can be restricted to subintervals. The implementation described below assumes that traces are analyzed *offline*, i.e., given a particular trace, for each state, we have stored the time instant when it occurred and the set of state formulae of $\mathcal{F}$ which it satisfies. In consequence, we can use the trace to build a simple state machine $\mathcal{T}$ that runs concurrently

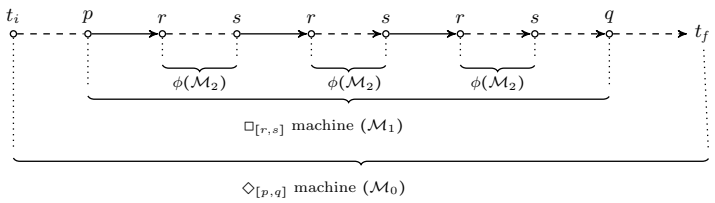


Fig. 3: Example of a network of state machines

with the network of machines $\mathcal{M}$. $\mathcal{T}$ sends to $\mathcal{M}$ events to start and finalize the analysis, and also the events included in the formula which are of interest for the correct execution of the network.

We use an example to intuitively explain how the network of machines $\mathcal{M}$ is constructed. Assume we want to evaluate $\langle \pi, t_i, t_f \rangle \vDash \Diamond_{[p,q]} \Box_{[r,s]} \phi$. The outer operator $\Diamond_{[p,q]}$ must find the different time intervals $[t_p, t_q] \subseteq [t_i, t_f]$, delimited by events p and q, to check if there exists at least one satisfying the sub-formula $\Box_{[r,s]}\phi$. Similarly, given one of the time intervals $[t_p, t_q]$, the inner operator $\Box_{[r,s]}$ has to find all time intervals $[t_r, t_s] \subseteq [t_p, t_q]$, determined by events r and s, to check whether ϕ holds in all of them.

The network $\mathcal{M}$ is composed of the parallel composition of *finite-state machines*, each one monitoring a different sub-formula. The network is hierarchized, that is, each state machine communicates through channels with the trace $\mathcal{T}$ being analysed and with the state machine of the formula in which it is nested. The state machine of the outer `eLTL` temporal operator starts and ends the evaluation, reporting the analysis result to $\mathcal{T}$. Each state machine has a unique identifier `id` which allows it to access the different input/output channels. Thus, channel `cm[id]` is a synchronous channel through which the state machine `id` is started and stopped. Channel `rd[id]` is used by machine `id` to send the result of its evaluation. Finally, `ev[id]` is an asynchronous channel through which each state machine receives from $\mathcal{T}$ the events in which it is interested along with the time instant they have occurred (`[`t_e`,e]`).

Figure 3 gives an intuition about how the network of state machines of the example is constructed. The network of the example is composed by three state machines $\mathcal{M}_0 \| \mathcal{M}_1 \| \mathcal{M}_2$. $\mathcal{M}_0$ is the highest level state machine that monitors operator $\Diamond_{[p,q]}$. Thus, it should receive from $\mathcal{T}$ events p and q each time they occur in the trace. Similarly, $\mathcal{M}_1$ monitors $\Box_{[r,s]}$, and it should be informed when events r or s occur. Finally, $\mathcal{M}_2$ is devoted to checking ϕ. It is worth noting that all machines are initially active, although they are blocked until the reception of the start message `STT`. Machine $\mathcal{A}_0$ is started when $\mathcal{T}$ begins its execution. Each machine `id` receives the start and stop messages `STT` and `STP` though channel `cm[id]`. Events arrive to machine `id` via channel `ev[id]` and it returns the result of its evaluation (*true* or *false*) using channel `rd[id]`. In the example, when $\mathcal{M}_0$ receives event p, it sends a `STT` message to the nested

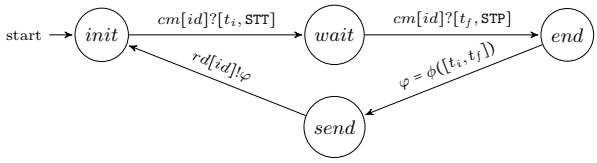


Fig. 4: State machine of an interval formulae

machine $\mathcal{M}_1$. Similarly, when $\mathcal{M}_1$ receives event r, it sends message `STT` to $\mathcal{M}_2$. Each time $\mathcal{M}_1$ receives event s sends `STP` to $\mathcal{M}_2$.When $\mathcal{M}_2$ receives `STP`, it ends its execution, evaluates the interval formula and sends the result through channel `rd[2]`. Similarly, when $\mathcal{M}_0$ receives event q via channel `ev`[0], it sends `STP` to $\mathcal{M}_1$. When a machine receives `STP`, it *tries* to finish its execution immediately. But, before stopping, it has to process all the events stored in its `ev` channel since they could have occurred before the `STP` were sent. To know this, each message contains a timestamp with the time instant when the event took place. This is needed since events and `STP` are sent via different channels. Thus, it is possible for a machine to read `STP` before reading a previous event in the trace.

Finite-state Machine Templates We now show finite-state machines *templates* that implement `eLTL` operators. In these machines, `id` refers to the state machine being implemented, and `c1`, `c2` are, respectively, the identifiers of the state machines of the first and the second nested operators, if they exist. All machines described below follow the same pattern. First, each machine starts after receiving message `STT`, and initiates its sub-machines, if necessary. Then, it continues processing the input events in which it is interested. These events are directly sent from the instrumented trace $\mathcal{T}$ that is being monitored. When the machine receives `STP`, it returns the result of its evaluation via channel `rd`. To simplify the diagrams, we have used sometimes guarded transitions of the form $G|Action$. When guard G is a message reception via a synchronous channel, it is executable iff it is possible to read the message and, as a side effect, the message is extracted from the channel. Due to lack of space, we have not included operator $\mathcal{U}_p$ since its machine is a simplified version of that of $\mathcal{U}_{[p,q]}$.

Interval formula (ϕ): Figure 4 shows the state machine for an interval formula ϕ without `eLTL` operators. An interval formula is evaluated on a time interval $[t_i, t_f]$ that is communicated to the process via the channel `cm[id]` with messages `[`t_i`,STT]` and `[`t_f`,STP]`. After detecting the interval end, the state machine evaluates the expression $\phi([t_i, t_f])$ and sends the result to the parent machine through `rd[id]`.

Negation ($\neg\psi$): Figure 5 shows the negation operator. The state machine synchronizes with the machine of its nested formula (ψ) as soon as it receives the `STT` and `STP` commands. Observe that, in this case, to simplify the diagram, we

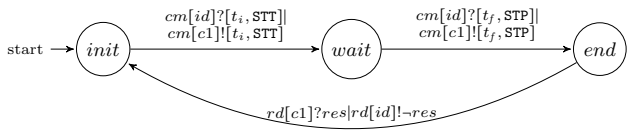


Fig. 5: State machine of the *not* operator

have used guarded transitions. When the nested machine finishes, the machine negates the result and returns it through the `rd` channel.

Or ($\psi_1 \vee \psi_2$): Figure 6 shows the state machine of the *or* operator. This machine checks if any of the two sub-formulae ψ_1 and ψ_2 holds on the same interval $[t_i, t_f]$ over which the *or* operator is being evaluated. Similarly to the `NOT` machine, this machine waits the successive reception of the `STT` and `STP` messages and resends them to the machines of its sub-formulae to start and stop them.

Until ($\psi_1 \mathcal{U}_{[p,q]} \psi_2$): Figure 7 shows the state machine template of the *until* operator. As can be observed, it is much more complex than the previous templates. Assuming that the whole formula is evaluated on interval $[t_i, t_f]$, the first sub-formula ψ_1 must be true on an interval $[t_i, t_p]$ (t_p being a time instant when event p has occurred), and the second one ψ_2 must be true on the time interval $[t_p, t_q]$ (t_q being the time instant when event q has first occurred after p). The machine `id` starts accepting the message `STT` from its parent and, then, it resends the message to the state machine of ψ_1. In state `wait p`, the machine is waiting for the p event to occur or for the `STP` message to arrive. If p arrives at a correct time instant (after t_i), the machine sends `STP` to machine `c1` and waits for its result in state `wait c1`. If ψ_1 is not valid in the interval $[t_i, t_p]$, machine `id` records *false* in variable `res1` and waits in state `wait q` the following event q, then it transits to `wait p` and restarts machine `c1`. This is because machine `id` has found that formula does not hold on a time interval determined by the occurrence of p and q and, in consequence, it starts searching for the following

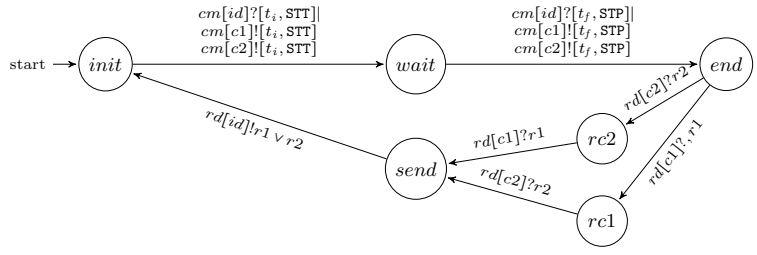


Fig. 6: State machine for *or* operator

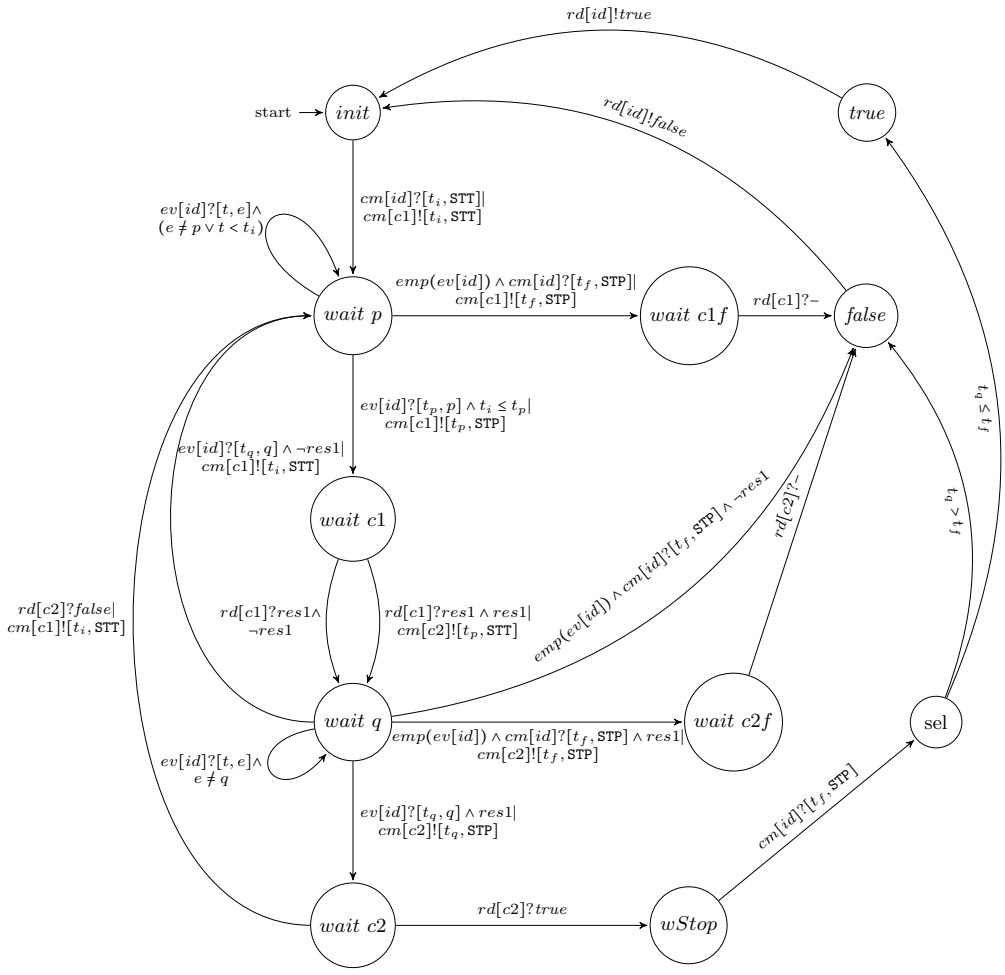


Fig. 7: State machine for *until* operator

interval given by $[p, q]$ in the trace. Otherwise, if ψ_1 holds on $[t_i, t_p]$, machine sends `STT` to machine `c2` and waits for its result in state `wait c2`. In this state, machine `id` behaves in a similar way as in state `wait p`. If `c2` returns *false*, it restarts again machine `c1` and goes back to state `wait p` to search for the following time interval determined by events p and q. Conversely, if `c2` returns *true*, machine `id` waits for message `STP` to send its result. Note that it only sends *true* if event q has occurred before the end of the interval t_f. Otherwise, the machine returns *false*, since ψ_2 could not be evaluated in time. Observe that in states `wait p` and `wait q`, message `STP` is only accepted when the event channel is empty (`emp(ev[id])`). This is to prioritize reading events p and q before `STP` and simplify the implementation.

Theorem 1. *Let f be an eLTL formula, and $\mathcal{M}_{id}$ the network of state machines implementing f, then given a finite trace $\langle \pi, t_i, t_f \rangle$, $\langle \pi, t_i, t_f \rangle \vDash f$ if and only if $\mathcal{M}_{id}$ finishes its execution by sending true via channel `rd[id]` (`rd[id]!`true).*

5 Conclusions

In this paper, we have presented an event-driven interval logic (eLTL) suitable for describing properties in terms of time intervals determined by trace events. We have transformed each eLTL formula into network of finite state machines to evaluate it using runtime verification procedures, and have proved the correctness of the transformation. We have constructed a prototype implementation of these machines in Promela to be executed on Spin.

Our final goal is to apply the approach to analyze execution traces of real systems against extra-functional properties, such as evaluating the performance of mobile apps in different network scenarios [17]. Currently, the transformation from eLTL formula into Promela code, and the transformation of the traces are manually done, although the automatic transformation will be carried out in the near future. We also plan to use the approach in other domains such as the EuWireless project [15]. This project is designing an architecture to dynamically create network slices to run experiments. In this context, it is of great importance to monitor the different network slices and the underlying infrastructure to ensure safety (e.g. isolation of slices) and extra-functional properties related to performance and quality of service.

Bibliography

[1] Alur, R., Feder, T., Henzinger, T.A.: The benefits of relaxing punctuality. J. ACM **43**(1), 116–146 (Jan 1996)
[2] Behrmann, G., David, A., Larsen, K.: A Tutorial on Uppaal. In: Formal Methods for the Design of Real-Time Systems. pp. 200–237. No. 3185 in LNCS, Springer–Verlag (September 2004)
[3] Chaochen, Z., Hansen, M.R.: Duration Calculus - A Formal Approach to Real-Time Systems. Monographs in Theoretical Computer Science. An EATCS Series, Springer (2004)

[4] Dang, T., Nahhal, T.: Coverage-guided test generation for continuous and hybrid systems. Form. Methods Syst. Des. **34**(2), 183–213 (Apr 2009)
[5] De Nicola, R., Vaandrager, F.W.: A temporal dynamic logic for verifying hybrid system invariants. In: Proc. Ecole de Printemps on Semantics of Concurrency. LNCS, vol. 496, pp. 407–419. Springer (1990)
[6] Espada, A.R., Gallardo, M.M., Salmerón, A., Panizo, L., Merino, P.: A formal approach to automatically analyze extra-functional properties in mobile applications. STVR **29**(4-5) (2019)
[7] Gallardo, M.M., Panizo, L.: An Event-driven interval temporal logic for hybrid systems (Work in progress). In: Actas de las XVIII Jornadas de Programación y Lenguajes (PROLE 2018), Sevilla. (2018)
[8] Gallardo, M.M., Merino, P., Panizo, L., Linares, A.: A practical use of model checking for synthesis: generating a dam controller for flood management. Software Practice & Experience **41**(11), 1329–1347 (Jan 2011)
[9] Goodloe, A., Muñoz, C., Kirchner, F., Correnson, L.: Verification of numerical programs: From real numbers to floating point numbers. In: 5th Int. Symp. on NASA Formal Methods. LNCS, vol. 7871, pp. 441–446 (2013)
[10] Hennessy, M., Milner, R.: On observing nondeterminism and concurrency. In: de Bakker, J., van Leeuwen, J. (eds.) Automata, Languages and Programming. pp. 299–309. Springer Berlin Heidelberg (1980)
[11] Holzmann, G.: The model checker SPIN. IEEE Transactions on Software Engineering **23**(5), 279–295 (1997)
[12] Lerda, F., Kapinski, J., Maka, H., Clarke, E.M., Krogh, B.H.: Model checking in-the-loop: Finding counterexamples by systematic simulation. In: 2008 American Control Conference. pp. 2734–2740 (June 2008)
[13] Maler, O., Nickovic, D., Pnueli, A.: Real Time Temporal Logic: Past, Present, Future. In: Formal Modeling and Analysis of Timed Systems. pp. 2–16. Springer (2005)
[14] Maler, O., Ničković, D.: Monitoring properties of analog and mixed-signal circuits. STTT **15**(3), 247–268 (2013)
[15] Merino, P., Panizo, L., Díaz, A., et al.: EuWireless: Design of a pan-European mobile network operator for research. In: European Conference on Networks and Communications (EuCNC2018). pp. 392–393 (2018)
[16] Ničković, D., Lebeltel, O., Maler, O., Ferrère, T., Ulus, D.: AMT 2.0: Qualitative and quantitative trace analysis with extended signal temporal logic. In: 24th Int. Conf.TACAS. pp. 303–319 (2018)
[17] Panizo, L., Diaz-Zayas, A., Garcia, B.: Model-based testing of apps in real network scenarios. STTT pp. 1–10 (2019)
[18] Platzer, A.: A temporal dynamic logic for verifying hybrid system invariants. LNCS, vol. 4514, pp. 457–471. Springer (2007)
[19] Ramakrishna, Y., Melliar-Smith, P., Moser, L., Dillon, L., Kutty, G.: Interval logics and their decision procedures: Part ii: a real-time interval logic. Theoretical Computer Science **170**(1), 1 – 46 (1996)
[20] Schwartz, R.L., Melliar-Smith, P.M., Vogt, F.H.: An interval logic for higher-level temporal reasoning. In: Procs. of the 2nd ACM Symp. on Principles of Distributed Computing. pp. 173–186 (1983)

The Prolog debugger and declarative programming

Włodzimierz Drabent

Institute of Computer Science, Polish Academy of Sciences, and
IDA, Linköpings universitet, Sweden
drabent *at* ipipan *dot* waw *dot* pl

Version 2.1, September 2019

Abstract. Logic programming is a declarative programming paradigm. Programming language Prolog makes logic programming possible, at least to a substantial extent. However the Prolog debugger works solely in terms of the operational semantics. So it is incompatible with declarative programming. This report discusses this issue and tries to find how the debugger may be used from the declarative point of view. Also, the box model of Byrd, used by the debugger, is explained in terms of SLD-resolution.

1 Introduction

The idea of logic programming is that a program is a set of logic formulae, and a computation means producing logical consequences of the program. The logical consequence is that of the standard first order logic. So it is a *declarative* programming paradigm. The program is not a description of any computation, it may be rather seen as a description of a problem to solve. Answers of a given program (the *logic*) may be computed under various strategies (the *control*), the results depend solely on the former. This semantics of programs, based on logic, is called *declarative semantics*.

Programming language Prolog is a main implementation of logic programming. Its core, which may be called "pure Prolog", is an implementation of SLD-resolution under a fixed control. (SLD-resolution with Prolog selection rule is called LD-resolution.) For a given program P and query Q, Prolog computes those logical consequences of P which are instances of Q. If the computation is finite then, roughly speaking, all such consequences are computed.[1]

On the other hand, Prolog may be viewed without any reference to logic, as a programming language with a specific control flow, the terms as the data, and a certain kind of term matching as the main primitive operation. Such a view is even necessary when we deal with non logical features of full Prolog, like the built-ins dealing with input/output. Of course such *operational view* loses all the advantages of declarative programming.

[1] See e.g. [Apt97] for details. We omit the issue of unification without occur-check; it may lead to incorrect answers (which are not logical consequences of the program).

In the author's opinion, Prolog makes practical declarative programming possible. A program treated as a set of logical clauses is a logic program. The logic determines the answers of the program. At a lower level, the programmer can influence the control. This can be done by setting the order of program clauses and the order of premises within a clause (and by some additional Prolog constructs). Changing the control keeps the logic intact, and thus the program's answers are unchanged; the logic is separated from the control [Kow79]. What is changed is the way they are computed, for instance the computation may be made more efficient. In particular, an infinite computation may be changed into a finite one.

In some cases, programs need to contain some non-logical fragments, for instance for input-output. But the practice shows that Prolog makes possible building programs which are to a substantial extent declarative; in other words, a substantial part of such program is a logic program. Numerous examples are given in the textbooks, for instance [SS94]. For a more formal discussion of this issue see [Dra18].

It should be noted that the operational approach to Prolog programming is often overused. In such programs it is not the declarative semantics that matters. A typical example is the red cut [SS94] – a programming technique which is based on pruning the search space; the program has undesired logical consequences, which are however not computed due to the pruning. Understanding such program substantially depends on its operational semantics. And understanding the operational semantics is more difficult than that of the declarative semantics. In particular, examples are known where a certain choice of the initial query leads to unexpected results of a program employing the red cut [SS94, p. 202-203], [CM03, Chapter 4]. In seems that some Prolog textbooks over-use such style of programming (like [CM03,Bra12], at least in their earlier editions).

Debugging tools of Prolog. We begin with a terminological comment. Often the term "debugging" is related to locating errors in programs. However its meaning is wider; it also includes correcting errors. So a better term for locating errors is *diagnosis*. However this text still does not reject the first usage, as it is quite common.

Despite Prolog has been designed mainly as an implementation of logic programming, its debugging tools work solely in terms of the operational semantics. So all the advantages of declarative programming are lost when it comes to locating errors in a program. The Prolog debugger is basically a tracing tool. It communicates with the programmer only in terms of the operational semantics. She (the programmer) must abandon the convenient high abstraction level of the declarative semantics and think about her program in operational terms.

Declarative diagnosis. In principle, it is well known how to locate errors in logic programs declaratively, i.e. abstracting from the operational semantics (see e.g. [Dra16, Section 7] and the references therein). The approach is called *declarative diagnosis* (and was introduced under a name *algorithmic debugging* by Shapiro [Sha83]). Two kinds of errors of the declarative semantics of a program are dealt with: *incorrectness* – producing results which are wrong according to the specifica-

tion, and *incompleteness* – not producing results which are required by the specification. We learn about an error by encountering a *symptom* – a wrong or missing answer obtained at program testing. Given a symptom, an incorrectness (respectively incompleteness) diagnosis algorithm locates an error semi-automatically, asking the user queries about the specification.

Unfortunately, declarative diagnosis was not adapted in practice. No tools for it are included in current Prolog systems.

Intended model problem. A possibly main reason for lack of acceptance of declarative diagnosis was discussed in [Dra16, Section 7]. Namely, declarative diagnosis requires that the programmer exactly knows the relations to be defined by the program. Formally this means that the programmer knows the least Herbrand model of the intended program. (In other words, the least Herbrand model is the specification.) This requirement turns out to be unrealistic. For instance, in an insertion sort program we do not know how inserting an element into an unsorted list should be performed. This can be done in any way, as the algorithm inserts elements only into sorted lists. Moreover, this can be done differently in various versions of the program. See [Dra18] for a more realistic example.[2] We call this difficulty *intended model problem.*

Usually the programmer knows the intended least Herbrand model of her program only approximately. She has an *approximate specification*: she knows a certain superset S_{corr} and a certain subset S_{compl} of the intended model. The superset tells what may be computed, and the subset – what must be computed. Let us call the former, S_{corr}, the *specification for correctness* and the latter, S_{compl}, the *specification for completeness.* Thus the program should be correct with respect to the former specification and complete with respect to the latter: $S_{compl} \subseteq M_P \subseteq S_{corr}$ (where M_P is the least Herbrand model of the program). In our example, it is irrelevant how an element is inserted into an unsorted list; thus the specification for correctness would include all such possible insertions (and the specification for completeness would include none).

Now it is obvious that when diagnosing incorrectness the programmer should use the specification of correctness instead of the intended model, and the specification of completeness should be used when diagnosing incompleteness [Dra16]. The author believes that this approach can make declarative diagnosis useful in practice.

Intended model problem was possibly first noticed by Pereira [Per86]. He introduced the notion of *inadmissible* atomic queries. A formal definition is not given.[3] We may suppose that ground inadmissible atoms are those from $S_{corr} \setminus S_{compl}$. Generally, this notion seems operational; an inadmissible atom seems to be one that should not appear as a selected atom in an SLD-tree of the program.

[2] In the main example of [Dra18], the semantics of a particular predicate differs at various steps of program development.

[3] "a goal is admissible if it complies with the intended use of the procedure for it – i.e. it has the correct argument types – irrespective of whether the goal succeeds or not" (p. 6 of the extended version of [Per86]).

Naish [Nai00] proposed a 3-valued diagnosis scheme. The third value, *inadmissible*, is related to the search space of a diagnosis algorithm, and to its queries. The form of queries depends on the particular algorithm, e.g. it may be an atom together with its computed answers. So the third value is not (directly) related to the declarative semantics of programs. It turns out that applying the scheme to incorrectness diagnosis ([Nai00, Section 5.1]) boils down to standard diagnosis w.r.t. S_{corr}, and applying it to incompleteness diagnosis ([Nai00, Section 5.2]) – to the standard diagnosis w.r.t. S_{compl} (where S_{compl} is the set of correct atoms, and S_{corr} is the union of S_{compl} and the set of inadmissible atoms).

This paper. The role of this paper is to find if, how, and to which extent the Prolog debugger can be used as a tool for declarative logic programming. We focus on the debugger of SICStus Prolog. We omit its advanced debugging features, which are sophisticated, but seem not easy to learn and not known by most of programmers.

The paper is organized as follows. The next section deals with the Prolog debugger and the information it can provide. Section 3 discusses applying the debugger for diagnosing incorrectness and incompleteness. The last section contains conclusions.

2 Prolog debugger

In this section we present the Prolog debugger and try to find out how to use it to obtain the information necessary from the point of view of declarative programming. First we relate the computation model used by the debugger to the standard operational semantics (LD-resolution). We also formalize the information needed for incorrectness and incompleteness diagnoses. For incorrectness diagnosis, given an atomic answer A we need to know which clause $H \leftarrow B_1, \ldots, B_n$ have been used to obtain the answer A (A is an instance of H), and which top-level atomic answers (instances of $B_1, \ldots, B_n$) have been involved. For incompleteness diagnosis, the related information is which answers have been computed for each selected instance of each body atom B_i of each clause $H \leftarrow B_1, \ldots, B_n$ resolved with a given atomic query A. In Section 2.2 we describe the messages of the debugger. Section 2.3 investigates how to extract from the debugger's output the information of interest.

2.1 Byrd box model and LD-resolution

The debugger refers to the operational semantics of Prolog in terms of a "Byrd box model". Roughly speaking, the model assigns four ports to each atom selected in LD-resolution. From a programmer's point of view such atom can be called a procedure call. The model is usually easily understood by programmers. However it will be useful to relate it here to LD-resolution, and to introduce some additional notions. In this paper, we often skip "LD-" and by "derivation" we mean "LD-derivation" (unless stated otherwise).

Structuring LD-derivations. Let us consider a (finite or infinite) LD-derivation D with queries $Q_0, Q_1, Q_2 \ldots$, the input clauses $C_1, C_2, \ldots$, and the mgu's $\theta_1, \theta_2, \ldots$. By a **procedure call** of D we mean the atom selected in a query of D. Following [DM88,Dra17], we describe a fragment of D which may be viewed as the evaluation of a given procedure call A.

Definition 1. Consider a query $Q_{k-1} = A, B_1, \ldots, B_m$ $(m \geq 0)$ in a derivation D as above. If D contains a query $Q_l = (B_1, \ldots, B_m)\theta_k \cdots \theta_l$, $k \leq l$, then the call A (of Q_{k-1}) **succeeds** in D.

In such case, by the **subderivation** for A (of Q_{k-1} in D) we mean the fragment of D consisting of the queries Q_i where $k-1 \leq i \leq l$, and for $k-1 \leq i < l$ each Q_i contains more than m atoms.[4] We call such subderivation **successful**. The (computed) **answer** for A (of Q_{k-1} in D) is $A\theta_k \cdots \theta_l$.

If A (of Q_{k-1}) does not succeed in D then the **subderivation** for A (of Q_{k-1} in D) is the fragment of D consisting of the queries Q_i where $k-1 \leq i$.

By a *subderivation* (respectively an *answer*) for A of Q in an LD-tree $\mathcal{T}$ we mean a subderivation (answer) for A of Q in a branch D of $\mathcal{T}$.

Now we structure a subderivation D for an atom A by distinguishing in D top-level procedure calls. Assume A is resolved with a clause $H \leftarrow A_1, \ldots, A_n$ in the first step of D. If then an instance of A_i becomes a procedure call, we call it a top-level call. More precisely:

Definition 2. Consider a subderivation D for A. Its first two queries are $Q_{k-1} = A, Q'$, $Q_k = (A_1, \ldots, A_n, Q')\theta_k$, and $A_1, \ldots, A_n$ is the body of the clause used in the first step of the subderivation. Assume $n > 0$. Let $|Q_k|$ be the length of Q_k (this means the number of atoms in Q_k).

Consider an index j, $1 \leq j \leq n$. If there exists in D a query of the length $|Q_k| + 1 - j$ and $Q_{i_j} = (A_j, \ldots, A_n, Q')\theta_k \cdots \theta_{i_j}$ is the first such query then we say that $A_j\theta_k \cdots \theta_{i_j}$ (of Q_{i_j}) is a **top-level call** of D, and the subderivation D' for $A_j\theta_k \cdots \theta_{i_j}$ (of Q_{i_j}) in D is a **top-level subderivation** of D.

A top-level call of a subderivation D for A will be also called a top-level call for A.

Notice that if A is resolved with a unary clause ($n = 0$, and D consists of two queries) then D has no top-level subderivations. Also, the last query of a top-level subderivation of D is the first query of the next subderivation, or it is the last query of D.

We are ready to describe what information to obtain from the debugger in order to facilitate incorrectness and incompleteness diagnosis. First we describe which top-level answers correspond to an answer for A; we may say that they have been used to obtain the answer for A.

[4] Thus each such Q_i is of the form $A_1, \ldots, A_{m_i}, (B_1, \ldots, B_m)\theta_k \cdots \theta_i$ where $m_i > 0$. This implies that the least $l > k$ is taken such that Q_l is of the form $(B_1, \ldots, B_m)\theta_k \cdots \theta_l$.

Definition 3. If subderivation D for A as in Def. 2 is successful then it has n top-level subderivations, for atoms $A_j\theta_k \cdots \theta_{i_j}$ $(j = 1, \ldots, n)$. Their answers in D are, respectively, $A'_j = A_j\theta_k \cdots \theta_{i_{j+1}}$ (where i_{n+1} is the index of the last query $Q_{i_{n+1}}$ of D). In such case, by the **top-level success trace** for A (in D) we mean the sequence $A'_1, \ldots, A'_n$ of the answers.

Top-level success traces will be employed in incorrectness diagnosis. For diagnosing incompleteness, we need to collect all the answers for each top-level call.

Definition 4. Consider an LD-tree $\mathcal{T}$ with a node Q. Let A be the first atom of Q. By the **top-level search trace** (or simply *top-level trace*) for A (of Q in $\mathcal{T}$) we mean the set of pairs

$$\left\{ (B, \{B_1, \ldots, B_k\}) \;\middle|\; \begin{array}{l} B \text{ is the first atom of a node } Q' \text{ of } \mathcal{T}, \\ Q' \text{ occurs in a subderivation } D' \text{ for } A \text{ of } Q \text{ in } \mathcal{T}, \\ B \text{ is a top-level call of } D', \\ B_1, \ldots, B_k \text{ are the answers for } B \text{ of } Q' \text{ in } \mathcal{T} \end{array} \right\}.$$

2.2 Debugger output.

For the purposes of this paper, this section should provide a sufficient description of the debugger. We focus on the debugger of SICStus. For an introduction and further information about the Prolog debugger see e.g. the textbook [CM03] or the manual `http://sicstus.sics.se/`.

Prolog computation can be seen as traversal of an LD-tree. The Prolog debugger reports the current state of the traversal by displaying one line items, such an item contains a single atom augmented by other information. A procedure call A is reported as an item

$$n \quad d \ \texttt{Call:}\ A$$

and a corresponding answer $A' = A\theta_k \cdots \theta_i$ as

$$n \quad d \ \texttt{Exit:}\ A'$$

Here n, d are, respectively, the unique invocation number and the current depth of the invocation; we skip the details. What is important is that, given an Exit item, the invocation number uniquely determines the corresponding Call item.

Note that a node in an LD-tree may be visited many times, and usually more than one item correspond to a single visit. For instance, to the last node Q_l of a successful subderivation (from Definition 1) there correspond, at least, an Exit item with atom $A\theta_k \cdots \theta_l$ and a Call item with atom $B_1\theta_k \cdots \theta_l$ (provided $m > 0$). Note that such a node is often the last query of more than one successful subderivations (cf. Def. 2). In such case other Exit items correspond to Q_l. They are displayed in the order which may be described as leaving nested procedure calls. More formally, the order of displaying the Exit items is that of the increasing lengths of the corresponding successful subderivations. (The displayed invocation depths of these items are decreasing consecutive natural numbers.)

An Exit item is preceded by `?` when backtrack-points exist between the corresponding Call and the given Exit. Thus more answers are possible for (the atom of) this Call.

At backtracking the debugger displays Redo items of the form

$$n \quad d \texttt{ Redo: } A'$$

Such item corresponds to an Exit item with the same numbers n, d and atom A'. Both items correspond to the same node of the LD-tree. The Redo item appears, speaking informally, when the answer A' is abandoned, and the computation of a new answer for the same query begins. SICStus produces a Redo item only when the corresponding Exit item was preceded by `?`.

A Fail item

$$n \quad d \texttt{ Fail: } A$$

is displayed at backtracking. It means that a node with A selected is being left (and will not be visited anymore). The numbers and the atom in a Fail item are the same as those in the corresponding Call item. Both the Call and Fail items correspond to the same node of the LD-tree.

We described the output of the debugger of SICStus. The debuggers of most Prolog systems are similar. However important differences happen. For instance the debugger of SWI-Prolog (`http://swi-prolog.org/`) does not display the invocation number, and the approach presented below is inapplicable to it. On the other hand, the debuggers or Ciao (`http://ciao-lang.org/`) and Yap (`https://www.dcc.fc.up.pt/ vsc/yap/`) seem to display such numbers.

2.3 Obtaining top-level traces

We are ready to describe how to obtain top-level traces using the Prolog debugger. We first deal with the search trace.

Algorithm 1 (Top-level trace) Assume that we are at a Call port; the debugger displays

$$n \quad d \texttt{ Call: } A$$

We show a way of obtaining the top-level search trace for A.

1. Type enter to make one step of computation.
   (There are three possibilities. If the result is an item n_1 $d{+}1$ `Call:` B_1 then B_1 is an instance of the first atom of the body of the clause used in the resolution step. Obtaining $n\ d$ `Exit:` A' means that a unary clause was used and A succeeded immediately. Obtaining $n\ d$ `Fail:` A means that A failed immediately, as it was not unifiable with any clause head.)
2. Now repetitively do the following.
   (a) If item

   $$n \quad d \texttt{ Fail: } A$$

   is displayed then the search is completed. The trace is to be extracted from the output that the debugger has produced.

(b) If item

$$n_i \quad d{+}1 \texttt{ Call: } B_i \qquad \text{or} \qquad n_i \quad d{+}1 \texttt{ Redo: } B_i$$

is displayed then type **s**, to go to the corresponding `Exit` or `Fail` port.

(c) If

$$n_i \quad d{+}1 \texttt{ Exit: } B_i \qquad \text{or} \qquad n_i \quad d{+}1 \texttt{ Fail: } B_i$$

is displayed then type enter, to make a single step.

(d) Obtaining an item

$$n \quad d \texttt{ Exit: } A'$$

means that an answer A' for A has been produced. In other words, a successful subderivation for A has been constructed. Unfortunately, the Prolog debugger does not directly support continuing the search for further answers for A. We can do this as follows.

- If the item $n\ d$ `Exit:` A' is obtained by applying a unary clause to A (i.e. obtained immediately by typing enter at $n\ d$ `Call:` A or at $n\ d$ `Redo:` A) then
  i. If the item $n\ d$ `Exit:` A' is not preceded by `?` then the search is completed (similarly as in case 2a).
  ii. Otherwise the item is a backtrack point, perform the backtracking by issuing the command (or rather the command sequence) **jr** n. Item $n\ d$ `Redo:` A' is displayed. Type enter, as in step 1.
- Otherwise (a non-unary clause has been used) identify in the printed items the top-level success trace for A, as described below.
  Find the last item of the trace that is preceded by `?`, assume it is

$$\texttt{?} \qquad n_j \quad d{+}1 \texttt{ Exit: } B_j$$

This is the backtrack-point to which backtracking from $n\ d$ `Exit:` A' should go.
  iii. If there is no such item then the search is completed (similarly as in case 2a).
  iv. Otherwise perform the backtracking by issuing the command **jr** n_j. Item $n_j\ d{+}1$ `Redo:` B_j is displayed. Type enter, as in step 1.

3. Now the top-level trace has to be extracted from the information printed by the debugger in stage 2 above. A call and its corresponding answers have the same unique invocation number n_j. Thus the calls may be paired with their answers by means of sorting the debugger output.
   In a particular case of using SICStus under Emacs, the sorting can be done by running a shell command `cut -b 2- | sort -nk 1 | egrep 'Call:|Exit:'` on the debugger output in buffer `*prolog*`, using Emacs command `shell-command-on-region`. This removes from each line the first character (space or `?`), sorts the items according to the first number (which is the invocation number) and selects Call and Exit items.
   The result of sorting provides the top-level trace for A in a readable form.

An alternative version of step 3 of the algorithm is running Prolog for each query from the Call items of the debugger output.

Algorithm 2 (Top-level success trace) Assume that we obtained an Exit item containing an answer A'. The item corresponds to the last query of a successful subderivation D for an atom A. In order to extract from the debugger output the top-level trace for D, we need that the debugger has displayed the Call and Exit items containing the top-level calls of D and the corresponding answers. If this is not the case then, at the n d `Exit:` A' item, type **r** to arrive to the corresponding Call item. Then start constructing top-level search trace, until arriving again to the Exit item.

This may be made more efficient, by re-starting the computation with A' the initial query. Then the search space to obtain a success of A' (and the corresponding top-level trace) may be substantially smaller than that for original atomic query from the Call item. Additionally, case 2d of Algorithm 1 never needs to be performed.

To select a top-level success trace from the printed debugger items, do repetitively the following. The trace will be constructed backwards. Initially the current item is n d `Exit:` A'.

The current item is

$$n \quad d \texttt{ Exit: } A' \qquad \text{or} \qquad n_j \quad d{+}1 \texttt{ Call: } B_j$$

Consider the preceding item.
If the immediately preceding item is

$$n_{j'} \quad d{+}1 \texttt{ Exit: } B'_{j'}$$

then $B'_{j'}$ is obtained as an element of the success trace. Find the corresponding

$$n_{j'} \quad d{+}1 \texttt{ Call: } B_{j'}$$

item, and make it the current item
Otherwise, the preceding item is

$$n \quad d \texttt{ Call: } A$$

and all the elements of the success trace have been found.

3 Diagnosis

This section discusses first diagnosis of incorrectness, and then that of incompleteness. In each case we first present the diagnosis itself, and then discuss how it may be performed employing the Prolog debugger.

3.1 Diagnosing incorrectness.

A *symptom* of incorrectness is an incorrect answer of the program. More formally, consider a program P and a Herbrand interpretation S_{corr}, which is our specification for correctness. A symptom is an answer[5] Q such that $S_{corr} \not\models Q$, where S_{corr} is the specification for correctness. (In other words, Q has a ground instance $Q\theta$ such that $Q \notin S_{corr}$.) When testing finds such a symptom, the role of diagnosis is to find the error, this means the reason of incorrectness. An error is a clause of the program which out of correct (w.r.t. S_{corr}) premises produces an incorrect conclusion. More precisely:

Definition 5. Given a definite program P and a specification S_{corr} (for correctness), an **incorrectness error** is an instance

$$H \leftarrow B_1, \ldots, B_n \qquad (n \geq 0)$$

of a clause of P such that $S_{corr} \models B_i$ for all $i = 1, \ldots, n$, but $S_{corr} \not\models H$.

An *incorrect clause* is a clause C having an instance $C\theta$ which is an incorrectness error.

In other words, C is an incorrect clause iff $S_{corr} \not\models C$. In what follows, by a *correct atom* we consider an atom A such that $S_{corr} \models A$ (where S_{corr} is the considered specification for correctness).

Note that we cannot formally establish which part of the clause is erroneous. Easy examples can be constructed showing that an incorrect clause C can be corrected in various ways; and each atom of C remains unchanged in some corrected version of C [Dra16, Section 7.1].

The incorrectness diagnosis algorithm is based on the notion of a proof tree, called also implication tree.

Definition 6. Let P be a definite program and Q an atomic query. A **proof tree** for P and Q is a finite tree in which the nodes are atoms, the root is Q and

$$\text{if} \quad \begin{array}{l} B_1, \ldots, B_n \text{ are the} \\ \text{children of a node } B \end{array} \quad \text{then} \quad \begin{array}{l} B \leftarrow B_1, \ldots, B_n \text{ is an} \\ \text{instance of a clause of } P \end{array} \qquad (n \geq 0).$$

Note that the leaves of a proof tree are instances of unary clauses of P.

Now diagnosing incorrectness is rather obvious. If an atom Q is (an instance of) an answer of P then there exists a proof tree for P, Q. The tree must contain an incorrectness error (otherwise the root of the tree is correct, i.e. $S_{corr} \models Q$). A natural way of searching for the error, in other words an incorrectness diagnosis algorithm, is as follows: Begin from the root and, recursively, check the children $B_1, \ldots, B_n$ of the current node whether they are correct (formally, whether $S_{corr} \models B_i$). If all of them are correct, the error is found. Otherwise take an incorrect child B_i, and continue the search taking B_i as the current node.

Obviously, such search locates a single error. So correcting the error does not guarantee correctness of the program.[6]

[5] An answer in the sense used here is called a "correct instance of a query" in [Apt97]. In other words, it is the result of applying a correct answer substitution to a query.

[6] This does not even guarantee that the symptom we began with would disappear – there may be some other errors involved.

3.2 Prolog debugger and incorrectness

Now we try to find out to which extent the algorithm described above can be mimicked by the standard Prolog debugger. Unfortunately, the debugger does not provide a way to construct a proof tree for a given answer. We can however employ top-level success traces to perform a search similar to that done by the incorrectness diagnosing algorithm described in Section 3.1.

A strategy for incorrectness errors Here we describe how to locate incorrectness errors using the Prolog debugger.

Algorithm 3 Assume that while tracing the program we found out an incorrect answer A' (for a query A). So we are at an Exit item containing A'. Type **r** to arrive to the corresponding Call item n d `Call:` A. Do repetitively the following:

1. Construct the top-level success trace $B'_1, \ldots, B'_m$ for the subderivation D (for an atom A, where A' is the answer for A in D), as described in Algorithm 2.
2. Check whether the atoms of the trace are correct (formally, whether $S_{corr} \models B'_i$). If all of them are, then the search ends. Otherwise take an item n_i $d{+}1$ `Exit:` B'_i containing an incorrect B'_i. Type a command sequence **jc** n_i to arrive to the corresponding Call item n_i $d{+}1$ `Call:` B. Now repeat the search (with A, A' replaced by, respectively, B, B'_i).

The last obtained top-level success trace $B'_1, \ldots, B'_m$ points out the erroneous clause of the program. The clause is $C = H \leftarrow B_1, \ldots, B_m$ such that the obtained answers are instances of the body atoms of C: each B'_j is an instance of B_j, for $j = 1, \ldots, m$. The head H of C is unifiable with the last call B for which the top-level success trace was built. Formally, C has an instance which is an incorrectness error (cf. Def. 5).

Obviously, the algorithm can be improved by checking the correctness of each element B'_i of the trace as soon as it is located. (So the trace needs to be constructed only until an incorrect element is found.)

The approach of Algorithm 3 is rather tedious. A more natural way to locate incorrectness errors is as follows.

Algorithm 4

1. Assume, as above, that an incorrect answer A' was found. Begin as above, by arriving to the call A that resulted in the incorrect answer, and starting constructing a top-level search trace.
2. For each obtained item n_i $d{+}1$ `Exit:` B' check if B' is correct.
3. If B' is an incorrect answer, then restart the search from B'.
4. If no incorrect answer has appeared until arriving to the incorrect answer A' then the error is found. It is the last clause C whose head was unified with A in the computation. (Formally, an instance of C is an incorrectness error.) The clause may be identified, as previously, by extracting the top-level success trace (for the subderivation that produced A').

Comments

In Algorithm 4, it is often not necessary to know the (whole) top-level success trace to identify the erroneous clause in the program. In many cases, knowing the last one or two answers of the trace is sufficient. For instance, let n' d' `Call:` B be the last call for which top-level trace was inspected. The last item displayed by the debugger is n' d' `Exit:` B' (where B' is incorrect). Assume that the previous item is n_j $d'+1$ `Exit:` B'_j. Then the top-level trace of interest is not empty, B'_j is its last atom and is an instance of the last body atom of an erroneous clause. If the program has only one such clause, then finding the rest of the top-level success trace is unnecessary.

The error located by the second approach (Algorithm 4) may be not the one that caused the initial incorrect answer A'. This is because the search may go into a branch of the LD-tree distinct from the branch in which A' is produced. Anyway, an actual error has been discovered in the program. This outcome is useful, as each error in the program should be corrected.

Note that the approach is complete, in the sense that the error(s) responsible for A' can be found. This is due to the nondeterministic search performed by the algorithm. The error(s) will be located under some choice of incorrect answers in the top-level search traces.

The search may be made more efficient if, instead of tracing the original computation, we re-start it with an incorrect answer as a query. The corresponding modification (of both algorithms) is as follows. Whenever an incorrect answer B' is identified, instead of continuing the search for the corresponding call B, one interrupts the debugger session and begins a new one by starting Prolog with query B'. The query will succeed with B' (i.e. itself) as an answer, but the size of the trace may be substantially smaller (and is never greater).

The Prolog debugger does not facilitate searching for the reason of incorrectness. Finding a top-level success trace is tedious and far from obvious. In particular, there seems to be no way of skipping the backtracking that precedes obtaining the wrong answer. The abilities of the debugger make Algorithm 4 preferable; this approach in a more straightforward way uses what is offered by the debugger.

Looking for the reason of an incorrect answer is a basic task. It is strange that such a task is not conveniently facilitated by the available debugging tools.

3.3 Diagnosing incompleteness

A specification for completeness is, as already stated, a Herbrand interpretation which is the set of all required ground answers of the program. A symptom of incompleteness is lack of some answers of the program. More formally, given a program P and a specification S_{compl}, by an incompleteness symptom we may consider an atom A such that $S_{compl} \models A$ but $P \not\models A$. As a symptom is to be obtained out of an actual computation, we additionally require that the LD-tree for A is finite. We will consider a more general version:

Definition 7. Consider a definite program P and a specification S_{compl} (for completeness). Let A be an atomic query for which an LD-tree is finite and let $A\theta_1, \ldots, A\theta_n$ be the computed answers for A from the tree. If there exists an instance $A\sigma \in S_{compl}$ such that $A\sigma$ is not an instance of any $A\theta_i$ $(i = 1, \ldots, n)$ then $A, A\theta_1, \ldots, A\theta_n$ is an **incompleteness symptom** (for P w.r.t. S_{compl}).

We will often skip the sequence of answers, and say that A alone is the symptom. The definition can be generalized to non-atomic queries in an obvious way.

Definition 8. Let P be a definite program, and S_{compl} a specification. A ground atom A is **covered** by a clause C w.r.t. S_{compl} if there exists a ground instance $A \leftarrow B_1, \ldots, B_n$ of C $(n \geq 0)$ such that all the atoms $B_1, \ldots, B_n$ are in S_{compl}.

A is *covered by the program* P (w.r.t. S_{compl}) if A is covered by some clause $C \in P$.

Informally, A is covered by P if it can be produced by a rule from P out of some atoms from the specification.

For any program P incomplete w.r.t. S_{compl} there exists an atom $A \in S_{compl}$ uncovered by P w.r.t. S_{compl} [Dra16]. Such an uncovered atom $p(\boldsymbol{t})$ locates the error in P. This is because no rule of P can produce $p(\boldsymbol{t})$ out of atoms required to be produced. This shows that the procedure p (the set of clauses beginning with p) is the reason of the incompleteness and has to be modified, to make the program complete. Note that similarly to the incorrectness case, we cannot locate the error more precisely. Various clauses may be modified to make $p(\boldsymbol{t})$ covered, or a new clause may be added. An extreme case is adding to P a fact $p(\boldsymbol{t})$.

Incompleteness diagnosis means looking for an uncovered atom, or – more generally – for an atom with an instance which is uncovered: Such atom localizes the procedure of the program which is responsible for incompleteness.

Definition 9. Let P be a definite program, and S_{compl} a specification. An **incompleteness error** (for P w.r.t. S_{compl}) is an atom that has an instance which is not covered (by P w.r.t. S_{compl}).

Name "incompleteness error" may seem unnatural, but we find it convenient.

A class of incompleteness diagnosis algorithms employs the following idea. Start with an atomic query A (which is a symptom) and construct a top level trace for it. Inspect the trace, whether it contains a symptom B. If so then invoke the search recursively with B. Otherwise the error is located; it is some instance of A, and we located the procedure of the program which is the reason of the error. Such approach (see e.g. [Per86,DNTM89]) is sometimes called *Pereira-style* incompleteness diagnosis [Nai92].

3.4 Prolog debugger and incompleteness

We show how Pereira-style diagnosis may be performed with help of the Prolog debugger.

Algorithm 5 (Incompleteness diagnosis) Begin with a symptom A. Obtain the top-level search trace for A. In the trace, check if the atom B from a Call item together with the answers $B_1, \ldots, B_n$ from the corresponding Exit items is an incompleteness symptom. If yes, invoke the same search starting from B. If the answer is no for all Call items of the trace, the search is ended as we located A as an incompleteness error.

Comments

Standard comments about incompleteness diagnosis apply here. To decrease the search space, it is useful to start the diagnosis from a ground instance $A\theta \notin S_{compl}$ of the symptom A (instead of A itself). The same for each symptom B found during the search – re-start the computation and the diagnosis from an appropriate instance of B.

Often an incorrectness error coincides with an incompleteness error – a wrong answer is produced instead of a correct one. The programmer learns about this when facing an incorrect answer B_i (appearing in a top-level trace). A standard advice in such case [DNTM89,Nai92] is to switch to incorrectness diagnosis. This is because incorrectness diagnosis is simpler, and it locates an error down to a program clause (not to a whole procedure, as incompleteness diagnosis does). The gain of such switch is less obvious in our case, since the effort needed for incorrectness diagnosis (Algorithm 4) seems not smaller than that for incompleteness (Algorithm 5).

4 Conclusions

Prolog makes declarative logic programming possible – programs may be written and reasoned about in terms of their declarative semantics, to a substantial extent abstracting from the operational semantics. This advantage is lost when it comes to locating errors in programs, as the Prolog debugger works solely in terms of the operational semantics. This paper is an attempt to study if and how the Prolog debugger can be used for declarative programming. It presents how the debugger can be used to perform incorrectness and incompleteness diagnosis. The debugger used is that of SICStus; the presented approach is inapplicable to the debugger of SWI-Prolog, as the latter does not display unique invocation numbers.

We may informally present the underlying idea of this paper in a different way: To understand what program execution can tell us about the declarative semantics of the program, we need to be able to obtain the following information. 1. For a given atomic answer A, what are the top-level answers that have lead to A?[7] 2. For a given atomic query Q, and for each top-level atomic query B in the computation for Q, what are all the answers for B?

It turns out that the information needed to declaratively locate errors in logic programs is difficult and tedious to obtain by means of a standard Prolog

[7] The top-level answers are instances of the body atoms $B_1, \ldots, B_n$, of the clause $H \leftarrow B_1, \ldots, B_n$ that was used at the first LD-resolution step in computing the answer A. This is formalized in Section 2.1 as top-level success trace.

debugger. In the author's opinion, this is a substantial drawback for employing declarative logic programming in practice.

This drawback particularly concerns incorrectness diagnosis. Additionally, debugging of incorrectness seems more important than that of incompleteness. This is because incompleteness is often caused by producing incorrect answers instead of correct ones. Also, incorrectness diagnosis is more precise, as it locates the erroneous fragment of the program more precisely than incompleteness diagnosis. Hence the first thing to do in order to facilitate more declarative debugging is to implement a tool supporting incorrectness diagnosis. The author supposes that it does not need to be a full implementation of an incorrectness diagnosis algorithm. It should be sufficient to provide a tool for convenient browsing of a proof tree (which provides an abstraction of the part of computation responsible for the considered incorrect answer).

References

Apt97. K. R. Apt. *From Logic Programming to Prolog.* International Series in Computer Science. Prentice-Hall, 1997.

Bra12. Ivan Bratko. *Prolog Programming for Artificial Intelligence, 4th Edition.* Addison-Wesley, 2012.

CM03. W.F. Clocksin and C.S. Mellish. *Programming in Prolog: using the ISO standard.* Springer, 5 edition, 2003.

DM88. W. Drabent and J. Małuszyński. Inductive assertion method for logic programs. *Theoretical Computer Science*, 59:133–155, 1988.

DNTM89. W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. Algorithmic Debugging with Assertions. In H. Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 501–522. The MIT Press, 1989.

Dra16. W. Drabent. Correctness and completeness of logic programs. *ACM Trans. Comput. Log.*, 17(3):18:1–18:32, 2016.

Dra17. W. Drabent. Proving completeness of logic programs with the cut. *Formal Aspects of Computing*, 29(1):155–172, 2017.

Dra18. W. Drabent. Logic + control: On program construction and verification. *Theory and Practice of Logic Programming*, 18(1):1–29, 2018.

Kow79. Robert A. Kowalski. Algorithm = logic + control. *Commun. ACM*, 22(7):424–436, 1979.

Nai92. L. Naish. Declarative diagnosis of missing answers. *New Generation Comput.*, 10(3):255–286, 1992.

Nai00. L. Naish. A three-valued declarative debugging scheme. In *23rd Australasian Computer Science Conference (ACSC 2000)*, pages 166–173. IEEE Computer Society, 2000.

Per86. L. M. Pereira. Rational debugging in logic programming. In E. Y. Shapiro, editor, *ICLP*, volume 225 of *Lecture Notes in Computer Science*, pages 203–210. Springer, 1986. Extended version at `https://userweb.fct.unl.pt/~lmp/`.

Sha83. E. Shapiro. *Algorithmic Program Debugging.* The MIT Press, 1983.

SS94. L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 2 edition, 1994.

An Optional Static Type System for Prolog

Isabel Wingen, Philipp Körner ✉[0000−0001−7256−9560]

Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf, Germany
{isabel.wingen,p.koerner}@uni-duesseldorf.de

Abstract. Benefits of static type systems are well-known: typically, they offer guarantees that no type error will occur during runtime and, inherently, inferred types serve as documentation on how functions are called. On the other hand, many type systems have to limit expressiveness of the language because, in general, it is undecidable whether a given program is correct regarding types. Another concern that was not addressed so far is that, for logic programming languages such as Prolog, it is impossible to distinguish between intended and unintended failure and, worse, intended and unintended success without additional annotations.
In this paper, we elaborate on and discuss the aforementioned issues. As an alternative, we present a static type analysis which is based on *plspec*. Instead of ensuring full type-safety, we aim to statically identify type errors on a best-effort basis without limiting the expressiveness of Prolog programs. Finally, we evaluate our approach on real-world code featured in the SWI community packages and a large project implementing a model checker.

Keywords: Prolog, static verification, optional type system, data specification

1 Introduction

Dynamic type systems often enable type errors during development. Generally, this is not too much of an issue as errors usually get caught early by test cases or REPL-driven development. Prolog programs however do not follow patterns prevalent in other programming paradigms. Exception are thrown rarely and execution is resumed at some prior point via backtracking instead, before queries ultimately fail. This renders it cumbersome to identify type errors, their location and when they occur.

There has been broad research on type systems offering a guarantee about the absence of type errors. Yet, in dynamic programming languages such as Prolog, a complete well-typing of arbitrary programs is undecidable [14]. Thus, in order for the type system to work, the expressiveness of the language often is limited. This hinders adaptation to existing code severely, and, as a consequence, type errors are often ignored in larger projects.

This paper contributes the following:

- A type analysis tool that can be used for *any* Prolog program without modification. It can handle a proper “any” type and can easily be extended for any Prolog dialect.
- An empirical evaluation of the amount of inferred types during type analysis.
- Automatic inference and generation of pre- and postconditions.

2 A Note on Type Systems and Related Work

Static type systems have a huge success story, mostly in functional programming languages like Haskell [6], but also in some Prolog derivatives, such as Mercury [4]. Even similar dynamic languages such as Erlang include a type specification language [5]. Many static type systems for logic programming languages have been presented [13], including the seminal works of [12], which also influenced Typed Prolog [8], and a pluggable type system for Yap and SWI-Prolog [16].

All type systems have some common foundations, yet usually vary in expressiveness. Some type systems suggest type annotations for functions or predicates, some require annotations of all predicates or those of which the type cannot be inferred automatically to a satisfactory level. Yet, type checking of logic programs is, in general, undecidable [14]. This renders only three feasible ways to deal with typing:

1. Allow only a subset of types, for which typing is decidable, e.g., regular types [2] or even only mode annotations [15].
2. Require annotations where typing is not decidable without additional information.
3. Work on a best-effort basis which may let some type errors slip through.

Most type systems fall into the first or the second category. Yet, this usually limits how programs can be written: some efficient or idiomatic patterns may be rejected by the type system. As an example, most implementations of the Hindley-Milner type system [11] do not allow heterogeneous lists. Additionally, most type system refuse to handle a proper "any" type, where not enough information is available and arguments may, statically, be any arbitrary value. Such restrictions render adaptation of type systems to existing projects infeasible. Annotations, however, can be used to guide type systems and allow more precise typing. The trade-off is code overhead introduced by the annotations themselves, which are often cumbersome to write and to maintain.

Into the last category falls the work of Schrijvers et al. [16], and, more well-known, the seminal work of Ciao Prolog [3] featuring a rich assertion language which can be used to describe types. Unfortunately, [16] seems to be abandoned after an early publication and the official release was removed. Ciao's approach, on the other hand, is very powerful, yet is incompatible with other Prolog dialects.

We strongly agree with the reasoning and philosophy behind Ciao stated in [3]: type systems for languages such as Prolog must be optional in order retain the usefulness, power and expressiveness of the language. Mycroft-O'Keefe identified two typical mistakes that can be uncovered: firstly, omitted cases and, secondly, transposed arguments. We argue that omitted cases might as well be intended failure and, as such, should not be covered by a type system at all. Additionally, traditional type systems such as the seminal work of Mycroft-O'Keefe [12] often are not a good fit, as typing in Prolog is a curious case: due to backtracking and goal failure, type errors may lead to behaviour that is valid, yet unintended.

Backtracking Prolog predicates are allowed to offer multiple solutions which is often referred to as non-determinism. Once a goal fails, execution continues at the last choice point where another solution might be possible. Thus, if a predicate was called incorrectly, the program might still continue because another solution is found, e.g., based

on other input. Consider an error in a specialised algorithm: if there is a choice point, a solution might still be found if another, slower, fall-back implementation is invoked via backtracking. Such errors could go unnoticed for a long time as they cannot be uncovered by testing if a correct solution is still found in a less efficient manner.

Goal Failure Most ISO Prolog predicates throw an exception if they are called with incorrect types. However, non-ISO predicates (such libraries as code written by programmers) usually fail as no solution is found because the input does not match with any clause. E.g., consider a predicate as trivial as `member`:

```
member(H, [H|_]).      member(E, [_|T]) :- member(E, T).
```

Querying `member(1, [2,3,4])` will fail because *the first argument is not in the list*, which is the second argument. We name this *intended failure*. Yet, if the second argument is not a list, e.g., when called as `member(1, 2)`, it will fail because *the second argument is not a list*. We call this *unintended failure*, as the predicate is called *incorrectly*. The story gets even worse: additionally to failure cases, there can also be unintended *success*. Calling `member(2, [1, 2|foo])` is not intended to succeed, as the second argument is not a list, yet the query returns successfully. Distinguishing between intended and unintended behaviour is impossible as they use the same signal, i.e. goal failure (or success). We argue that the only proper behaviour would be to raise an error on unintended input instead because this most likely is a programming error.

In this paper, we investigate the following questions: Can we implement an optional type system that supports *any Prolog dialect*? How well does such a type system perform and is a subset of errors that are identified on *best-effort basis* sufficient? We think that the most relevant class of errors is that an argument is passed incorrectly, i.e. the type is wrong. Thus, an important question is how precise type inference by such a type system could be. If it works well enough, popular error classes such as transposed arguments, as described by [12], can be identified in most cases.

For this, we build on top of *plspec* [7] which offers some type annotations that can be instrumented as runtime checks. Other libraries that provide annotations as presented, e.g. in [3,16], can be supported by including minor syntactic transformations as *plspec*'s feature set is very similar overall. In the following, the library and its annotations are presented, as they form the foundations for a static type analysis on top.

3 Foundation: *plspec*

plspec is an ad-hoc type system that executes type checks at runtime via co-routining. With *plspec*, it is possible to add two kinds of annotations. The first kind of annotation allows introduction of new types. *plspec* offers three different ways for this. For our type system, we currently focus only on the first one and implement shipped special cases that fall under the third category, i.e. tuples, lists and compound terms:

1. recombination of existing types
2. providing a predicate that acts as characteristic function
3. rules to check part of a term and generate new specifications for sub-terms

plspec's built-in types are shown in Fig. 1. They correspond to Prolog types, with the addition of "exact", which only allows a single specified atom (like a zero-arity compound), and "any", which allows any value. Some types are polymorphic, e.g. lists can be instantiated to lists of a specific type. There are also two combinators, `one_of` that allows union types as well as `and`, which is the intersection of two types.

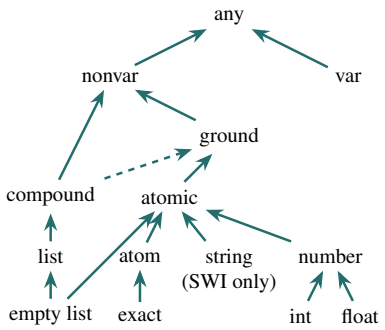


Fig. 1. Abstract Type Domain

Combination of built-in types is certainly very expressive. While such structures cannot be inferred easily without prior definition, as a realistic example, it is possible to define a tree of integer values by using the `one_of` combinator as follows:

```
defspec(tree, one_of([int, compound(node(tree, int, tree))])).
```

Valid `trees` are `1`, `node(1, 2, 3)`, `node(node(0, 1, 2), 3, 4)` but not, e.g. `tree(1, 2, 3)`, where the functor does not match, or `node(a, b, c)` which stores atoms instead of integer values. Note that it is also possible to use a wildcard type to define a tree `tree(specvar(X))`, which passes the variable down into its nodes. `specvars` are a placeholder to express that two or more terms share a common, but arbitrary type. This can be used to define template-like data structures which can be instantiated as needed, e.g., as a `tree(int)`.

The second kind of annotations specifies how predicates may be called and, possibly, what parameters are return values. We re-use two different annotations for that:

1. *Preconditions* specify types for all arguments of a predicate. For a call to be valid, at least one precondition has to be satisfied.
2. *Postconditions* add promises for a predicate: if the predicate was called with certain types and if the call was successful, specified type information holds on exit.

Both pre- and postconditions must be valid for every clause of the specified predicate. Consider a variation of `member/2`, where the second argument *has to be* a list of atoms, and the first argument can either be an `atom` or `var`:

```
atom_member(H,[H|_]).   atom_member(E,[_|T]) :- atom_member(E,T).
```

Instead of checking the terms in the predicate, type constraints describing intended input are added via *plspec*'s pre- and postconditions. The following preconditions express the valid types one has to provide: the first argument is either a variable or an atom, and the second argument must be a list of atoms.

```
:- spec_pre(atom_member/2, [var, list(atom)]).
:- spec_pre(atom_member/2, [atom, list(atom)]).
```

As the second argument is always a ground list of atoms, we can assure callers of `atom_member/2`, that the first term is bound after the execution using a postcondition:

```
:- spec_post(atom_member/2, [var, list(atom)], [atom, list(atom)]).
```

Postconditions for a predicate are defined using two argument lists: they are read as an implication. For `atom_member/2` above, this means that "if the first argument is a variable and the second argument is a list of atoms, and if `atom_member/2` succeeds, it is guaranteed that the second argument is still a list of atom, but also that the first argument will be bound to an atom". If the premise of the postcondition does not hold or the predicate fails, no information is gained.

Extensions to plspec The traditional understanding if there are two instances of the same type variable, e.g. in a call such as `spec_pre(identity/2, [X, X])`, is that both arguments *share all types*. Yet, we want to improve on the expressiveness of, say, `spec_pre(member/2, [X, list(X)])`, and allow heterogeneous lists. This extension is not yet implemented in *plspec* itself and is only part of the static analysis in *plstatic*. In order to express how the type of type variables is defined, we use `compatible` for the homogeneous and `union` for the heterogeneous case.

If a list is assigned the type `list(compatible(X))`, every item in the list is assigned the type `compatible(X)`. Now *plstatic* checks whether all these terms share all types, thus enforcing a homogeneous list. Analogously, if a list is assigned the type `list(union(X))`, every item in the list is assigned the type `union(X)`. But instead of intersecting, *plstatic* collects the types of these terms and builds a union type.

To give an example for the semantics of compatible and union, the list `[1, a]` has the *inner* type `one_of([int, atom])` under the semantics of a union, and results in a type error (as the intersection of int and atom is empty) if its elements should be compatible. A correct annotation for `member/2` would be the following postcondition: `spec_post(member/2,[any,list(any)],[compatible(X),list(union(X))])`, i.e., the list is heterogeneous, and the type of the first argument must occur in this list.

4 Our Type System

In the following, we describe a prototype named *plstatic*. It uses an abstract interpreter in order to collect type information on Prolog programs and additionally to identify type errors on a best-effort basis, without additional annotations. The tool is available at `https://github.com/isabelwingen/prolog-analyzer`.

Purpose and Result The tool *plstatic* performs a type analysis on the provided code. All inferred information can be written out in form of annotations in *plspec* syntax, or HTML data that may serve, e.g., as documentation. Naturally, *plstatic* shows an overview of type errors, which were found during the analysis.

As typing can be seen as a special case of abstract interpretation [1], we use *plspec*'s annotations to derive an abstract value, i.e. a type, for terms in a Prolog clause. Abstract types correspond to the types shown in Fig. 1, where a type has an edge pointing to a strict supertype. However, as distinguishing ground from nonvar terms often is important, compound terms are tried to be abstracted to the ground type first, represented by the dashed edge. We use the least upper bound and greatest lower bound operations as they are induced by the type subset relation. This analysis is done statically and without concrete interpretation of Prolog code, based on *plspec* annotations and term literals.

A Note on Annotations *plstatic* works without additional annotations in the analysed code. We are able to derive type information from (a subset of) built-in (ISO) predicates. For those predicates, we provided pre- and postconditions, which are also processed by the term expander. We also annotated a few popular libraries, e.g. the lists library.

For predicates lacking annotations, types can be derived if type information exists for predicates called in their body or can be inferred from unification with term structure in the code. Derived types describe intended success for the unannotated predicate. Naturally, precision of the type analysis improves with more annotations.

4.1 Tool Architecture

plstatic is implemented in Clojure, a Lisp dialect running on the JVM. It might as well have been implemented as a meta-interpreter in Prolog, but Clojure allows easier integration into text editors, IDEs and potentially also web services. However, this requires to extract a representation of the Prolog program. We decided against parsing Prolog as operator definitions are hard to add during runtime and it is guaranteed to lose transformations done by term expanders[1]. Instead, we add a term expander ourselves before we load the program. It implements *plspec*'s syntax for annotations and extracts those alongside the program itself. All gathered information is written into a separate file in edn syntax[2], which can easily be read back in Clojure.

plstatic consists of two parts: an executable jar containing the tool for static analysis and the aforementioned Prolog term expander. The architecture of *plstatic* is pictured in Fig. 2. The analysis core is started with parameters specifying the path to a Prolog source file or directory and a Prolog dialect (for now, "swipl" or "sicstus"). Additionally, the path to the term expander can be passed as an argument as well, if another syntax for annotations than *plspec*'s is desired.

Special care has to be taken when an entire directory is analysed: when modules are included, it is often not obvious where a predicate is located. In particular, it can be hard to decide whether a predicate is user-defined, shipped as part of a library or part of the built-in predicates available in the user namespace. Thus, when the edn-file is imported, a data structure is kept in order to resolve calls correctly.

As our evaluation in Section 5 uses untrusted third-party code, we take care that the Prolog code, that may immediately run when loaded, is not executed. Instead, the term expander does not return any clause, effectively removing the entire program during compilation. Trusted term expanders can be loaded beforehand if required.

4.2 Analysis

Our approach to type inference implements a classical abstract interpreter. In the first phase, every clause is analysed individually. We use *plspec*'s annotations of the clause and the sub-goals to derive an abstract type domain for all terms in the clause.

In the second phase, we broaden the analysis to a global scope: After the first phase, we have obtained a typing for every clause, which describes the types that the terms

[1] Term expansion is a mechanism that allows source-to-source transformation.
[2] https://github.com/edn-format/edn

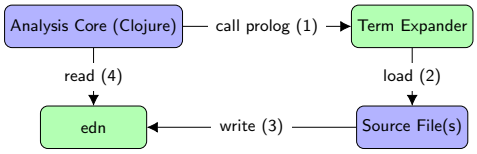


Fig. 2. Tool Architecture

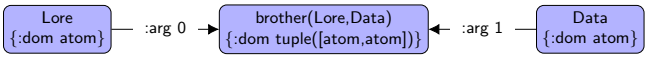


Fig. 3. An Example Environment (Using edn-Formatted Maps)

have after a successful execution of the clause. The inferred type information for all clauses of a predicate, can be stored as a postcondition. Perhaps this postcondition is more accurate than the already provided one. In this case, the analysis of a predicate `p` would in turn improve the analysis result for clauses that call `p`.

So, we create new postconditions for every predicate. Afterwards, we repeat the first and the second phase, until, eventually, a fixpoint is reached. As the type domains are finite, the types of the single terms can only reach a certain degree of accuracy. Therefore, a fixpoint will be reached eventually. Then, no further information can be obtained and the type analysis terminates.

Example: Rate My Ship The following code will accompany us during this section.

```
ship(Ship) :- member(Ship, [destiny, galactica, enterprise]).
rating(stars(Rate)) :- member(Rate, [1,2,3,4,5]).
rate_my_ship(S,R) :- ship(S), rating(R).
```

Preparation Before the analysis, we prepare the `edn` produced by the term expander. To determine the module of a sub-goal, a mapping of modules to predicates as well as imports is kept in order find the correct module of each predicate call. For every loaded predicate, we check, if there are pre- and postconditions specified. Otherwise, they are created containing any-types during the preparation as follows:

The construction uses the literals in the clause heads. Lists, compounds and the different atomic terms are recognised as such. For variable terms, we assume the type `any`. Consider the clause head `check(nil, foo(X,Y))`, for which *plstatic* creates `[one_of([var,atom(nil)]), one_of([var, compound(foo(any, any))])]` as a precondition, and `[any,any] => [atom(nil),compound(foo(any, any))])` as a postcondition. It is important that the created pre- and postconditions are general enough to be valid for every clause.

Below, we show the generated specs for our example after the preparation step:

```
:- spec_pre(ship/1, [any]).
:- spec_post(ship/1, [any], [any]).
:- spec_pre(rating/1, [one_of([var, compound([stars(any)])])]).
:- spec_post(rating/1, [any], [compound([stars(any)])]).
:- spec_pre(rate_my_ship/2, [any, any]).
:- spec_post(rate_my_ship/2, [any, any], [any, any]).
```

Phase 1: Clause-Local Analysis Because of the logic nature of prolog, it is not sufficient to store the current type for a variable at a given point, as relationships between terms caused, e.g., by unification, have to be considered as well.

Contrary to traditional approaches, we use an environment in form of a directed graph to store relationships between variables per clause. Every term that occurs in the currently considered clause is represented as a node in the environment. The inferred types of the terms are saved as attributes in the corresponding nodes. Relationships between terms and sub-terms (e.g. `[H|T]`) or postconditions are saved as labelled edges between the term nodes. An example is given in Fig. 3, where the structure of a compound term `brother(Lore, Data)` is shown.

During the analysis of a clause, the type domains of the terms are updated and their precision is improved. For this, annotations of called predicates and the relationships between the terms are considered. When new type information about a term is gained, the greatest lower bound is calculated by intersecting both domains. When considering variables in Prolog however, this comes with some pitfalls that are discussed in more details in *Step 2*. If the type intersection is empty, no concrete value is possible for the Prolog term and a type error is reported. However, we have to assume that all given annotations are correct.

Step 1: Clause Head The environment is initialised with all terms occurring in the head of the clause. Information about the head of the clause can be derived from the preconditions. According to *plspec*, at least one precondition must be fulfilled.

Consider the following example: the predicate `cake(X, Y)` is annotated with the preconditions `[atom, int]` and `[int, atom]`. This means that `cake/2` expects an atom and an integer, no matter the order. We might derive `one_of([atom, int])` as type for both `X` and `Y`. Then, `X=1,Y=2` would be valid input, as both fulfil their individual type constraint, but together, they violate the original precondition. To prevent such errors, we create an artificial tuple containing all arguments, whose domain is a union-type containing all supplied preconditions. This artificial term functions as a "watcher", and ensures all type constraints. For the `cake` predicate, the term `[X,Y]` is added to the environment, with `one_of([tuple([atom,int]), tuple([int,atom])])` as its type. Once we know a more specific type for, e.g., `Y`, we can derive which of the two options must be valid for the "watcher", and therefore, we can derive a type for `X`. The environment is pictured in Fig. 4.

Due to page limitations, we only present the environment of `rating/1`.

```
[Rate]  domain: tuple([compound(stars([any]))]).
Rate    domain: compound(stars([any])).
```

A Note on Compound and List Terms Whenever a type is added to the domain of a compound or list term, types for the children of this term are automatically derived and added to their domain, if possible.

Step 2: Evaluate Body We analyse the body step by step, making use of (generated or annotated) pre- and postconditions of all sub-goals one after another using abstract

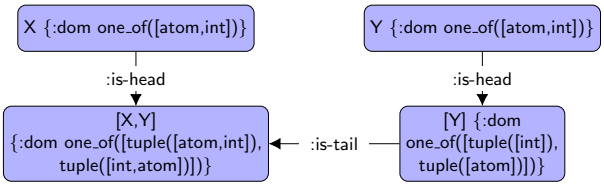


Fig. 4. Environment with a Watcher (Using edn-Formatted Maps)

Table 1. Environment for `rate_my_ship/2`

Variable Term	Clause Head	after 1st sub-goal	after 2nd sub-goal
[S, R]	tuple([any, any])	tuple([any, any])	tuple([any, any])
[R]	tuple([any])	tuple([any])	tuple([any])
R	any	any	compound(star([any]))
S	any	any	any

types as values. On the first occurrence of a term, it is added to the environment. Similarly, to the clause head, at least one precondition of the sub-goal must be compatible with the combination of the arguments it is called with.

The analysis does not step into the sub-goal, and only uses pre- and postconditions. A postcondition specifies type constraints on a term after the called predicate succeeds. Thus, it is checked which premises of postconditions are fulfilled. Then, the greatest lower bound of the current type domain and the possible conclusion of the postconditions is calculated in order to improve precision. An example is shown in Table 1.

A Note on Type Variables We have introduced two new kinds of type variables (cf. Section 3): `union` and `compatible`. It is possible to use `union(X)` or `compatible(X)`, where `X` is a type variable. Both are placeholders for yet unknown types and express two different relationships between terms:

Every term that is assigned the type `union(X)` contributes to the definition of the type that is `X`. The connection is made by adding a labelled edge `:union` between the term and `X`. Then, the domains of the terms are filled as described. At the end of the analysis step, the union type is inferred via the least upper bound of all connected terms.

On the other hand, terms that are assigned the type `compatible(X)`, must be compatible with all other occurrences of this type, and the inferred union type of `X`, implying that their intersection is not empty. As with the `union` type, we create a labelled edge `:compatible` connecting the term to `X`. These edges are processed *after* all union edges have been visited. We create the intersection of all involved terms, and assign it to the domain of all terms connected via a `:compatible` edge.

In order to determine the type for a union type variable, it is required to know all contributing terms. If a term is known to be a compound or a list of a known size, the assigned type is passed down to its sub-terms using the mechanisms described above. Yet, consider a term that is a Prolog variable, e.g. `T`, with the inferred type

`list(union(X))`. Then, the length of the list and its possible elements are unknown, as the Prolog variable might be bound at a later point.This requires an additional step in order to ensure that the domain for the type variable `X` is compiled correctly: we opted to add a `:has-type` edge to the environment, which connects a Prolog variable, e.g `T`, to an artificially created variable `T__<uuid>` storing the inner type, i.e. `union(X)` in the example above. Whenever the domain of the variable is updated, so is its list type. The artificial list type variable then is connected with `union(X)`.

For `compound` and `tuple` *type specifications*, an artificial term is created and linked to the variable term via an edge labelled `:artificial`. This is required to mimic unification of Prolog variables. Whenever the domain of the variable term is updated, the artificial term's domain is updated as well. Finally, the information is propagated into the corresponding sub-terms if required.

Have a look at `member/2` used in the body of `ship/1`. The provided postcondition is `post_spec(member/2, [any, any], [compatible(X), list(union(X))])`. Therefore, after analysing the body of `ship/1`, we obtain the following environment:

```
galactica     atom            galactica  ----union---> X
destiny       atom            destiny    ----union---> X
enterprise    atom            enterprise ----union---> X
Ship          atom            Ship       -compatible-> X
```

Step 3: Term Relationships After analysing the body, all terms in the clause are included in the environment. Then, nodes that may be destructured, i.e. lists and compound terms, are looked up in the graph. As sub-terms, e.g. `X` in `a(X)`, can be used individually, i.e. without their enclosing compound term, in subsequent sub-goals, inferred information has to be propagated back to the larger compound term. We introduce the following edges in order to provide the necessary mechanism:

For lists, we extract the head and tail terms and add them to the environment, if they are not already contained. Those terms are marked with special edges `:is-tail` and `:is-head` (cf. Fig. 4) pointing to the original list. For compounds, we add the argument terms to the environment and store the position of every term in the compound by adding an edge `:pos` (cf. Fig. 3).

The following edges are added to the environment for `rate_my_ship/2`:
`R --is-head--> [R]`, `S --is-head--> [S, R]` and `[R] --is-tail--> [S, R]`

A Note on Prolog Variables The any-type can be split into two disjoint sets: variables and non-variable terms. Non-variable terms may only be inferred to be more concrete in every step unless a type error is determined. Yet, Prolog variables have the unique property that their type can change, as they can be bound to, say, an atom, which is not a sub-type. To take this into account, a different intersection mechanism is required for variables:

- Preconditions of the *currently analysed* predicate may render a variable non-variable.
- Preconditions of a *called sub-goal* cannot render a variable term non-variable.
- Postconditions of a *called sub-goal* may render a variable term non-variable.
- Once a Prolog variable is bound to a non-variable, it behaves like any non-variable.

Step 4: Fixed-Point Algorithm During the prior steps, we have added some edges. These are now used to update the domains of the linked terms using the knowledge expressed by the edges. If the environment no longer changes, we have consumed all collected knowledge and have found a preliminary result.

`R` links back to the list it is part of in the environment of `rate_my_ship/2`. We can therefore update those terms containing `R`, as we have found a more precise type for `R`.

```
[S, R]  tuple([any, compound(star([any]))]).
[R]     tuple([compound(star([any]))])    R   -head-> [R]
R       compound(star([any]))             S   -head-> [S, R]
S       any                               [R] -tail-> [S, R]
```

Phase 2: Global Propagation of Type Information During the local analysis, each clause was inspected in isolation. The type domains in the returned environments contain the types after a successful execution of a clause *with the initial knowledge*. The gathered information can be propagated to the caller of the corresponding predicate in order to improve the precision of the type inference.

Each environment can be used to generate a conclusion of a postcondition. If a predicate succeeds, at least one of its clauses succeeded. As a postcondition must be valid for the entire predicate, the conclusion of a new postcondition is the union of the all gained conclusions of the corresponding clauses. This newly gained knowledge (in form of a postcondition) is added to the analysed data for every predicate. Afterwards, both local analysis and global propagation are triggered, until a fixed point is reached. Inferred pre- and postconditions can be written out after analysis in *plspec*'s syntax.

Example: append/2 Consider the append program:

```
append([], Y, Y).    append([H|T], Y, [H|R]) :- append(T, Y, R).
```

For the first clause, *plstatic* would derive the types `[list(any), any, any]`. For the second clause, we gain no additional information from the body, because `append/2` is calling itself, so we derive the types `[list(any), any, list(any)]`. To create a conclusion of a postcondition for the predicate, we need to combine the results of the two clauses. Unfortunately, as the type of the third argument is `any` in one case, it swallows the more precise type `list(any)`. We obtain the following conclusion: `[list(any), any, any]`. While the *intention* is that the second and third arguments are lists as well, this cannot be inferred without annotations.

As you have probably noticed, *plstatic* has not yet found the accurate type `atom` for `S` or `R` in `rate_my_ship/2`. This is because the pre- and postconditions of `ship/1` have not been updated yet, so *plstatic* has no way of knowing that `S` is an atom. In the first phase, we have concluded that the argument given to `ship/1` must be of type `atom` after a successful execution. As `ship/1` has only one clause, we can infer the postcondition: `:- post_spec(ship/1, [any], [atom]).` Analogously, we obtain `:- post_spec(rating/1, [any], [compound(stars([atom]))]).`

The propagation of the newly gained knowledge is shown in Table 2. Afterwards we can update the pre- and postconditions for `rate_my_ship/2`, but `ship/1` and `rating/1` are not affected from this. If our program has no more clauses, the fixpoint is reached, and the analysis stops.

Table 2. Environment for `rate_my_ship/2`

Variable Term	Newly Gained Knowledge	After Propagation
[S, R]	tuple([any, compound(star([any]))])	tuple([*atom*, compound(star([*atom*]))])
[R]	tuple([compound(star([any]))])	tuple([compound(star([*atom*]))])
R	*compound(star([atom]))*	compound(star([atom]))
S	*atom*	atom

A Note on Backtracking Preconditions specify a condition which must be fulfilled at the moment of the call, and postconditions can provide information about the type of the used terms after a successful execution. The caller of a predicate is unaware which clause provided the result. Thus, the union of all gained type information has to be considered in the second phase. As a result, it is safe to ignore backtracking: yet, precision could in some cases be improved if clause ordering and cuts (`!`) were considered.

5 Evaluation

To our knowledge, papers on type systems for Prolog usually omit an evaluation of their applicability for existing, real-world Prolog code and offer insights on their type inference mechanisms on small toy examples, such as the well-known `append` predicate. However, we want to consider code that is more involved than homework assignments. There is no indication to what extent type inference approaches are applicable to the real world, or how much work has to be spent re-writing code for full-fledged type systems.

In contrast, we baptise *plstatic* by fire and evaluate for how many variables in the code we can infer a type that is more precise than `any`. For this, we use smaller SWI community packages[3], as well as ProB [9], a model checker and constraint solver that currently consists of more than 120 000 lines of Prolog code.

5.1 Known Limitations

Currently, we face three limitations in *plstatic*: firstly, as we try to avoid widening whenever possible, i.e., we try to use the most precise type like a `one_of` instead of generalising to their common supertype, performance is not too good. Analysis of small projects runs neglectably fast, yet ProB requires several hours to complete a full analysis. Secondly, libraries throw a wrench into our scheme: modern Prolog systems pre-compile the code. Hence, meta-programs, such as term expanders, cannot access their clauses. Thus, library code is not considered and *plstatic* has to rely on annotations. Currently, we only provide annotations for large parts of the lists library (for both *SWI Prolog* and *SICStus Prolog*) and the AVL tree library (for *SICStus Prolog* only). Otherwise, for all library predicates that are not annotated, an `any` type has to be assumed. Thirdly, we currently do not consider disjunctions and if-then-else constructs, but may gain additional precision once this is implemented.

[3] `http://www.swi-prolog.org/pack/list`

Table 3. Amount of Inferred Types for Variables

Repository	# Variables	Inferred Types	Unknown Calls
bddem	196	31.63 %	57.6 %
dia	400	68.5 %	8.23 %
maybe	32	6.25 %	70.0 %
plsmf	67	37.31 %	37.5 %
quickcheck	122	42.6 %	34.1 %
thousands	19	94.73 %	0.0 %
∅ SWI Community Packages	68344	21.8 %	39.0 %
ProB	81893	21.2 %	20.8 %

Additionally, there is an inherent limitation in our analysis strategy: some predicates may really work on *any* type, e.g. term type checking predicates (such as `ground/1` or `nonvar/1`) or the `member/2` predicate regarding the first argument. As no similar analysis for Prolog programs exists yet and type inference by hand is infeasible for large programs, it is certainly hard to gauge the precision of our type inference.

5.2 Empirical Evaluation

In Table 3, the results of some repositories[4] and the mean value of the 198 smallest community packages is shown. We give the amount of Prolog variables, and the percentage of which we can infer a type that is a strict sub-type of `any`. For reference, we also give the amount of calls to unknown predicates in order to give an idea how many missing types are caused by, e.g., library predicates lacking annotations. Though, once a variable is assigned an `any` type, the missing precision typically is passed on to terms that are interacting with the `any` term as the predicate is implemented in a library.

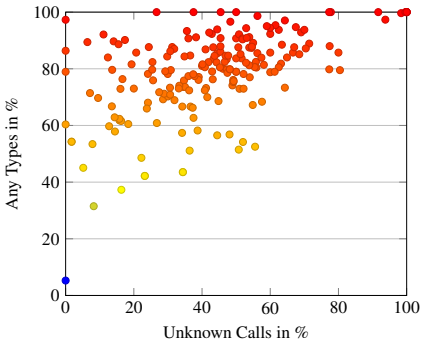


Fig. 5. Correlation Between Unknown Calls and Inferred Types

At first glance, the fraction of inferred types seems to be rather low. For some repositories, such as “dia” and “thousands”, a specific type could be inferred for a large percentage of variables. Note that in return, the amount of unknown calls is relatively low. Then, there are repositories such as “bddem” and “plsmf”, which both are wrappers of a C library. As such, the interop predicates are unknown and the inferred types are significantly lower. Finally, there are packages like “maybe”, “quickcheck” and projects such as ProB, that make use of other libraries, conditional compilation, meta-calls and other features that decrease accuracy of type inference.

[4] Full results: `https://github.com/pkoerner/plstatic-results/tree/lopstr19`

Overall, we were surprised how small the amount of inferred types was. Though, one has to consider that a large amount of predicates are library calls, e.g. into the popular CLP and CHR libraries. In Fig. 5, we show this relation. One can clearly recognise that (unknown) library calls negatively impact the results of our type analysis. Yet, many auxiliary predicates are written to be polymorphic and deal with any type.

plstatic was able to find several errors: many SWI libraries have been broken with changes introduced in *SWI Prolog* 7 [18]. Strings now are proper strings, where legacy code relies on the assumption that they are represented as code lists. Furthermore, *plstatic* located calls in ProB that were guaranteed to fail every time due to type errors. These calls decide whether a backend is usable in order to solve a given predicate and always fail. Thus, the errors have gone unnoticed for eight years, as the backend simply was not used. One error was reported due to missing term expansion as we did not execute untrusted Prolog code. We found another false-positive due to meta predicate annotations which add the module to a goal, thus altering the term structure. Additionally, we found some extensions *SICStus Prolog* made to the ISO standard that we were not aware of: e.g., arithmetic expressions allow expressions such as `X is integer(3.14)` or `log(2, 42)` that are not part of ISO Prolog but valid in *SICStus Prolog* were reported as errors in our type describing arithmetic expressions.

6 Conclusion and Future Work

In this paper, we presented *plstatic*, a tool that re-uses its annotations in order to verify types statically where possible. *plstatic* was able to locate type errors in several existing Prolog repositories. Yet, without annotations of further libraries, the amount of actual inferred types remains relatively low. We invite the Prolog community to discuss whether such type annotations are desired and should be shipped as part of packages.

There remains some work in *plstatic*: performance bottlenecks need to be reviewed to reduce the time required for analysis. Furthermore, the analysis would heavily benefit from a mechanism for the term expander to hook into library packages or manual annotations. It might also be possible to analyse some pre-compiled library beforehand and re-use those results in the analysis of the main program. We also plan to implement semantics for new types, for which the structure is not specified, but they may only be created by libraries. E.g., Prolog streams are impossible to create one without calling the corresponding predicates. Other examples include ordered sets or AVL trees, where it is possible to create or manipulate such a term, but it is heavily discouraged as it is very easy to introduce subtle errors.

Moreover, it would be exciting to compare the amount of inferred types to similar implementations such as CiaoPP. We assume their analysis to be stronger, but suspect that Ciao's approach might not scale as well for larger programs. Yet, comparison might be hindered, again, because features of other Prolog systems are not supported.

In [17] and also in the evaluation of *plspec* [7], it was determined that the overhead of run-time type checks can be enormous, especially if applied to recursive predicates. With additional type information, a large amount of run-time checks can be eliminated, as, e.g., proposed by [17]. It is fairly straightforward to generate a list of already discharged annotations and use that as a blacklist in *plspec*.

It is well-known that compilers often benefit heavily from type information. An interesting research question is to investigate the impact of type information, e.g. gained by *plstatic* or by annotations, when added to the binding-time analysis of a partial evaluator, such as LOGEN [10]. This might greatly reduce the work required of manually improving generated annotations in order to gain additional performance.

As a more pragmatic approach to future work, it would be greatly appreciated if the state-of-the-art of Prolog development tooling could be improved. Currently, IDEs and editor integrations are lacking. Including type information would be a great start.

References

1. P. Cousot. Types as abstract interpretations. In *Proceedings POPL*, pages 316–331. ACM, 1997.
2. J. P. Gallagher and K. S. Henriksen. Abstract domains based on regular types. In *Proceedings ICLP*, pages 27–42. Springer, 2004.
3. M. V. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J. F. Morales, and G. Puebla. An overview of Ciao and its design philosophy. *TPLP*, 12(1-2):219–252, 2012.
4. D. Jeffery. *Expressive Type Systems for Logic Programming Languages*. PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, 2002.
5. M. Jimenez, T. Lindahl, and K. Sagonas. A Language for Specifying Type Contracts in Erlang and Its Interaction with Success Typings. In *Proceedings ERLANG*, pages 11–17. ACM, 2007.
6. S. P. Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.
7. P. Körner and S. Krings. plspec – A Specification Language for Prolog Data. In *Proceedings WFLP*, volume 10997 of *LNAI*, pages 198–213. Springer, 2017.
8. T. Lakshman and U. S. Reddy. Typed Prolog: A Semantic Reconstruction of the Mycroft-O'Keefe Type System. In *ISLP*, volume 91, pages 202–217, 1991.
9. M. Leuschel and M. J. Butler. Prob: A model checker for B. In *Proceedings FME*, volume 2805 of *LNCS*, pages 855–874. Springer, 2003.
10. M. Leuschel, S. J. Craig, M. Bruynooghe, and W. Vanhoof. Specialising interpreters using offline partial deduction. In *Program Development in Computational Logic*, pages 340–375. Springer, 2004.
11. R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.
12. A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial intelligence*, 23(3):295–307, 1984.
13. F. Pfenning. *Types in logic programming*. MIT Press Cambridge, Massachusetts, USA, 1992.
14. F. Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fundam. Inform.*, 19(1/2):185–199, 1993.
15. E. Rohwedder and F. Pfenning. Mode and termination checking for higher-order logic programs. In *European Symposium on Programming*, pages 296–310. Springer, 1996.
16. T. Schrijvers, V. S. Costa, J. Wielemaker, and B. Demoen. Towards typed Prolog. In *Proceedings ICLP*, volume 5366 of *LNCS*, pages 693–697. Springer, 2008.
17. N. Stulova, J. F. Morales, and M. V. Hermenegildo. Reducing the overhead of assertion run-time checks via static analysis. In *Proceedings PPDP*, pages 90–103. ACM, 2016.
18. J. Wielemaker. SWI-Prolog version 7 extensions. In *Proceedings CICLOPS-WLPE*, page 109, 2014.

A Port Graph Rewriting Approach to Relational Database Modelling

Maribel Fernández[1], Bruno Pinaud[2] and János Varga[1]

[1] King's College London, Department of Informatics, London WC2B 4BG
[2] LaBRI, Université de Bordeaux, 351 Cours de la Libration, 33405 Talence cedex
janos.varga@kcl.ac.uk

Abstract. We present new algorithms to compute the Syntactic Closure and the Minimal Cover of a set of functional dependencies, using strategic port graph rewriting. We specify a Visual Domain Specific Language to model relational database schemata as port graphs, and provide an extension to port graph rewriting rules. Using these rules we implement strategies to compute a syntactic closure, analyse it and find minimal covers, essential for schema normalisation. The graph program provides a visual description of the computation steps coupled with analysis features not available in other approaches. We prove soundness and completeness of the computed closure. This methodology is implemented in PORGY.

Keywords: relational databases, database design, port graph, graph transformation, functional dependency, minimal cover

1 Introduction

Relational database design includes conceptual and logical modelling, as well as physical modelling. The theory behind these steps is well-understood (it is part of the syllabus of many databases courses [16]), and highlights the advantages of developing normalised database designs. Yet, database professionals often consider normalisation too cumbersome and do not apply normalisation theory, due to the lack of adequate tools to support logical modelling [6].

Formal, graph-based approaches to database design have used labelled graphs or hypergraphs [1,5,7]. We advocate a new approach to database modelling using *attributed port graphs*, which are graphs where edges are connected to nodes at specific points, called ports. Attributes of nodes, edges and ports are used to represent properties of the system modelled. Port graphs were introduced in [2] to model biochemical systems and have been used in various domains [14]. Port graphs are a good data structure to store and to visualise relational schema: ports provide additional visual information about the design. We propose to represent relational attributes and functional dependencies as nodes, and use edges to link attributes and dependencies; ports indicate the role of the attribute in the dependency. This representation has advantages when computing properties of the schema, such as syntactic closure of the set of dependencies, a crucial step in producing a normalised schema. We specify an algorithm to compute closures

using *port graph rewriting rules* controlled by strategies. Our system has been implemented in PORGY [3] – a visual, interactive modelling tool. PORGY provides a graphical interface to specify an initial model, port graph rewriting rules and strategies. It displays the set of rewrite derivations (a *derivation tree*) and includes features such as cycle detection, to facilitate debugging.

Summarising, our main contributions are:

1. a Visual Domain Specific Language (VDSL) specifically tailored to model relational database schemata (Section 3);
2. a new visual representation of Armstrong's axioms to infer functional dependencies, using the port graph VDSL mentioned above (Section 4.1);
3. a sound and complete strategic graph program to compute the syntactic closure of a set of functional dependencies, with examples (Sections 4.2,4.3);
4. an implementation[3] in PORGY, together with a set of techniques to query the relational database design, using PORGY's derivation tree and graphical interface to analyse properties of the model: In particular, we show how to solve *the membership problem* (Section 4.4);
5. a strategy and a set of transformation rules to simplify sets of dependencies, as required to compute a minimal cover (Section 5).

Related Work. Hypergraphs are used for relational database design in [7,13]. Using directed graphs candidate keys of a relation are computed in polynomial time [24]. A special family of labelled graphs, FD-graphs, were introduced in [5] to obtain closures of functional dependencies. In terms of graph transformations for database modelling we highlight two works. Hypergraph rewriting was used for the representation of functional dependencies [7] and Triple Graph Grammars were used to optimise a database schema [17].

Our contribution and main difference with respect to these works is the design of a domain-specific visual language with emphasis on interactive modelling, including strategies to control the application of rules, and the use of the derivation tree as part of the visualisation framework, giving the modeller access to all the sequences of transformation steps, to facilitate the analysis of the system. Port graphs were used to compute transitive closures in [27]. Here we compute the full Armstrong closure (not just transitive closure), and show how to use the derivation tree to analyse closures and answer queries about the database model, such as whether a given functional dependency is in the closure of a set of dependencies (the membership problem), and compute minimal covers.

PORGY's strategy language is strongly inspired by PROGRES [25], GP [23] and by strategy languages developed for term rewriting [11,19]. None of the available graph rewriting tools permits users to visualise the derivation tree, as in PORGY, where users can interactively visualise alternative derivations, follow the development of specific redexes, etc. When computing the closure of a set of dependencies, the derivation tree permits to see how each dependency is generated, offering a direct visualisation of the inference steps according to Armstrong's axioms.

[3] github.com/janos-varga/Porgy

2 Background

2.1 Relational Databases

We assume that the reader is familiar with the theory of logical design of relational databases [22], in particular, the definitions of: *relation schema*, *attribute*, *candidate key* and *functional dependency* (FD). We refer to a single attribute with letters from the beginning of the alphabet $A, B, \ldots$ and to attribute sets with letters from the end of the alphabet W, X, Y, Z. Let $\mathcal{R}(\mathcal{A}) = \{R_1, \ldots, R_k\}$ be a set of relation schemata over a set $\mathcal{A}$ of attributes. Let $\mathcal{FD} = \{\Sigma_1, \ldots, \Sigma_k\}$ be the respective sets of functional dependencies and $\mathcal{CK} = \{C_1, \ldots, C_k\}$ be the respective sets of candidate keys. A relational database schema is a tuple $\mathcal{DB} = (\mathcal{R}(\mathcal{A}), \mathcal{FD}, \mathcal{CK})$.

We assume familiarity with the inference rules known as Armstrong's Axioms: Reflexivity (or Trivial Dependency), Augmentation, Transitivity, Union, Decomposition, Pseudotransitivity. These rules are sound and complete [4,9]. From now on, by syntactic closure we will mean Amstrong's syntactic closure, that is, the set Σ^+ of all FDs that can be inferred from Σ using Armstrong's Axioms, or equivalently, using the sound and complete subset consisting of Reflexivity, Transitivity and Augmentation, stated below following Beeri et al. [9].

(A1) Reflexivity: **if** $Y \subseteq X$ **then** $X \to Y$.
(A2) Augmentation: **if** $Z \subseteq W$ and $X \to Y$ **then** $XW \to YZ$.
(A3) Transitivity: **if** $X \to Y$ and $Y \to Z$ **then** $X \to Z$.

Syntactic closures are used to compute **minimal covers** of sets of FDs. The *minimal cover* Σ_{min} of Σ is a set of dependencies that fully represent Σ and satisfy the following three conditions [21]:

1. all the FDs in Σ_{min} have singleton right sides;
2. Σ_{min} is left-reduced: if one attribute is removed from any left side then Σ can no longer be inferred from Σ_{min};
3. Σ_{min} is nonredundant: if any FD is removed from Σ_{min} then Σ can no longer be inferred from it.

Our goal is to provide visual algorithms to compute syntactic closures and minimal covers. This work assumes that a) FDs have singleton right sides and b) there are no cyclical dependencies. Assumption (a) is standard in the relational database literature, without loss of generality, under Armstrong's Decomposition rule. The problem of cyclical dependencies reduces to kernel search in a directed graph which is NP-complete.

2.2 Port Graph Rewriting and Porgy

We recall the notion of attributed port graph rewriting (see [14] for more details).

Definition 1 (Attributed port graph). *An* attributed port graph $G = (V, P, E, D)_{\mathcal{F}}$ *is a tuple* (V, P, E, D) *of pairwise disjoint sets where:*

- *V is a finite set of nodes; $n, n_1, \ldots$ range over nodes;*
- *P is a finite set of ports; $p, p_1, \ldots$ range over ports;*
- *E is a finite set of edges between ports; $e, e_1, \ldots$ range over edges; two ports may be connected by more than one edge;*
- *D is a set of records, which are sets of pairs attribute-value;*

and a set $\mathcal{F}$ of functions Connect, Attach and $\mathcal{L}$abel such that:

- *for each edge $e \in E$, Connect(e) is the pair (p_1, p_2) of ports connected by e;*
- *for each port $p \in P$, Attach(p) is the node n to which the port belongs;*
- *$\mathcal{L}abel : V \cup P \cup E \mapsto D$ is a labelling function that returns a record for each element in $V \cup P \cup E$.*

For each node $n \in V$, $\mathcal{L}$abel(n) contains an attribute Interface whose value is the list of names of its ports.

A *port graph rewrite rule* is itself a port graph $L \Rightarrow_C R$ consisting of two subgraphs L and R, called *left-hand side* and *right-hand side*, respectively, together with an *arrow* node that links them. Each rule is characterised by its arrow node, which has a unique name (the rule's label), an optional attribute *Where* defining a Boolean condition C that restricts the rule's matching, and ports to control the rewiring operations when rewriting steps are computed. Each port in the arrow node has an attribute *Type* that can have one of three different values: *bridge*, *wire* and *blackhole*. A port of type *bridge* must have edges connecting it to L and to R (one edge to L and one or more to R): it thus connects a port from L to ports in R. A port of type *blackhole* must have edges connecting it only to L (one edge or more). A port of type *wire* must have exactly two edges connecting to L and no edge connecting to R.

The ports and edges associated with the arrow node specify a mapping between ports in the left and right-hand sides of the rule, following the Single-PushOut approach [20]. This mapping is used during rewriting, to redirect the edges that connect the redex to the rest of the graph once the redex is rewritten (as explained below).

For examples of rewrite rules, we refer the reader to Section 4. It is possible to specify a rule condition requiring that a particular edge does NOT exist in the graph to be rewritten. In Porgy such conditions are graphically represented as a double line grey edge with an X, which is called an anti-edge [15].

A *match* $g(L)$ of the left-hand side is found in G if there is a total port graph morphism g from L to G such that if the arrow node has an attribute *Where* with value C, then $g(C)$ is true in G. C is of the form *saturated*$(p_1) \wedge \ldots \wedge$ *saturated*$(p_n) \wedge B$, and *saturated*$(g(p_i))$ holds if there are no edges between $g(p_i)$ and ports outside $g(L)$ in G – this ensures that no edges will be left dangling in rewriting steps. B is a Boolean expression such that all its variables occur in L.

Let G be a port graph. A *rewrite step* $G \Rightarrow H$ via the port graph rewrite rule $L \Rightarrow_C R$ is obtained by replacing in G a subgraph $g(L)$ by $g(R)$, where g is a morphism from L to G satisfying C, and connecting $g(R)$ to the rest of the graph as indicated by the arrow-node edges in the rule: Any edges arriving

to a port in g(L) connected by a bridge arrow port to R are transferred to the corresponding ports in $g(R)$; edges connecting to ports in $g(L)$ that are connected to a blackhole port in the arrow node are deleted. Wire ports in the arrow node trigger a rewiring: the ports in G that connect to the ports in $g(L)$ associated to a wire port in the arrow node are linked by an edge in the rewritten graph.

A sequence of rewriting steps is called a *derivation*. A *derivation tree* is a collection of rewriting derivations with a common root.

PORGY [14] includes functionality to create port graphs and port graph rewrite rules, and to apply rules to graphs according to user-defined strategies. The functions *Connect* and *Attach* (see Definition 1) are represented as attributes in records (i.e., records contain data attributes, visualisation attributes such as colour or shape, and structural attributes such as Connect and Attach). Rules are displayed as graphs, and edges that run between ports of L, R and the arrow node are coloured red to distinguish them from normal edges. PORGY also provides a visual representation of the rewriting derivations, which can be used to analyse the rewriting system. PORGY's *strategy language* allows us to control the way derivations are generated. We can specify not only the rule to be used in a rewriting step, but also the position where the rule should (or should not) be applied. Formally, the rewriting engine works with *graph programs*.

Definition 2 (Graph Program). *A graph program consists of a* located port graph*, a set of port graph rewriting rules, and a strategy expression. A* located port graph *is a port graph with two distinguished subgraphs: a* position *subgraph and a* banned *subgraph, denoted* G^P_Q*. Rewrite rules can only be applied to* G *if they match a subgraph which superposes* P *and does not superpose* Q*.*

We briefly describe below the strategy constructs that we use in our programs (see [14] for more details). The keywords `crtGraph`, `crtPos`, `crtBan` denote, respectively, the current graph being rewritten and its Position and Banned subgraphs. For example, the strategy expression `setPos(crtGraph)` sets the position graph as the full current graph. If T is a rule, then the strategy `one`(T) randomly selects one possible redex for rule T in the current graph G, which should superpose the position subgraph P and not overlap the banned subgraph Q. This strategy fails if the rule cannot be applied. Constants `id` and `fail` denote success and failure, respectively. `while`$(S)[(n)]$`do`(S') executes strategy S' (not exceeding n iterations if the optional parameter n is specified) while S succeeds. `repeat`$(S)[max\ n]$ repeatedly executes a strategy S, not exceeding n times. It can never fail (when S fails, it returns `id`).

3 Port Graphs for Database Modelling

First, we define a visual domain specific language (VDSL) for logical design of relational databases. It includes a class of attributed port graphs to represent objects of a relational database, and a language to specify rewrite rules and strategies for those graphs. We also define Database Port Graphs, to represent a relational database schema $\mathcal{DB} = (\mathcal{R}(\mathcal{A}), \mathcal{FD}, \mathcal{CK})$ (see section 2.1).

3.1 A Visual Domain Specific Language for Database Modelling

The visual building blocks of the language correspond to those of relational databases. Port graph nodes will have an attribute DbType whose value indicates the role of the node. To avoid confusion, we will use Proper Case for relational database concepts (e.g., Attribute) and lower case for port graph concepts (e.g., attributed port graph).

We note here that an Attribute can occur in multiple relations. To this end, we can define a conceptual attribute node. Then we can define an attribute occurrence node to distinguish between appearances in different Relations. In this work, from now on, we only use occurrences and we call them Attribute.

Definition 3 (Relational Database Port Graph VDSL, RDPG-VDSL). *A* Relational Database Port Graph VDSL *is an attributed port graph* $G_{RDB} = (V, P, E, D)_{\mathcal{F}}$*, such that* V *includes the following disjoint sets of nodes (as well as application specific nodes):*

- V_R*: relation nodes (*DbType *= REL);*
- V_A*: attribute nodes (*DbType *= ATTR);*
- V_{FD}*: functional dependency nodes (*DbType *= FD);*
- V_{CK}*: candidate key nodes (*DbType *= CK).*

P includes the following disjoint sets of ports:

- P_{ATT}*: contained attribute ports pATT;*
- P_{REL}*: parent relation ports pREL;*
- P_{DA}*: dependency attribute ports pFD;*
- P_{FD}*: functional dependency ports pFDLHS and pFDRHS;*
- P_{CK}*: relation candidate key ports pCK;*
- P_{KEY}*: (candidate) key attribute ports pKEY.*

and the functions Attach and Connect are such that:

- *if* $p \in P_{ATT}$*,* $Attach(p) \in \{V_R \cup V_{CK}\}$*;*
- *if* $p \in P_{REL}$*,* $Attach(p) \in \{V_A \cup V_{CK}\}$*;*
- *if* $p \in P_{DA}$*,* $Attach(p) \in V_A$*;*
- *if* $p \in P_{FD}$*,* $Attach(p) \in V_{FD}$*; each node in* V_{FD} *has two ports, pFDLHS and pFDRHS;*
- *if* $p \in P_{CK}$*,* $Attach(p) \in V_R$*;*
- *if* $p \in P_{KEY}$*,* $Attach(p) \in V_A$*.*

Connect includes the following pairs of ports (and associated edges):

- Functional Dependency: (pFD, pFDLHS) and (pFDRHS, pFD)*, where pFD* $\in P_{DA}$ *and pFDLHS, pFDRHS* $\in P_{FD}$*. Given a dependency* $\varphi : X \to A$ *the pFD port of every attribute node corresponding to* X *will be connected to the pFDLHS port of the dependency node corresponding to* φ *and the pFDRHS port of the FD node representing* φ *will be connected to the pFD port of the attribute node representing* A*.*

- Attribute in relation: (pATT, pREL), *where* $pATT \in P_A$ *and* $pREL \in P_{REL}$. *Given a relation* R_i *and its attribute* A*, the* $pATT$ *port of the node representing* R_i *will be connected to the* $pREL$ *port of the node representing* A.
- Attribute in candidate key: (pATT, pKEY), *where* $pATT \in P_A$ *and* $pKEY \in P_{CK}$. *Given a candidate key* CK_i *and every attribute* $A_j \in CK_i$*, the* $pKEY$ *port of the node corresponding to* A_j *will be connected to the* $pATT$ *port of the node corresponding to* CK_i.
- Candidate Key of Relation: (pREL, pCK), *where* $pREL \in P_{REL}$ *and* $pCK \in P_{CK}$. *Given a relation* R_i *and its candidate key* CK_j*, the* pCK *port of the node representing* R_i *will be connected to the* $pREL$ *port of the node representing* CK_j.

As a particular case of the above defined class, we now define the Database Port Graph (DBPG) that represents $\mathcal{DB} = (\mathcal{R}(\mathcal{A}), \mathcal{FD}, \mathcal{CK})$. Most importantly, we constrain that one Relation is represented by only one relation node and similarly, one FD is represented by only one FD node. Also, each Attribute occurrence is represented by one attribute node. This design decision is based on the separation of concerns principle.

Definition 4 (Database Port Graph, DBPG). *A* Database Port Graph *is an RDPG such that the following constraints are satisfied:*

- V_R*: one node* DbType = REL *per Relation schema in* $\mathcal{R}$*;*
- V_A*: one node* DbType = ATTR *per Attribute occurrence in any of the* R_i*;*
- V_{FD}*: one node* DbType = FD *per Functional Dependency in* $\mathcal{FD}$*;*
- V_{CK}*: one node* DbType = CK *per Candidate Key in* $\mathcal{CK}$.

The Functional Dependency Port Graph (FDPGs) [27] is a particular case of the above defined Database Port Graph, with Attribute and FD nodes only. An example FDPG is given in Figure 10 in Appendix E.

3.2 Variadic Rewriting Rules

To deal with functional dependencies of various arities, previous works used multiple rules [27] or internal data structures (e.g. compound node in [5]). Here, we present an extension to the port graph rewriting rule language, called variadic rewriting rules (VRRs), inspired by Variadic Interaction Nets [18]. A variadic rule represents a family of rules that differ only in the number of times a subgraph is repeated. First, we propose a container structure that clearly identifies in a port graph the subgraph that will be repeated.

Definition 5 (Pattern Container). *A* pattern container *is a subgraph within a port graph such that if an edge links two ports that belong to the container, the edge also belongs to the container.*

A pattern container has two attributes: a name*, and a* multiplicity *that specifies the maximum number of times the encapsulated pattern will be repeated.*

Edges that connect a port in a pattern container and a port in the outside graph are called variadic edges*. Variadic edges also have an attribute* multiplicity *to control the number of repetitions.*

Definition 6 (Variadic Port Graph Rewrite Rule, VRR). *A* variadic port graph rewrite rule*, denoted $L \Rightarrow^{\mathcal{V}} R$, is a port graph rewrite rule with at least one* pattern container *on the LHS. Multiple pattern containers must not overlap. Pattern container names must be unique on the LHS.*

Given a VRR, we obtain its family of rules by running the Expansion algorithm defined below.

Definition 7 (Variadic Pattern Expansion). *For each pattern container, we generate i copies, where i is the value of the* multiplicity *attribute, as follows:*

1. ***Synchronized Expansion:***
   If a pattern container is present on both sides of a VRR (i.e. their names are identical), then the pattern is expanded in an iterative way on both sides of the rule until i number of copies of the encapsulated subgraph are generated. If variables are used in attributes then a different variable should be used in each copy. The expansion iterator works pairwise; that is, not all combinations of expansions are generated on LHS and RHS, but only the same number of repetitions on the two sides.
2. ***LHS-only Expansion:***
   If the pattern container is defined on LHS only, the expansion happens on LHS only, in the same iterative way.

If multiple patterns are defined, they are expanded independently, i.e. all combinations are generated, by nested iteration. Generally, the order in which the combinations are generated, does not matter.

A variadic edge is expanded based on the value j of its multiplicity attribute. If it equals the multiplicity i of the container it belongs to, then it is fully *expanded, i.e. it is created in all i instances of the container. A* partially *expanded variadic edge ($j < i$) is only created in the first $1 \ldots j$ instances. In other words, in the* n*th iteration of the expansion of the pattern container the variadic edge belongs to, if $n < j$ then n copies are created; otherwise j copies are created.*

In PORGY, which does not have a mechanism to define variadic rules, Definition 7 can be implemented as a macro expansion. The pattern container is visually represented by an enclosing rectangle (a metanode): attributes *name* and i are displayed at the top of the rectangle; and, on the LHS, a + sign in its upper-right corner as shown in the example below (Figures 1 and 2). Fully expanded variadic edges are also marked with a + sign over them and partially expanded variadic edges have the attribute value j displayed over them.

In this example, because the pattern appears in both sides, the expansion will generate a rule version with one node Y, another with 2 and another with 3, we show in the picture just the version corresponding to 3. If the node Y has an attribute a whose value is an expression containing variables, for example x, then each copy of the node Y will have attribute a with values x_1, x_2, x_3.

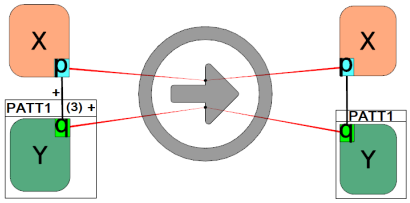


Fig. 1: VRR example

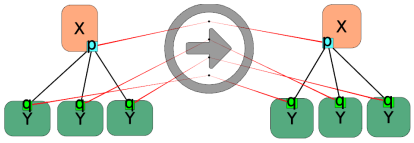


Fig. 2: VRR example expanded, $i = 3$

4 Computing the Syntactic Closure of Σ

Given an FDPG representing set of functional dependencies Σ, we compute its syntactic closure by applying the rules Reflexivity, Augmentation and Transitivity, defined below, controlled by Strategy 1: Syntactic Closure. From now on, we colour-code nodes, as a visual aid. Attribute nodes are green, FD nodes are purple and ports are dark blue. We use other colours for highlighting purposes.

4.1 Rewriting Rules

In the rules below, $x, y, \ldots$ represent name variables for attribute nodes, and $f1, f2, \ldots$ are name variables for FD nodes.

Augmentation. The Augmentation rule (see Figure 3) finds every attribute node y that doesn't have an edge into the FDLHS port of $f1$, regardless of what other attributes are connected there already. We find all non-connected attribute nodes by using the anti-edge feature [26] of PORGY. An anti-edge is represented by a grey double line in the rule editor. The matching algorithm deems the candidate sub-graph isomorphic if no edge is to be found between the two ports connected by the anti-edge.

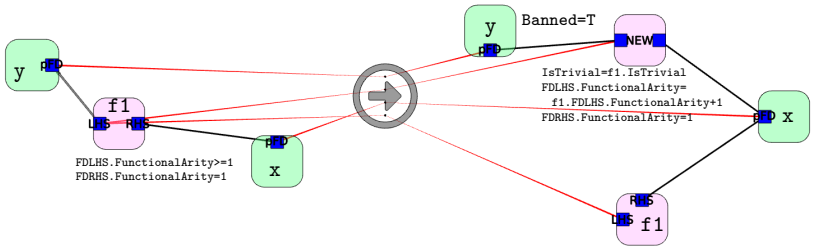


Fig. 3: Augmentation rule.

The rule creates a new FD node and assigns all pre-existing attributes and y to the left side of $f1$. The original $f1$ dependency node is also kept. We use bridge ports (red edges) to keep and copy the already existing edges into the FDLHS ports of $f1$ and *NEW*. If $f1$ was trivial then the new dependency will also be marked trivial. The rule also increases the *FunctionalArity* counter by 1 indicating that a new attribute is connected to the FDLHS port.

Reflexivity. The Reflexivity rule (Figure 4) applies to a node representing the attribute x and generates a trivial dependency $x \to x$. Then the attribute node x is banned so the rule cannot apply again on the same attribute.

The red edges in the arrow node indicate that when applying the rule, any edges connected to the **pFD** port of x in the left-hand side should be transferred to the corresponding **pFD** port of x in the right-hand side.

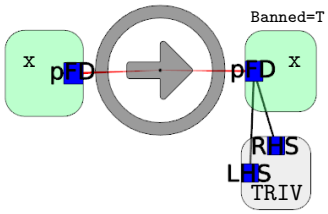


Fig. 4: Reflexivity rule.

Transitivity. A family of Transitivity rules was described in [27] to detect transitive functional dependency chains $f_1 : X \to Y$ and $f_2 : Y \to A$. Instead, here we provide a compact representation of the transitivity axiom in the form of a variadic rule, shown in Figure 5. This rule subsumes the family of Transitivity rules used in previous work.

As mentioned in Section 3.2, $|Y| = k \geq 1$ means that f_1 turns into a set of dependencies $f_1^1 \dots f_1^k$. The connections between the pFD ports of X attribute nodes and the pFDLHS ports of $f_1^1 \dots f_1^k$ nodes have to be preserved as well as copied onto the pFDLHS port of the newly created FD node (called NEW in Figure 5). We achieve this without needing to include the attribute nodes representing X in the rule, thanks to the bridge ports of the arrow node and the connecting red edges, as explained in Definition 6. Then, to cover all cases, we define a VRR pattern over f_1 and Y, with $i = 1 \dots k$. By definition the bridge ports, red edges and the normal edges into y_1.pFD, $\dots$, y_k.pFD will be repeated during the expansion.

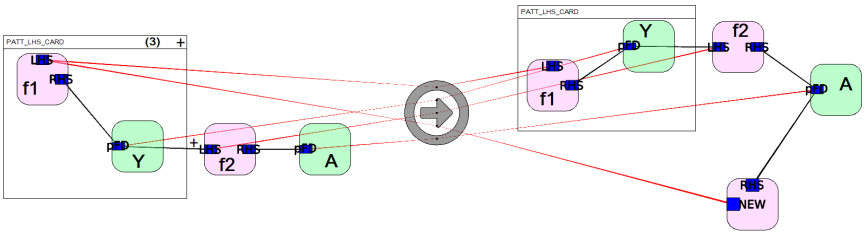


Fig. 5: Variadic Transitivity rule.

We show an example expansion of the Transitivity VRR to Transitivity-3, i.e. with 3 repetitions, on Figure 11 in Appendix E. FD nodes are labelled by records containing an attribute UID that uniquely identifies the Functional Dependency, except for trivial dependencies that are all given UID = 1. Nodes representing non-trivial FDs are given a prime number as UID. This offers extra, domain-specific backtracking functionality for dependencies, as explained below.

Note that the Reflexivity, Augmentation and Transitivity rules never remove the matching subgraph. Therefore, these rules could run for ever. To prevent this, we use conditional rules and focusing constructs in Section 4.2 to define the Syntactic Closure strategy. To ensure that the iteration of the Transitivity rule terminates when no new transitive dependencies can be inferred, we use

the UID attribute of FD nodes. When the Transitivity rule creates a transitive dependency node, it multiplies the UIDs of the contributing FDs and assigns the result as UID of the new FD node. We forbid the application of the rule if a node already exists with that UID (using `NotNode()` in the rule condition).

4.2 Syntactic Closure Strategy

The Strategy 1: Syntactic Closure applies first the Reflexivity rule as much as possible in the current graph. Each application bans an Attribute node, which ensures termination since matching is not allowed on banned nodes.

In lines 5-6, we set the Position subgraph to be the whole graph and the Banned subgraph to empty. Then, while there is at least one FD node the Augmentation rule hasn't *visited* and *iterated*, `one`(*AugIterOn*) sets *AugIter* and *AugVisit* flags to true on a randomly selected FD node. The Augmentation rule will be applied on this FD node and all attribute nodes that are not connected to the FDLHS port of said FD node. Every attribute node used by this rule is banned to prevent re-application. Once all possible applications are processed, *AugIter* flag is set to false and the Banned subgraph to empty. The iteration proceeds to the next, not yet visited, FD node. All new FD nodes, created by the Augmentation rule, are assigned a unique UID using the `update()` construct to call the function, `GenerateNextPrime()`, using PORGY's Python API.

Strategy 1: Syntactic Closure

```
1  //——— Reflexivity ———
2  setPos(all(crtGraph));
3  repeat(one(Reflexivity));
4  //——— Augmentation ———
5  setPos(all(crtGraph));
6  setBan(all([emptySet]));
7  while(match(AugIterOn))do(
8    one(AugIterOn);
9    repeat(one(Augmentation));
10   one(AugIterOff);
11   setBan(all([emptySet]))
12 );
13 update("GenerateNextPrime"{result : UID});
14 //——— Transitivity ———
15 while(match(IterOn))do(
16   one(IterOn);
17   repeat(one(Transitivity);#Augmentation#);
18   update("GenerateNextPrime"{result : UID});
19   one(IterOff)
20 );
21 repeat(one(ResetVisitedFlags));
22 //——— Cleanup ———
23 repeat(one(Cleanup))
```

Next, we compute transitive dependencies (lines 15-21), calling the Augmentation strategy (lines 5-12) after each application of the Transitivity variadic rewriting rule in line 17.

Since the rewrite rules may generate functional dependencies that already exist in the graph (despite the condition in the Transitivity rule), we add CleanUp rules to remove duplicates: see the Cleanup VRR in Figure 12 and an example expansion in Figure 13, both in Appendix E.

4.3 Example of application

Our strategy computed the syntactic closure of $\Sigma = \{AB \to C, ABC \to D\}$; the resulting FDPG can be seen on Figure 6.

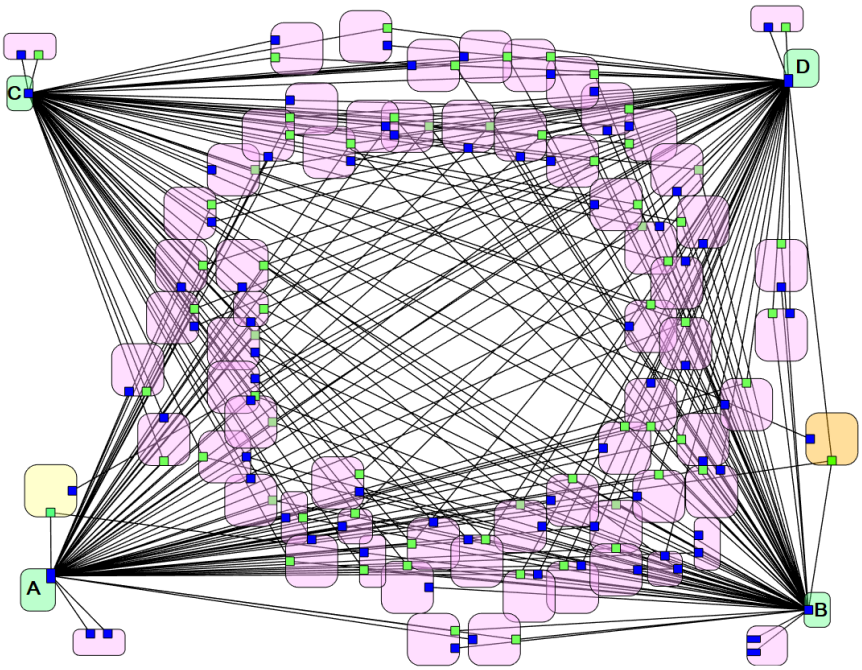


Fig. 6: Syntactic closure of $\Sigma = \{AB \to C, ABC \to D\}$.

Attributes A, B, C and D and their trivial dependencies can be seen in the four corners of the graph. As an example, we highlighted two FD nodes. The first one, in orange on the right hand side of the image, represents the dependency $ABD \to C$ which was created by augmenting $AB \to C$ with D. The second one represents $AB \to D$. This FD, shown on the left side of Figure 6 in yellow, was found by the Transitivity rule, matching on dependencies $A \to A, B \to B, AB \to C$ and $ABC \to D$.

4.4 Visual Analysis of the Closure

We now turn our attention to usual questions about Σ^+. For example, using the derivation tree in PORGY, it is possible to track how and when a particular dependency was generated: If we alter the colour of any FD node in a leaf node of the derivation tree, PORGY will back-propagate this change up the tree. This way, we can identify the exact step where the FD was created, and by zooming on the edges of the derivation tree we can see which of Armstrong's axioms generated the dependency.

Strategy 2: Membership Problem

```
setPos(all(
property(crtGraph, port,DbType == "FDLHS" && FunctionalArity == 3)
∩ ngb(property(crtGraph, node,DbType == "ATTR"
        && viewLabel == "A"), edge,DbType == "L")
∩ ngb(property(crtGraph, node,DbType == "ATTR"
        && viewLabel == "B"), edge,DbType == "L")
∩ ngb(property(crtGraph, node,DbType == "ATTR"
        && viewLabel == "D"), edge,DbType == "L")
∩ ngb(property(crtGraph, node,DbType == "ATTR"
        && viewLabel == "C"), edge, DbType == "R") ) ); //end setPos
(isEmpty(crtPos))orelse(repeat(one(Highlight)))
```

Another important question in database design is the *Membership Problem* [8]: given a set of FDs Σ, and an FD, φ, determine if $\varphi \in \Sigma^+$. Two groups of algorithms were developed to solve the membership problem: 1. generate a syntactic closure and check if $\varphi : X \to A$ is in it, or 2. compute the closure of X, X^+ and check if A is in it. Following the first approach, we can solve the problem by running a strategy to find and highlight the FD node that represents φ in the syntactic closure, if it exists, and fail if $\varphi \notin \Sigma^+$. For example, Strategy 2 was used to find the dependency $ABD \to C$, highlighted in Figure 6 (due to space constraints, we refer the reader to [14] for explanations of the constructs used). Following the second approach, we can simply use Strategy 1 but focusing on the set X of attributes. We only need to replace the expression `setPos(all(crtGraph))` with one that describes X, then the rewriting steps will apply on the attribute set in question. For example, if $X = B$, we write `setPos(all(property(crtGraph, node, DbType ==”ATTR” && viewLabel==”B”)))`.

4.5 Correctness

The strategic program $Cl = [\mathcal{F}, closure]$ consisting of an initial port graph $\mathcal{F}$ representing a set of functional dependencies Σ, and the syntactic closure strategy defined in Strategy 1, correctly computes the syntactic closure Σ^+. Proofs of the propositions stated below are given in the Appendix.

Proposition 1 (Termination). *For any initial FDPG $\mathcal{F}$, the strategic program $Cl = [\mathcal{F}, closure]$ terminates.*

To prove the correctness of our program, we first show that the three rules Reflexivity, Augmentation and Transitivity are sound and complete, that is, given Σ, we can compute Σ^+ by using these three rules.

Proposition 2 (Soundness and Completeness of the Rules). *The Reflexivity, Augmentation and Transitivity rules stated below are sound and complete:*

1. *Reflexivity: for any attribute A, $A \to A$.*
2. *Augmentation: If $X \to A$ then $XY \to A$ for any attribute A and sets X, Y of attributes.*
3. *Transitivity: If $X \to A_i$ $(1 \leq i \leq n)$ and $A_1, \ldots, A_n \to B$ then $X \to B$.*

Since the Reflexivity, Augmentation and Transitivity port graph rewriting rules implement the rules stated in Proposition 2, to prove that Cl is sound and complete it suffices to show that any sequence of applications of these three rules can be transformed into a sequence in the order defined by the closure strategy.

Definition 8 (Canonical Form). *A sequence of applications of Reflexivity, Augmentation and Transitivity is in canonical form if it consists of applications of Reflexivity, followed by Augmentation, followed by Transitivity and Augmentation:* $(Reflexivity)^*(Augmentation)^*(Transitivity; Augmentation^*)^*$

Proposition 3 (Soundness and Completeness of the Strategy). *Canonical sequences are sound and complete.*

5 Finding a Minimal Cover

To generate a Minimal Cover (see Section 2 for the definition) we have to ensure that there are no extraneous attributes on FD left sides and there are no redundant FDs (we already have singleton right sides). Standard algorithms check this by running the Membership algorithm on an altered Σ: remove $X \to A$ and replace it with $X \setminus B \to A$. If this still yields the same Σ^+ then B is extraneous in X. Instead, since we have Σ^+ at hand, it suffices to check if there exists $Z \subset X \to A$, for all proper subsets Z. We use a variadic rewriting rule (Section 3.2) to specify a family of rules for every possible subset pair (n, k), where $n = |X|$ and $k = |Z|$. Since $Z \subset X$, we make use of the partially expanded variadic edge feature by restricting one variadic edge to only k expansions. We show the variadic rule (parameterised already with $n = 3, k = 2$) in Figure 7 (and an expansion in Figure 14 in Appendix E).

Next, we have to remove redundant FDs. A functional dependency $\varphi : X \to A$ is redundant, if $(\Sigma \setminus \varphi)^+ = \Sigma^+$ [21]. Previously published algorithms detected this by running a Membership check on $(\Sigma \setminus \varphi)$ to see if it yields φ. We note that in an FDPG representing $(\Sigma \setminus \varphi)^+$ there is an FDPG-Path from X to A. This path exists as an FD node created by the Syntactic Closure strategy, and

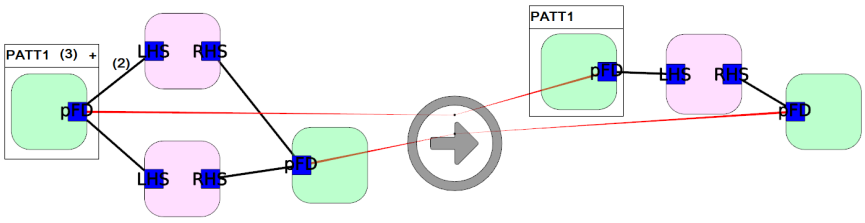


Fig. 7: Extraneous variadic rule.

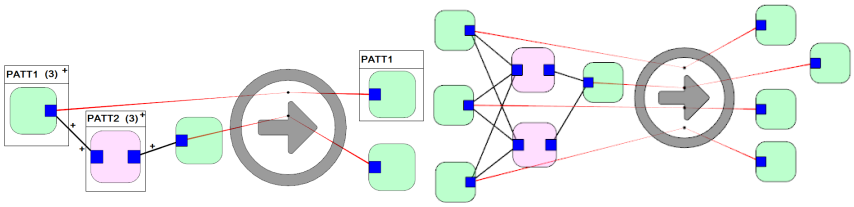


Fig. 8: Nonredundancy VRR.

Fig. 9: Nonredundancy rule, $i_1 = 3, i_2 = 2$.

if a dependency can be inferred in multiple ways, it is present multiple times. Since $|X| \geq 1$, we use a VRR to detect and remove the redundant FD nodes. We present the rule and an expansion in Figures 8 and 9.

Using the Extraneous and Nonredundancy rules, Strategy 3 computes a Minimal Cover. We reuse the Syntactic Closure strategy, but without the Clean Up rules. The Nonredundancy rule gets rid of duplicates. Lastly, we remove trivial dependencies (FD nodes where UID = 1), as they are not in the minimal cover.

Strategy 3: Minimal Cover

```
1 #Syntactic Closure without Cleanup#
2 setPos(all(crtGraph)); setBan(all([emptySet]));
3 repeat(one(Extraneous)); repeat(one(Nonredundancy));
4 repeat(one(RemoveTrivial));
```

6 Conclusion and Future Work

We introduced variadic rewriting rules and used these rules to define strategies that compute and analyse the syntactic closure of a set of Functional Dependencies. We have shown that these strategies are terminating, sound and complete. We have also defined additional rules and a strategy to compute Minimal Covers. A minimal cover is the input of algorithms to find candidate keys [24] and of Bernstein's 3NF Synthesis Algorithm [10]. The strategies that find these will make use of the already defined CK and Relation nodes. Furthermore, 3NF Relations will require the introduction of the notion of Foreign Key.

References

1. Abrial, J.: Data semantics. In: Klimbie, J.W., Koffeman, K.L. (eds.) Data Base Management, Proceeding of the IFIP Working Conference Data Base Management, Cargèse, Corsica, France, April 1-5, 1974. pp. 1–60. North-Holland (1974)
2. Andrei, O.: Rewriting Calculus for Graphs: Applications to Biology and Autonomous Systems. Ph.D. thesis, Institut National Polytechnique de Lorraine, Nancy, France (November 2008)
3. Andrei, O., Fernández, M., Kirchner, H., Melançon, G., Namet, O., Pinaud, B.: Porgy: Strategy-driven interactive transformation of graphs. In: Echahed, R. (ed.) TERMGRAPH. EPTCS, vol. 48, pp. 54–68 (2011)
4. Armstrong, W.W.: Dependency structures of data base relationships. In: Information processing 74: proceedings of IFIP Congress 74. pp. 580–583. North-Holland, Amsterdam (1974)
5. Ausiello, G., D'Atri, A., Saccà, D.: Graph algorithms for functional dependency manipulation. J. ACM **30**(4), 752–766 (1983). https://doi.org/10.1145/2157.322404
6. Badia, A., Lemire, D.: A Call to Arms: Revisiting Database Design. SIGMOD Rec. **40**(3), 61–69 (Nov 2011). https://doi.org/10.1145/2070736.2070750
7. Batini, C., D'Atri, A.: Rewriting systems as a tool for relational data base design. In: Claus, V., Ehrig, H., Rozenberg, G. (eds.) Graph-Grammars and Their Application to Computer Science and Biology, International Workshop, Bad Honnef, October 30 - November 3, 1978. Lecture Notes in Computer Science, vol. 73, pp. 139–154. Springer (1978). https://doi.org/10.1007/BFb0025717
8. Beeri, C., Bernstein, P.A.: Computational problems related to the design of normal form relational schemas. ACM Trans. Database Syst. **4**(1), 30–59 (1979)
9. Beeri, C., Fagin, R., Howard, J.H.: A complete axiomatization for functional and multivalued dependencies in database relations. In: Smith, D.C.P. (ed.) SIGMOD Conference. pp. 47–61. ACM (1977)
10. Bernstein, P.A.: Synthesizing third normal form relations from functional dependencies. ACM Trans. Database Syst. **1**(4), 277–298 (1976)
11. Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P., Ringeissen, C.: An overview of ELAN. Electr. Notes Theor. Comput. Sci. **15**, 55–70 (1998). https://doi.org/10.1016/S1571-0661(05)82552-6, `https://doi.org/10.1016/S1571-0661(05)82552-6`
12. Ehrig, H., Engels, G., Kreowski, H., Rozenberg, G. (eds.): Handbook of graph grammars and computing by graph transformation: Applications, Languages and Tools, vol. 2. World Scientific (1999)
13. Embley, D.W., Mok, W.Y.: Mapping Conceptual Models to Database Schemas. In: Embley, D.W., Thalheim, B. (eds.) Handbook of Conceptual Modeling, vol. XIX, pp. 123–164. Springer (2011)
14. Fernández, M., Kirchner, H., Pinaud, B.: Strategic Port Graph Rewriting: an Interactive Modelling Framework. Mathematical Structures in Computer Science pp. 1–48 (Aug 2018). https://doi.org/10.1017/S0960129518000270, `https://hal.inria.fr/hal-01251871`
15. Fernández, M., Kirchner, H., Pinaud, B., Vallet, J.: Labelled graph strategic rewriting for social networks. J. Log. Algebr. Meth. Program. **96**, 12–40 (2018). https://doi.org/10.1016/j.jlamp.2017.12.005, `https://doi.org/10.1016/j.jlamp.2017.12.005`

16. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database systems - the complete book (2. ed.). Pearson Education (2014)
17. Jahnke, J.H., Zündorf, A.: Applying Graph Transformations to Database re-engineering. In: Ehrig et al. [12], pp. 267–286
18. Jiresch, E.: Extending the interaction nets calculus by generic rules. In: Alves, S., Mackie, I. (eds.) Proceedings 2nd International Workshop on Linearity, LINEARITY 2012, Tallinn, Estonia, 1 April 2012. EPTCS, vol. 101, pp. 12–24 (2012). https://doi.org/10.4204/EPTCS.101.2, https://doi.org/10.4204/EPTCS.101.2
19. Kalleberg, K.T.: Stratego: a programming language for program manipulation. ACM Crossroads **12**(3), 4 (2006). https://doi.org/10.1145/1144366.1144370, https://doi.org/10.1145/1144366.1144370
20. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theor. Comput. Sci. **109**(1&2), 181–224 (1993). https://doi.org/10.1016/0304-3975(93)90068-5, https://doi.org/10.1016/0304-3975(93)90068-5
21. Maier, D.: Minimum Covers in Relational Database Model. J. ACM **27**(4), 664–674 (1980). https://doi.org/10.1145/322217.322223
22. Maier, D.: The Theory of Relational Databases. Computer Science Press (1983)
23. Plump, D.: The design of GP 2. In: Escobar, S. (ed.) Proceedings 10th International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2011, Novi Sad, Serbia, 29 May 2011. EPTCS, vol. 82, pp. 1–16 (2011). https://doi.org/10.4204/EPTCS.82.1, http://dx.doi.org/10.4204/EPTCS.82.1
24. Saiedian, H., Spencer, T.: An efficient algorithm to compute the candidate keys of a relational database schema. Comput. J. **39**(2), 124–132 (1996). https://doi.org/10.1093/comjnl/39.2.124
25. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES Approach: Language and Environment. In: Ehrig et al. [12], pp. 551–603
26. Vallet, J.: Where Social Networks, Graph Rewriting and Visualisation Meet: Application to Network Generation and Information Diffusion. Ph.D. thesis, University of Bordeaux, France (2017), https://tel.archives-ouvertes.fr/tel-01691037
27. Varga, J.: Finding the Transitive Closure of Functional Dependencies using Strategic Port Graph Rewriting. In: Fernández, M., Mackie, I. (eds.) Proceedings Tenth International Workshop on Computing with Terms and Graphs, Oxford, UK, 7th July 2018. Electronic Proceedings in Theoretical Computer Science, vol. 288, pp. 50–62. Open Publishing Association (2019). https://doi.org/10.4204/EPTCS.288.5

Appendix A Proof of Proposition 1

Proposition (Termination)**.** *For any initial FDPG $\mathcal{F}$, the strategic program $Cl = [\mathcal{F}, closure]$ terminates.*

Proof. Since the strategy applies three strategies sequentially (Reflexivity, Augmentation and Transitivity), it is sufficient to show that each of them terminates. For this, we show that for each of them there is a measure which is strictly decreasing with respect to a well-founded ordering.

Reflexivity Strategy: The measure in this case is the number of non-banned nodes in the graph. Each application of the reflexivity rule (Figure 4 strictly decreases the number of non-banned nodes. Since the rule can only apply to non-banned nodes, the iteration defined in line 3 of the closure strategy terminates.

Augmentation Strategy: Two looping constructs are used in this strategy, a while-loop starting in line 7 and a repeat-loop in line 9. The while-loop is controlled by a condition `match`($AugIterOn$), which requires a non-visited node to succeed. The rule AugIterOn applied in line 8 sets the AugIter and AugVisit flags to true on a randomly selected FD node that has AugVisit flag = False. Each application of the Augmentation rule in the loop in line 9 bans the attribute node used, so the the repeat loop terminates, and afterwards the AugIter flag is set to false but the AugVisit flag remains untouched. This means that each iteration of the while loop in line 7 decreases the number of non-visited nodes, thus the strategy terminates.

Transitivity Strategy: This strategy, without the calls to the Augmentation strategy in line 17 and line 23, was shown to be terminating in [27]. We use the same measures, since they are not affected by the call to the Augmentation strategy: For `repeat`(`one`($Transitivity_k$); $\#Augmentation\#$), the measure is the number of possible matches of the LHS subgraph of f1. This is a good measure because a) the Rule Condition on UID in $Transitivity_k$ prevents reapplication of the rule to the same nodes and b) even though the NEW node is added in the right-hand side, it is not part of the LHS subgraph of f1. Similarly, the nodes added by Augmentation are not connected to f1. By the time we call the Augmentation strategy from Transitivity, f1 will have been augmented and its AugVisit flag is permanently set to True. The only FD node Augmentation will be able to touch at this point is NEW.

For `while`(`match`($IterOn$))`do`(...) the measure is $|V_{FD}|_{G0} + |\Sigma^+| - |V_{FD}|_{Gi}$. That is, the number of FD nodes in the initial graph plus the size of the closure less the number of FD nodes after the i-th application of the loop. With one successful application of the Transitivity rule, the number of FD nodes increases therefore the measure decreases.

Termination of the CleanUp rule iteration is straightforward, since these rules decrease the size of the graph. □

Appendix B Proof of Proposition 2

Proposition (Soundness and Completeness of the Rules)**.** *The Reflexivity, Augmentation and Transitivity rules stated below are a sound and complete:*

1. *Reflexivity: for any attribute A, $A \to A$.*
2. *Augmentation: If $X \to A$ then $XY \to A$ for any attribute A and sets X, Y of attributes.*
3. *Transitivity: If $X \to A_i$ $(1 \leq i \leq n)$ and $A_1, \ldots, A_n \to B$ then $X \to B$.*

Proof. First, we observe that the rules are sound: they are particular cases of the axioms $A1$-$A3$ given in Section 2, which are sound and complete. Hence, it is sufficient to prove that the rules permit us to derive the axioms $A1$-$A3$.

Remark. Our rules assume that right sides of functional dependencies consist of only one attribute. Since the Union and Decomposition Axioms can be derived from $A1$-$A3$ and by them one can infer $X \to YZ$ from $X \to Y$ and $X \to Z$, and reciprocally, from $X \to YZ$ one can infer $X \to Y$ and $X \to Z$, it is sufficient to consider single attributes in right-hand sides of dependencies.

Axiom $A1$ can be derived as follows: Let $Y = \{A_1 \ldots A_n\}$ and assume $Y \subseteq X$. Using the Reflexivity rule, we can derive $A_i \to A_i$ and using Augmentation, we derive $A_i X \to A_i$ for each $A_i \in Y$. This is sufficient as explained in the remark above.

Axiom $A2$ can be derived as follows: Let
$Y = \{A_1 \ldots A_n\}$,
$Z = \{A_n, A_{n+1}, \ldots A_p\}$ and
$W = \{A_n, A_{n+1}, \ldots A_p, A_{p+1}, \ldots, A_k\}$
and assume $X \to Y$, that is, $X \to A_i$ for each $A_i \in Y$ as explained in the remark above. By repeated applications of the Augmentation rule, we derive $XW \to A_i$ for each $A_i \in Y$. Note that by Reflexivity, we can derive $A_j \to A_j$ for any $A_j \in Z$ and by repeated applications of Augmentation, we obtain $XW \to A_j$, since $A_j \in W$. Therefore $XW \to A_i$ for each $A_i \in YZ$, as required.

Axiom $A3$ is derived using the Transitivity rule: Assume $X \to Y$, that is, $X \to A_i$ for each $A_i \in Y$, and $Y \to B_i$ for each $B_i \in Z$. Using the Transitivity rule, we derive $X \to B_i$, as required. □

Appendix C Proof of Proposition 3

Proposition (Soundness and Completeness of the Strategy)**.** *Canonical sequences are sound and complete.*

Proof. By Proposition 2, Reflexivity, Augmentation and Transitivity are sound and complete. Assume a set Σ of functional dependencies is inferred using a non-canonical sequence S of applications of Reflexivity, Augmentation and Transitivity rules. We show that the steps of application of the rules can be reorder to obtain a canonical sequence that derives the same set of dependencies. Since applications of Reflexivity are independent of other rules, we can reorder the steps, to move all applications of Reflexivity to the start of the sequence, obtaining a sequence $S_1 = S_{Refl}S'$ that derives Σ and such that S_{Refl} contains only applications of the Reflexivity rule, and S' has no applications of Reflexivity.

Similarly, any application of Augmentation in S' on a dependency that exists in Σ or has been obtained by Reflexivity or by a previous Augmentation step can be moved towards the start of S', obtaining a sequence $S_{Aug}S''$ where S_{Aug} consists only of Augmentation steps and all the applications of Augmentation in S'' use a dependency obtained by transitivity.

The sequence $S_{Refl}SAugS''$ is therefore canonical and derives the same set of functional dependencies as S. It follows that canonical sequences are sound and complete. □

Appendix D DBPG and FDPG Examples

The functional dependency graph $\Sigma = \{AB \to C, ABC \to D\}$:

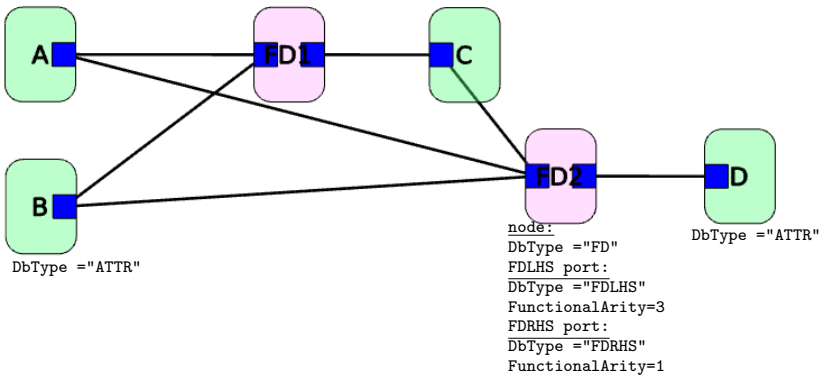


Fig. 10: The functional dependency graph $\Sigma = \{AB \to C, ABC \to D\}$.

Appendix E Variadic Rewriting Rule Expansion Examples

Example expansion of Transitivity VRR to Transitivity-3, where $i = 3$.

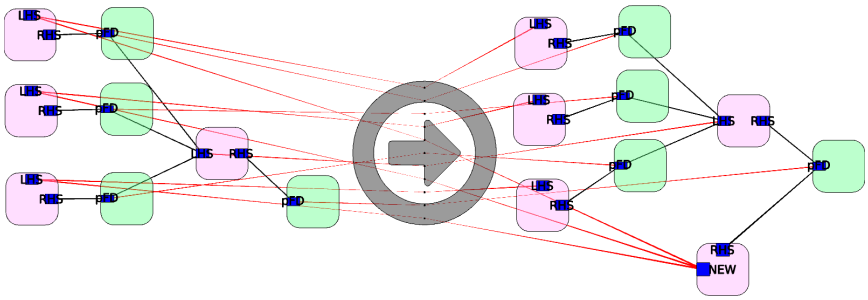


Fig. 11: An example expansion: Transitivity-3 rule.

The Cleanup variadic rewriting rule:

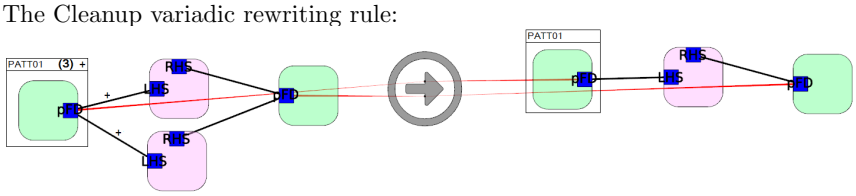


Fig. 12: Cleanup VRR.

Example expansion of Cleanup VRR to $i = 3$:

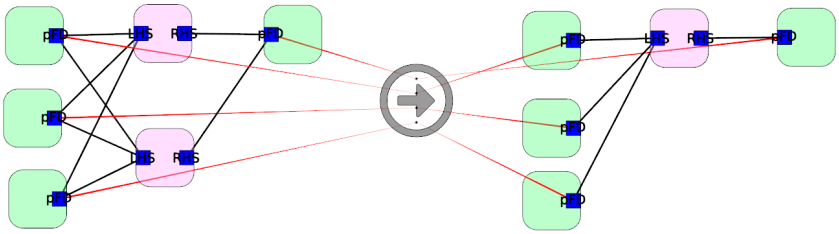


Fig. 13: An expansion of Cleanup VRR: Cleanup-3.

Example expansion of Extraneous VRR to $i = 2, j = 1$:

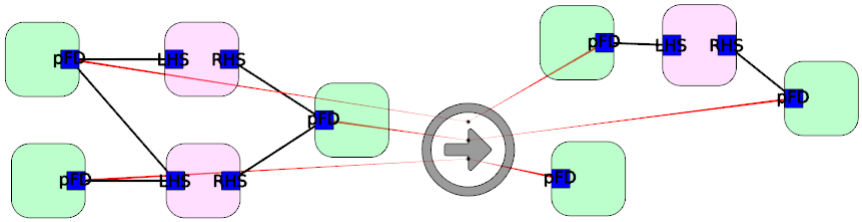


Fig. 14: Expanded extraneous rule, $i = 2, j = 1$.

Generalization-driven semantic clone detection in CLP

Extended abstract

Wim Vanhoof and Gonzague Yernaux

University of Namur
Belgium

Abstract. In this work, which is work in progress, we provide the necessary ingredients for an algorithm capable of searching for semantic clones in CLP program code. Two code fragments are considered semantically cloned (at least to some extent) when they can both be transformed into a single code fragment thus representing the functionality that is shared between the fragments. While the framework of what constitute such semantic clones has been established before, no algorithm exist that effectively performs the search. We discuss how the generalization of CLP goals can be a driving factor both for controlling the search process (i.e. keeping it finite) as for guiding the search (i.e. choosing what transformations to apply at what moment).

1 Introduction and motivation

Clone detection refers to the process of finding source code fragments that exhibit a sufficiently similar computational behavior, independent of them being textually equal or not. Such fragments are often called *clones*. While there is no standard definition of what constitutes a clone [1], in the literature one often distinguishes between four different classes, or types, of clones. The simplest class, sometimes called type-1 clones, refer to code fragments that differ only in layout and whitespace, whereas type-2 and type-3 clones allow for more (syntactical) variation such as renamed identifiers and statements and/or expressions that are different or lacking in one of the fragments. Type-4 clones on the other hand refer to fragments that are *semantically* equivalent, even if the respective source code fragments are quite different and seemingly unrelated [1]. This type of clones, also known as *semantic clones*, is arguably the most interesting albeit the most difficult type to find by automatic analysis.

While detecting semantic clones is an undecidable problem in general, it has applications in different domains such as program comprehension [2–4], plagiarism detection [5] and malware detection [6]. When approximated by program analysis, the resulting knowledge can also be used to drive advanced program transformations such as removal of redundant functionality from source code [7] and the automatic detection of a suitable parallelization strategy for a given code fragment [8, 9]. Unsurprisingly, most current clone detection techniques are

based on somehow comparing the syntactical structure of two code fragments and, consequently, are limited to detecting type-3 clones at best. Examples include the abstract syntax-tree based approaches for Erlang [10] and Haskell [11], as well as our own work [12] in the context of logic programming. Some approaches try to capture the essence of the algorithm at hand such as [13], where algorithms are converted into a system of recurrence equations or [14, 15] where programs are abstracted by means of software metrics and program schemas.

In previous work, we have devised a framework for detecting semantic clones in logic programming [16]. The basic idea in that work is that two predicates are considered semantic clones if they can each be transformed – by a sequence of semantics-preserving program transformations – into a single common predicate definition. This is in line with other approaches towards semantic clone detection [1] where fragments are often considered implementing the same functionality if one can be transformed in the other. This framework was generalized to handle CLP in [17], which is of particular interest since CLP (or constrained Horn clauses in general) has been recognized before as a suitable abstraction to represent algorithmic logic [18]. As such, the framework for detecting semantic clones is lifted to a framework for characterizing *algorithmic equivalence* between the code fragments that were translated into CLP. However, in neither of these works an attempt was made to formulate *how* the search for a suitable series of program transformations could be performed or controlled. The question is far from trivial, given the literally enormous search space involved. In this work in progress, we present some of the main ingredients for an algorithm capable of controlling this search. When concretized, it thus represents a workable decision procedure to test whether two given CLP fragments are (at least partial) semantic clones.

2 Semantic clones: setting the stage

While in practice CLP is typically used over a concrete domain, we will in this work make abstraction of the concrete domain over which the constraints are expressed. A program P is defined as a set of constraint Horn Clause definitions where each clause definition is of the form $p(V_1, \ldots, V_n) \leftarrow G$ with $p(V_1, \ldots, V_n)$ an atom called the head of the clause, and G a goal called the body of the clause. For simplicity we suppose that all arguments in the head are variables (represented, as usual, by uppercase letters) and that all clauses defining a predicate have the same head (i.e. use the same variables to represent the arguments). A goal is a set of atoms and/or constraints. A fact is a clause with only constraints in its body. When we say "a predicate p", it will be clear from the context whether we mean the symbol p or the set of clauses defining p. When the arity of the predicate is relevant, we will use p/n to represent the fact that the predicate p has n arguments.

As usual substitutions, being mappings from variables to terms, will be denoted by Greek letters. The application of a substitution θ to a term t will be represented by $t\theta$ and the composition of substitutions θ and σ will be denoted

$\theta\sigma$. A renaming is a substitution mapping variables to variables. We say that terms t_1 and t_2 are variants, denoted $t_1 \approx t_2$ iff they are equal modulo a bijective variable renaming.

While different semantics have been defined for CLP programs, for the remainder of this paper we can stick to the basic non-ground declarative semantics [19]. However, since the CLP predicates we wish to relate may originate from different sources, they potentially have a different number of arguments and, even if the predicates basically compute the same results, they may use different argument positions for storing what may essentially be the same values. The following definition captures what it means for two such predicates to compute the same result. It states that both predicates must have a subsequence of their argument positions (both sequences having the same size but containing possibly different argument positions and not necessarily in the same order) such that when the predicates are invoked with the corresponding arguments initialized with the same terms, then each predicate computes the same result. This means that for each pair of corresponding argument positions, the terms represented by these arguments must be the same (modulo a variable renaming) both at the moment the predicates are invoked (condition 1 in the definition) and at the moment the predicates return (condition 2 in the definition). As for notation, given a sequence R, we denote by R_i the i'th element of R.

Definition 1. *Given CLP programs P_1 and P_2, let p_s/n_s and p_q/n_q denote predicates in, respectively, P_1 and P_2 and let R and R' denote sequences of argument positions from respectively $\{1, \ldots n_s\}$ and $\{1, \ldots n_q\}$ such that $|R| = |R'| = n$. We say that (p_s, R)* computes in P_1 a subset of *(p_q, R')* in P_2 *if and only if for each call of the form $p_s(V_0, \ldots, V_{n_s})\theta$ with computed answer substitution θ', there also exists a call $p_q(V_0, \ldots, V_{n_q})\sigma$ with computed answer substitution σ' such that the following holds for all $k \in 1 \ldots n$:*

1. $(V_{R_k})\theta \approx (V_{R'_k})\sigma$
2. $(V_{R_k})\theta\theta' \approx (V_{R'_k})\sigma\sigma'$

Moreover, we say that (p_s, R) computes the same in P_1 *as does (p_q, R') in P_2, denoted by $[\![p_s]\!]_R^{P_1} = [\![p_q]\!]_{R'}^{P_2}$ if and only if (p_s, R) computes a subset of (p_q, R') and vice versa in their respective programs.*

The above definition allows us to characterize predicates as computing the same results, even if these predicates only *partially* exhibit the same behavior. Indeed, what matters is that they compute the same values when restricted to the arguments in R, respectively R'. The values computed by arguments *not* comprised in either R or R' are not concerned and may be different. When the programs are clear from the context, we will drop the superscript notation and simply write $[\![p_s]\!]_R = [\![p_q]\!]_{R'}$

Example 1. Consider the predicate $p/3$ computing in its third argument the product of its first two arguments

$$p(A, B, P) \leftarrow B = 1, P = A$$
$$p(A, B, P) \leftarrow B' = B - 1, p(A, B', P'), P = P' + A$$

and $sp/4$ computing in its third and four arguments the sum, respectively, the product of its first two arguments:

$$sp(A, B, S, P) \leftarrow A = 1, S = B + 1, P = B.$$
$$sp(A, B, S, P) \leftarrow A' = A - 1, sp(A', B, S', P'), S = S' + 1, P = P' + B.$$

Note how both predicates share the functionality of computing the product of their first two arguments (although the role of A and B is switched). Therefore, we have that $[\![sp]\!]_{\langle 1,2,4\rangle} = [\![p]\!]_{\langle 2,1,3\rangle}$.

In order to define our notion of semantic clones, we first need to introduce the following notions. Let us consider a given set $\mathcal{R}$ of available program transformations. While our work is parametric with respect to this set $\mathcal{R}$, transformations that would typically be considered are *unfolding* [20], *slicing* [21], argument removal, and so on. We first define the notion of an $\mathcal{R}$-transformation sequence as follows, based on [20].

Definition 2. *Let $\mathcal{R}$ be a set of program transformations and P a CLP program. Then an* $\mathcal{R}$-transformation sequence *of P is a finite sequence of CLP programs, denoted $\langle P_0, P_1, \ldots, P_n\rangle$, where $P_0 = P$ and $\forall i\ (0 < i \leq n) : P_i$ is obtained by the application of one transformation from $\mathcal{R}$ on P_{i-1}.*

Given a predefined set of program transformations $\mathcal{R}$ and CLP programs P and Q, we will often use $P \rightsquigarrow^*_{\mathcal{R}} Q$ to represent the fact that there exists an $\mathcal{R}$-transformation sequence $\langle P_0, P_1, \ldots, P_n\rangle$ with $P_0 = P$ and $P_n = Q$. We are only interested in transformation sequences that preserve the semantics of the original predicate, at least partially, i.e. with respect to a given sequence of argument positions.

Definition 3. *Given a set of program transformations $\mathcal{R}$, predicates p and p', and sequences of argument positions R and R'. A $\mathcal{R}$-transformation sequence $\langle P_0, P_1, \ldots, P_n\rangle$* correctly transforms *$(p, R)$ into (p', R') if and only if (p, R) computes the same result in P_0 as (p', R') in P_n.*

Example 2. Reconsider the definitions from Example 1 and a set $\mathcal{R}$ comprising at least the slicing and argument removal transformations. Then it is not hard to see that there exists an $\mathcal{R}$-transformation sequence that correctly transforms $(sp,\langle 1, 2, 4\rangle)$ into $(p,\langle 2, 1, 3\rangle)$. Indeed, it suffices to remove the third argument (S) from sp and slice away the literals that manipulate S to obtain

$$sp(A, B, P) \leftarrow A = 1, P = B.$$
$$sp(A, B, P) \leftarrow A' = A - 1, sp(A', B, P'), P = P' + B.$$

which is, basically, a variant of p where the role of the first and second argument has been switched. Hence the result.

Definition 3 essentially defines what we will see as a correct transformation sequence: one that preserves the computation performed by a predicate of interest, at least with respect to a subset of its arguments. Note that the definition

is parametrized with respect to the set $\mathcal{R}$ of allowed transformations. Also note that the definition is quite liberal, in the sense that it allows predicates to be renamed, arguments (and thus computations) to be left out of the equation, and arguments to be permuted. We are now in a position to define what we mean for the predicates to be semantic clones, at least with respect to a subset of their computations. The definition is loosely based on the notion of a semantic clone pair [16].

Definition 4. *Let p_1 and p_2 be predicates defined in, respectively the programs P_1 and P_2, and let R_1 and R_2 be sequences of argument positions. Then we define (p_1, R_1) and (p_2, R_2)* clones in *P_1 and P_2 if and only if there exists a program Q, predicate q and set of arguments R such that $P_1 \rightsquigarrow^*_{\mathcal{R}} Q$ correctly transforms (p_1, R_1) into (q, R) and $P_2 \rightsquigarrow^*_{\mathcal{R}} Q$ correctly transforms (p_2, R_2) into (q, R).*

Example 3. Reconsider the definitions from Example 1. If we permute, in the definition of p, the first and second arguments we obtain a predicate, say p', defined as follows:

$$p'(B, A, P) \leftarrow B = 1, P = A$$
$$p'(B, A, P) \leftarrow B' = B - 1, p'(B', A, P'), P = P' + A$$

which is a variant of the predicate in which *sp* was transformed using the transformation sequence from Example 2. Hence $(sp,\langle 1, 2, 4\rangle)$ and $(p,\langle 2, 1, 3\rangle)$ can be considered a clone pair since each can be correctly transformed into $(p',\langle 1, 2, 3\rangle)$.

Our approach towards defining semantic clones is somewhat different from other transformation-based approaches in the sense that we consider (parts of) programs to be semantic clones if each of them can be transformed into a third, common, program while preserving the semantics (with respect to a subset of argument positions). As such, the third program captures the essence of the computations performed by the two given programs. Note that if all transformations from $\mathcal{R}$ are reversible, then this is equivalent to transforming P_1 into P_2 or vice versa, as is the more common approach towards defining algorithmic equivalence by transformation [9]. Also note that our definition of semantic clones is parametrized with respect to the set of allowable transformations $\mathcal{R}$. Essentially this corresponds to defining a *family* of semantic clones.

3 Towards an algorithm

Searching whether two predicates $p \in P_1$ and $q \in P_2$ are considered cloned necessitates thus to construct two transformation sequences, one for each program in the hope to arrive at a common program Q. Two problems present themselves: (1) even when the set of admissible transformations $\mathcal{R}$ is limited (containing, for example only unfolding and slicing), there might be a considerable number of ways in which a partial transformation sequence $\langle P_0, \ldots, P_{k-1}\rangle$ can be extended into $\langle P_0, \ldots, P_k\rangle$. And (2), since we don't know the target program Q in advance, it is hard to steer the search process. To tackle these problems, we first organize

the constructed transformation sequences into a tree structure where the nodes are the transformed programs and each node is labeled by the argument positions that are preserved by the sequence of transformations from the root to the node:

Definition 5. *Given a program P_0 and a predicate $p/n \in P_0$, a transformation tree for p in P_0 is a tree in which each node has the form (P, R, R') where P is a program and R and R' are sequences over $\{1, \ldots, n\}$. The root of the tree is $(P_0, \langle\rangle, \langle\rangle)$ and for each node (P, R, R') it holds that $P_0 \rightsquigarrow^*_{\mathcal{R}} P$ correctly transforms (p, R) into (p, R'). For a tree τ we use leafs(τ) to represent the leaves of the tree.*

In other words, a transformation tree can be constructed by repeatedly extending one of its leafs by transforming the program contained in the leaf using one of the program transformations from $\mathcal{R}$. Next, we introduce the concept of abstraction that allows both to keep the tree finite and to limit its branches to the most promising ones. We assume given a quasi-order $\leq$ defined on goals such that for goals G and G', $G \leq G'$ denotes that G is more general than G'. We furthermore assume an abstraction operator based on $\leq$.

Definition 6. *Given a quasi-order $\leq$ on goals, an* abstraction operator $\mathcal{A}$ *allows to compute a generalization of two goals. Given goals G_1, G_2 then $\mathcal{A}(G_1, G_2)$ represents a goal G such that $G \leq G_1$ and $G \leq G_2$.*

While different incarnations of such a quasi-order can be defined, one typical definition could be the following: $G \leq G'$ if and only if there exists a substitution θ such that $G\theta \subseteq G'$. This is a straightforward adaption of the well-known "more general than" relation defined on atoms and (ordered) conjunctions (e.g. ([22]) and the one we use in this work. Given an abstraction operator on goals, it is possible to define the generalization of clauses and predicates as illustrated by the following example.

Example 4. Consider the predicate $s/3$ computing in its third argument the sum of its first two arguments.

$$s(A, B, S) \leftarrow B = 0, S = A$$
$$s(A, B, S) \leftarrow B' = B - 1, s(A, B', P'), S = S' + 1$$

Then it is not hard to see that

$$s'(A, B, S, N, I) \leftarrow B = N, S = A$$
$$s'(A, B, S, N, I) \leftarrow B' = B - 1, s'(A, B', S', N, I), S = S' + I$$

can be considered a generalization of the $s/3$ predicate defined in the present example and the $p/3$ predicate defined in Example 1. Indeed, it can be obtained by pairwise considering the predicates' clauses, constructing a new (generalized) clause by generalizing the respective body goals using the abstraction operator, introducing (a subset of) the new variables as arguments and careful renaming so that all clauses share the same head.

In previous work, we have showed that computing these generalizations – in particular the most specific, or most precise, generalization – is a non-straightforward problem, and have proposed an algorithm for computing a generalization that approximates the most specific generalization of two sets of atoms in polynomially bounded time [23]. In this work we take such an abstraction algorithm for granted (formalized by our abstraction operator $\mathcal{A}$) and we study how such an abstraction operator can be used for steering the search for semantic clone pairs. First we introduce the notion of a size measure, represented by $|.|$, being a function that defines the size of a syntactic construction (be it a goal, clause, or predicate definition). The size measure is such that

- For any syntactical constructs a and b that are variants of each other, then $|a| = |b|$
- For any syntactical constructs a and b, if a is more general than b ($a \leq b$), then $|a| \leq |b|$.

Such a size measure can be used to define a distance between two predicate definitions as in the following definition.

Definition 7. *Given an abstraction operator $\mathcal{A}$ and a size measure $|.|$ measuring the size of a predicate definition, then we define the* distance *between predicates p and q as follows:*

$$\delta(p, q) = 1 - \frac{2 \times |\mathcal{A}(p, q)|}{|p| + |q|}$$

Since, by definition, $|\mathcal{A}(p, q)| \leq |p|$ and $|\mathcal{A}(p, q)| \leq |q|$, we have that $\delta(p, q)$ is a value between 1 and 0. If the generalization $\mathcal{A}(p, q)$ is empty, the distance will be 1. On the other hand, the distance will be zero if the predicates are variants of each other. Now, given programs P_1 and P_2 and predicates $p_1 \in P_1$ and $p_2 \in P_2$, we can determine whether p_1 and p_2 are clones by constructing transformation trees τ_1 for p_1 in P_1 and τ_2 for p_2 in P_2. The construction proceeds by repeatedly applying the following steps:

- Select from $\mathit{leafs}(\tau_1) \times \mathit{leafs}(\tau_2)$ the N most promising pairs. These are the pairs $((P_1^i, R_1, R_1'), (P_2^j, R_2, R_2'))$ for which the definitions of $p_1 \in P_1^i$ and $p_2 \in P_2^j$ are closest in distance. We call this set of pairs S.
- For each pair $((P_1^i, R_1, R_1'), (P_2^j, R_2, R_2')) \in S$, create K children for each node in the pair by applying an admissible transformations from $\mathcal{R}$.

The process is repeated as long as the successive incarnations of the set S differ in at least one element of better quality (i.e. a pair of nodes having a strictly smaller distance than the element it replaces). Since the distances are bounded by zero, the process is necessarily terminating. Note that the process is parametrized by N and K. If $N = 1$ the process continues by transforming in each step *the* most promising couple. While this might be efficient, it is in no way guaranteed that the search finds the "right" transformation sequences as the process can be stuck in a local optimum. Using a larger value for N is a rudimentary way of eliminating this problem. The parameter K on the other hand allows to explore

different transformations (at least when $K > 1$) in order to extend a single node from S.

While the skeleton algorithm details how the search is controlled (it specifies how termination is guaranteed and how the N *most promising* pairs of leafs can be chosen in each round), it does not specify how to search for the K most interesting program transformations to apply to a single selected pair of leafs $((P_1^i, R_1, R_1'), (P_2^j, R_2, R_2')) \in S$. Although the details of such a search procedure need to be further investigated, the idea is to select those program transformations that, again, lower the distance between the current definitions of predicates p_1 and p_2 as they are defined in P_1^i and P_2^j respectively. For this, again information from the generalization process can be used to guide the selection. Indeed, for a pair of corresponding clauses $H_1 \leftarrow G_1$ and $H_2 \leftarrow G_2$ in the definition of p_1, respectively p_2, we can compute $G = \mathcal{A}(G_1, G_2)$ and, in general, substitutions θ_1 and θ_2 and goals Δ_1 and Δ_2 such that $G\theta_1 \cup \Delta_1 = G_1$ and $G\theta_2 \cup \Delta_2 = G_2$. In other words, the generalization G represents the part that is common to p_1 and p_2 while Δ_1 and Δ_2 represents the parts specific to the current definition of p_1, respectively p_2. Information from these structures can be exploited in order to select promising transformations:

- if $\Delta_1 = \emptyset$, it means that the generalization $\mathcal{A}(p_1, p_2)$ is of maximal size. Therefore, the only meaningful way in which the search can continue is by transforming p_2 in such a way that literals from Δ_2 are eliminated (e.g. by applying slicing transformations). The case where $\Delta_2 = \emptyset$ is analogous.
- If neither Δ_1 nor Δ_2 are empty, the search should focus on making Δ_1 and Δ_2 more similar, in order to enlarge the common part G shared by both clauses or, less preferably, render both Δ_1 and Δ_2 smaller. This can potentially be achieved by:
  (a) Unfolding atoms in Δ_1 (respectively Δ_2), in particular if unfolding gives rise to (variants of) constraints present in Δ_2 (respectively Δ_1).
  (b) Other transformations permitted by $\mathcal{R}$ might be applied in order to enlarge the similarity between Δ_1 and Δ_2, including for example specialization and folding.
  (c) Constraints and/or atoms could be sliced from Δ_1 and/or Δ_2.

A preliminary study on how to define a strategy for this local search has been undertaken [24] but needs to be developed further.

The following simplistic example is an illustration for some of the ideas driving the transformation process described above.

Example 5. Let us once more consider the predicates $p/3$ and $sp/4$ from Example 1. The following predicate is a typical most specific generalization of $p/3$ and $sp/4$:

$$g(G_1, G_2, G_3) \leftarrow G_2 = 1, G_3 = G_1$$
$$g(G_1, G_2, G_3) \leftarrow G_4 = G_2 - 1, g(G_1, G_4, G_5), G_3 = G_5 + G_1$$

It is easy to see that $g/3$ is in fact a variant of $p/3$. This isn't surprising given that a most specific common generalization by definition abstracts away the parts

that are not common to the two predicates under consideration. Now using Δ_1, respectively Δ_2, to represent the sets of atoms separating the body of (a variant of) $p/3$, respectively $sp/4$, from that of $g/3$, we have $\Delta_1 = \emptyset$ and $\Delta_2 \neq \emptyset$. This is an indication towards applying some judicious slicing transformation on $sp/4$, yielding:

$$sp(A, B, P) \leftarrow A = 1, P = B$$
$$sp(A, B, P) \leftarrow A' = A - 1, sp(A', B, P'), P = P' + B$$

which is a variant of $p/3$ (and $g/3$), leading to the conclusion that $(p, \langle 1, 2, 3\rangle)$ and $(sp, \langle 2, 1, 4\rangle)$ are clones.

4 Discussion

While the theoretical framework of semantic clones in logic programming has been established before, this work is – to the best of our knowledge – a first attempt in devising a practical algorithm capable of *searching* for a series of program transformations that reduce two given CLP fragments to a single code fragment that represents the functionality that is common to the two fragments; as such proving that the fragments are (at least to some extent) semantic clones.

The search algorithm that we propose is essentially comprised of two control levels: one level that controls the termination of the process and a second one that considers what candidate transformations to apply next. In that respect, it is not unlike control techniques used in partial deduction [25] where a *global* control level is used to ensure termination of the process and a *local* control is concerned by constructing a suitable SLD tree for an atom or a conjunction of atoms.

Working out the details of our search procedure is the topic of ongoing and future research. However, we have seen that a key ingredient is a generalization operator that allows to generalize two goals and that can, additionally, be used to compute a distance between these goals. Generalizing is a simple and well-known syntactical process, at least as far as single atoms or (ordered) conjunctions are concerned. It becomes more complicated when, as is the case in our setting, sets of atoms (or constraints) need to be considered, especially when the aim is to compute a most specific generalization. We have recently devised an approximation algorithm for computing most specific generalizations of sets of literals [23], and aim to incorporate this into the algorithm under development. Another topic of further work is to incorporate higher-order generalization capabilities in the algorithm, which is currently restricted to first-order generalizations only.

Acknowledgments

We thank anonymous reviewers for their constructive input and remarks.

References

1. Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.
2. Charles Rich, Howard E. Shrobe, and Richard C. Waters. Overview of the programmer's apprentice. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 827–828, 1979.
3. Cordell Green, David Luckham, Robert Balzer, Thomas Cheatham, and Charles Rich. Report on a knowledge-based software assistant. Technical report, Kestrel Institute, 1983.
4. Margaret-Anne D. Storey. Theories, methods and tools in program comprehension: Past, present and future. In *13th International Workshop on Program Comprehension (IWPC)*, pages 181–191, 2005.
5. Fangfang Zhang, Yoon-Chan Jhi, Dinghao Wu, Peng Liu, and Sencun Zhu. A first step towards algorithm plagiarism detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 111–121. ACM, 2012.
6. Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '14, pages 25–36. ACM, 2014.
7. Dhavleesh Rattan, Rajesh Kumar Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information & Software Technology*, 55(7):1165–1199, 2013.
8. Beniamino Di Martino and Giulio Iannello. PAP recognizer: A tool for automatic recognition of parallelizable patterns. In *4th International Workshop on Program Comprehension (WPC)*, page 164, 1996.
9. R. Metzger and Z. Wen. *Automatic Algorithm Recognition and Replacement*. The MIT Press, 2000.
10. Huiqing Li and Simon Thompson. Clone detection and removal for Erlang/OTP within a refactoring environment. In *Proceedings of the 2009 SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'09)*, pages 169–178. ACM, 2009.
11. Christopher Brown and Simon Thompson. Clone detection and elimination for Haskell. In *Proceedings of the 2010 SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'10)*, pages 111–120. ACM, 2010.
12. Céline Dandois and Wim Vanhoof. Clones in logic programs and how to detect them. In *Proceedings of the 21st International Conference on Logic-Based Program Synthesis and Transformation*, LOPSTR'11, pages 90–105, Berlin, Heidelberg, 2012. Springer-Verlag.
13. C. Alias and D. Barthou. Algorithm recognition based on demand-driven data-flow analysis. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*, pages 296–305, 2003.
14. Ahmad Taherkhani. Using decision tree classifiers in source code analysis to recognize algorithms: An experiment with sorting algorithms. *Comput. J.*, 54(11):1845–1860, 2011.
15. Ahmad Taherkhani and Lauri Malmi. Beacon- and schema-based method for recognizing algorithms from students' source code. *Journal of Educational Data Mining*, 5(2):69–101, 2013.

16. C. Dandois and W. Vanhoof. Semantic code clones in logic programs. In E. Albert, editor, *Proc. of the 22nd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'12)*, volume 7844 of *LNCS*, pages 35–50. Springer, 2012.
17. Frédéric Mesnard, Étienne Payet, and Wim Vanhoof. Towards a framework for algorithm recognition in binary code. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, PPDP '16, pages 202–213, New York, NY, USA, 2016. ACM.
18. l. Horn clauses as an intermediate representation for program analysis and transformation. *TPLP*, 15(4-5):526–542, 2015.
19. J. Jaffar, M. J. Maher, K. Marriott, and P. J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.
20. Alberto Pettorossi and Maurizio Proietti. Transformation of logic programs. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 697–787. Oxford University Press, 1998.
21. Gyöngyi Szilágyi, Tibor Gyimóthy, and Jan Małuszyński. Static and dynamic slicing of constraint logic programs. *Automated Software Engineering*, 9(1):41–65, 2002.
22. Morten Heine Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In John W. Lloyd, editor, *Logic Programming, Proceedings of the 1995 International Symposium, Portland, Oregon, USA, December 4-7, 1995*, pages 465–479. MIT Press, 1995.
23. Gonzague Yernaux and Wim Vanhoof. Anti-unification in constraint logic programming. *Theory Pract. Log. Program.*, 2019. Accepted for publication.
24. Gonzague Yernaux. Équivalence algorithmique par transformations de programmes logiques avec contraintes. Master's thesis, University of Namur, 2017.
25. Michael Leuschel and Maurice Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory Pract. Log. Program.*, 2(4-5):461–515, July 2002.

Semi-Inversion of Conditional Constructor Term Rewriting Systems

Maja Hanne Kirkeby[1] and Robert Glück[2]

[1] Roskilde University, Denmark kirkebym@acm.org
[2] DIKU, University of Copenhagen, Denmark glueck@acm.org

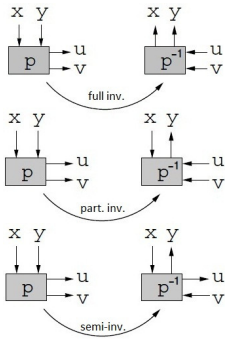
Abstract. Inversion is an important and useful program transformation and has been studied in various programming language paradigms. Semi-inversion is more general than just swapping the input and output of a program; instead, parts of the input and output can be freely swapped. In this paper, we present a polyvariant semi-inversion algorithm for conditional constructor term rewriting systems. These systems can model logic and functional languages, which have the advantage that semi-inversion, as well as partial and full inversion, can be studied across different programming paradigms. The semi-inverter makes use of local inversion and a simple but effective heuristic and is proven to be correct. A Prolog implementation is applied to several problems, including inversion of a simple encrypter and of a program inverter for a reversible language.

Keywords: program transformation, program inversion, conditional term rewriting systems, logic and functional programs

1 Introduction

Programs that are inverse to each other are widely used, such as encoding and decoding of data. The transformation of an encoder into a decoder, or vice versa, is called full inversion. *Semi-inversion*, the most general type of program inversion, transforms one relation into a new relation that takes a subset of the original input and output as the new input. For example, the transformation of a symmetric encrypter into a decrypter cannot be achieved by conventional full inversion because both programs take the same key as input.

In this paper, we present a polyvariant semi-inversion algorithm for an oriented *conditional constructor term rewriting system* (CCS) [22]. The algorithm makes use of local inversion and a simple but effective heuristic and is proven to be correct. A Prolog implementation is applied to several transformation problems, including the inversion of a simple symmetric encrypter. As a special transformation challenge, a program inverter for a reversible imperative language was inverted into a copy of itself modulo variable renaming.



We distinguish between three forms of program inversion: *Full inversion* turns a program `p` into a new program $\mathtt{p}^{-1}$, where the original inputs and outputs are exchanged. If `p` is injective, then $\mathtt{p}^{-1}$ implements a function. *Partial inversion* yields a program $\mathtt{p}^{-1}$ that inputs the original output (`u`, `v`) and some of the original input (`x`)

and then returns the remaining input (y). *Semi-inversion* yields a program $\mathtt{p}^{-1}$ that, given some of the original input (x) and some of the original output (v), returns the remaining input (y) and output (u). The programs p and $\mathtt{p}^{-1}$ may implement functions or more general relations. Full inversion is a subproblem of partial inversion, which is a subproblem of semi-inversion:

$$\text{full inversion} \subseteq \text{partial inversion} \subseteq \text{semi-inversion}.$$

Dijkstra was the first to study the full inversion of programs in a guarded command language [6]. Subsequently, some program inversion algorithms were developed for different forms of inversion and for different programming languages. Of those, only Nishida et al. [18] and Almendros-Jiménez et al. [2] have considered term rewriting systems, where the latter constrained the systems such that terms have a unique normal form, i.e., the systems express functional input-output relations. Mogensen [14,15], who developed the first semi-inversion algorithm, did so for a deterministic guarded equational language, i.e., with functional input-output relations. Methods for inversion have been studied in the context of functional languages [7–11]. The motivation for using *program inversion* instead of *inverse interpretation*, such as [1, 13], is similar to the motivation for using translation instead of interpretation.

The main advantage of oriented conditional constructor term rewriting systems is that they can model both *logic* and *functional* languages [5], and, hence, *functional logical* languages [12]. When modeling logic languages, an efficient evaluation requires narrowing (unification) [5, 12], which is typically slower than standard rewriting (matching). By requiring that the rewrite rules be not only left-orthogonal but also right-orthogonal and non-deleting, we can also model *reversible* languages [23]. This enables us to focus on the essence of semi-inversion without considering language-specific details. The semantics of *functions* and *relations* can be expressed and efficiently calculated in the same formalism (*cf.*, Ex. 2). The idea to use CCSs to investigate semi-inversion for different language paradigms was inspired by the partial inverter developed by Nishida et al. [18].

The new semi-inverter relates to some of the mentioned inversion algorithms:

	functions	relations
full inversion	Glück and Kawabe [8]	Nishida et al. [17, 19]
partial inversion	Almendros-Jiménez et al. [2]	Nishida et al. [18]
semi-inversion	Mogensen [14]	*This algorithm*

This paper provides (1) a polyvariant semi-inversion algorithm for CCSs that uses local inversion and is proven correct; (2) a simple but effective heuristic to avoid narrowing and to minimize the search space; and (3) an experimental evaluation by applying the Prolog implementation to a simple encryption algorithm, a physical discrete-event simulation, and a program inverter for a reversible language.

Overview: After an brief overview of the semi-inverter (Sect. 2), we formally define conditional term rewriting systems and semi-inversion (Sect. 3). Then, we present our algorithm (Sect. 4) and report on the experimental results (Sect. 5).[3]

[3] The extended abstract of a talk, Nordic Workshop on Programming Theory, Univ. of Bergen, Dept. of Informatics, Report 403, 2012, is partially used in Sections 1 and 5.

2 The Semi-Inversion Algorithm—An Overview

This section gives an informal overview of semi-inversion and illustrates the semi-inversion algorithm with a short, familiar example. Both semi-inversion and the algorithm will be formalized and defined in the following sections.

We let semi-inversion cover *rewritings* of the form

$$f(s_1, \ldots, s_n) \;\to^*_{\mathcal{R}}\; \langle t_1, \ldots, t_m \rangle,$$

where f is an n-ary *function symbol* with co-arity m defined in the conditional term rewriting system $\mathcal{R}$; *input and output terms* $s_1, \ldots, s_n$ and $t_1, \ldots, t_m$ are ground constructor terms; and $\langle ... \rangle$ is a special m-ary constructor containing the m output terms. The transformation of f into a semi-inverse $\underline{f}$ is w.r.t. indices of known input and output terms. If we assume that the first a input and b output terms are the known arguments, then the semi-inverse $\underline{f}_{\{1,\ldots,a\}\{1,\ldots,b\}}$ takes the form

$$\underline{f}_{\{1,\ldots,a\}\{1,\ldots,b\}}(s_1, \ldots, s_a, t_1, \ldots, t_b) \;\to^*_{\underline{\mathcal{R}}}\; \langle s_{a+1}, \ldots, s_n, t_{b+1}, \ldots, t_m \rangle.$$

The *input-output index sets* $\{1, \ldots, a\}$ and $\{1, \ldots, b\}$ label the new function symbol $\underline{f}$ and serve to distinguish different semi-inverses of the same f. The semi-inversion algorithm locally inverts each rule needed for the rewriting sequence in the *semi-inverted rewriting system* $\underline{\mathcal{R}}$ such that the known parameters specified by the two index sets occur on the left-hand side and all others occur on the right-hand side of the semi-inverted rules of f. Semi-inversion is *polyvariant* because $\underline{\mathcal{R}}$ may include several different semi-inversions of the rules defining f in $\mathcal{R}$, while, in contrast, full inversion is *monovariant*, as it requires only one variant per f.

Example 1. Take as an example the multiplication $x \cdot y = z$ of two unary numbers x and y defined by adding y x times (`s1`–`s4`, Fig. 1), which is similar to the unconditional system (`r1`–`r4`, Fig. 1) suggested by [18]. Inversion of multiplication `mul` is w.r.t. the second input y and the first (and only) output z. That is, input-output index sets $I = \{2\}$ and $O = \{1\}$ yield the rewrite rules for division $z/y = x$ and, as a subtask, partially inverts addition $x + y = z$ into subtraction $z - x = y$ (`t1`–`t4`, as shown in Fig. 1). Multiplication `mul(x,y)` and addition `add(x,y)` are inverted into division $\underline{\texttt{mul}}_{\{2\}\{1\}}$`(y,z)` and subtraction $\underline{\texttt{add}}_{\{1\}\{1\}}$`(x,z)` and are replaced by the forms `div(z,y)` and `sub(z,x)` for readability. Some of the inverted rules have a conditional part (the conjunction to the right of $\Leftarrow$), which must be satisfied to apply a rule and may bind variables, *e.g.*, `y`, in rule `t4`.

The algorithm is illustrated in Fig. 2 by the stepwise inversion of rule `s2`. First, all function symbols are labeled with index sets, starting with the given index sets on the left-hand side and then repeatedly (from left to right) labeling the function symbols in the conditions with indices of the known arguments (Step 1). Variable `w` is known after `add` is rewritten, so the rightmost `mul` is labeled $\texttt{mul}_{\{2\}\{1\}}$. Finally, *local inversion* brings all known parts to the left-hand side according to the index sets (Step 2), e.g., $\texttt{mul}_{\{2\}\{1\}}$`(s(x),y)`$\to\langle$`z`$\rangle$ into $\underline{\texttt{mul}}_{\{2\}\{1\}}$`(y,z)`$\to\langle$`s(x)`$\rangle$.

We note that in Fig. 1, division by zero, `div(z,0)`, is undefined (due to infinite rewriting by repeatedly subtracting `0` from `z`). Division of zero, `div(0,y)`,

Unconditional rules:

$\texttt{r1: mul(0,y)} \to \texttt{0}$ $\quad$ $\texttt{r2: mul(s(x),y)} \to \texttt{add(y,mul(x,y))}$

$\texttt{r3: add(0,y)} \to \texttt{y}$ $\quad$ $\texttt{r4: add(s(x),y)} \to \texttt{s(add(x,y))}$

Flat rules:

$\texttt{s1: mul(0,y)} \to \langle \texttt{0} \rangle$ $\quad$ $\texttt{s2: mul(s(x),y)} \to \langle \texttt{z} \rangle \Leftarrow \texttt{add(y,w)} \to \langle \texttt{z} \rangle \wedge \texttt{mul(x,y)} \to \langle \texttt{w} \rangle$

$\texttt{s3: add(0,y)} \to \langle \texttt{y} \rangle$ $\quad$ $\texttt{s4: add(s(x),y)} \to \langle \texttt{s(z)} \rangle \Leftarrow \texttt{add(x,y)} \to \langle \texttt{z} \rangle$

Inverted rules (after renaming):

$\texttt{t1: div(0,y)} \to \langle \texttt{0} \rangle$ $\quad$ $\texttt{t2: div(z,y)} \to \langle \texttt{s(x)} \rangle \Leftarrow \texttt{sub(z,y)} \to \langle \texttt{w} \rangle \wedge \texttt{div(w,y)} \to \langle \texttt{x} \rangle$

$\texttt{t3: sub(y,0)} \to \langle \texttt{y} \rangle$ $\quad$ $\texttt{t4: sub(s(z),s(x))} \to \langle \texttt{y} \rangle \Leftarrow \texttt{sub(z,x} \to \langle \texttt{y} \rangle$

Fig. 1. Partial inversion of multiplication into division.

Label all function symbols (index set propagation):

Step 1
$$\begin{cases} \texttt{mul(s(x),y)} \to \langle \texttt{z} \rangle \Leftarrow \texttt{add(y,w)} \to \langle \texttt{z} \rangle \wedge \texttt{mul(x,y)} \to \langle \texttt{w} \rangle \\ \texttt{mul}_{\{2\}\{1\}}\texttt{(s(x),y)} \to \langle \texttt{z} \rangle \Leftarrow \texttt{add}_{\{1\}\{1\}}\texttt{(y,w)} \to \langle \texttt{z} \rangle \wedge \texttt{mul}_{\{2\}\{1\}}\texttt{(x,y)} \to \langle \texttt{w} \rangle \end{cases}$$

Local inversion:

Step 2
$$\begin{cases} \texttt{mul}_{\{2\}\{1\}}\texttt{(s(x),y)} \to \langle \texttt{z} \rangle \Leftarrow \texttt{add}_{\{1\}\{1\}}\texttt{(y,w)} \to \langle \texttt{z} \rangle \wedge \texttt{mul}_{\{2\}\{1\}}\texttt{(x,y)} \to \langle \texttt{w} \rangle \\ \underline{\texttt{mul}}_{\{2\}\{1\}}\texttt{(y,z)} \to \langle \texttt{s(x)} \rangle \Leftarrow \underline{\texttt{add}}_{\{1\}\{1\}}\texttt{(y,z)} \to \langle \texttt{w} \rangle \wedge \underline{\texttt{mul}}_{\{2\}\{1\}}\texttt{(y,w)} \to \langle \texttt{x} \rangle \end{cases}$$

Fig. 2. Stepwise inversion of rule `s2` in Fig. 1 w.r.t. the input-output index sets {2} {1}.

where $\texttt{y} > 0$, returns zero. Both rules `t1` and `t2` match, but only rule `t1` can be applied (`sub(0,y)` in the condition of `t2` cannot be rewritten with $\texttt{y} > 0$). The pre-processing and post-processing that transform between unconditional and flat conditional constructor systems are standard techniques and not discussed here; see, *e.g.*, [18, 21, 22].

3 Conditional Constructor Systems and Semi-Inversion

First, we recall the basic concepts of conditional term rewriting systems following the terminology of Ohlebusch [22] and their ground constructor-based relation [19]. Then, we define what we call conditional constructor term rewriting systems (CCSs) and describe how they model a series of language paradigms. We also describe the properties for when they can be evaluated efficiently, which relates to the design goals for our semi-inverter. Finally, we define semi-inversion of such systems and give the first insights into the nature of semi-inversion.

3.1 Preliminaries for Conditional Term Rewriting

We assume a countable set of variables $\mathcal{V}$. A finite signature $\mathcal{F}$ is assumed to be partitioned into two disjoint sets: a set of *defined function* symbols $\mathcal{D}$, each $f \in \mathcal{D}$ with an arity n and a co-arity m, written $f/n/m$, and a set of constructor symbols $\mathcal{C}$, each $a \in \mathcal{C}$ with an arity n. We denote the set of all terms over $\mathcal{F}$ and $\mathcal{V}$ by $T(\mathcal{F}, \mathcal{V})$. A term s is a *ground* term if it has no variables, a *constructor* term if it contains no function symbols, and a *ground constructor* term if it is both a ground term and a constructor term. Every subterm s of a term t has at least a position p, and we denote this subterm by $t|_p = s$, with the root symbol denoted $root(t)$. Furthermore, we let $t[s']_p$ denote a new term where the subterm at position p in t is replaced by a new (sub)term s'. A *substitution* σ

is a mapping from variables to terms, a *ground substitution* is a mapping from variables to ground terms, and a *constructor substitution* is a mapping from variables to constructor terms.

A *conditional rewrite rule* is of the form $l \to r \Leftarrow c$, where the left-hand side l is a non-variable and $root(l) \in \mathcal{D}$, the right-hand side r is a term, and the *conditions* c are a (perhaps empty) conjunction of conditions $l_1 \to r_1 \wedge \ldots \wedge l_k \to r_k$.

A *conditional term rewriting system* $\mathcal{R}$ *over a signature* $\mathcal{F}$, abbreviated *CTRS*, is a finite set of conditional rewrite rules $l_0 \to r_0 \Leftarrow l_1 \to r_1 \wedge \ldots \wedge l_k \to r_k$ over all terms in $T(\mathcal{F}, \mathcal{V})$ such that the defined functions $\mathcal{D} = \{root(l) \mid l \to r \Leftarrow c \in \mathcal{R}\}$ and constructors $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$. The conditions are interpreted as reachability, defining a so-called oriented CTRS, e.g., [22].

A *ground constructor-based rewrite relation* $\to_{\mathcal{R}}$ associated with a CTRS over $\mathcal{F}$ $\to_{\mathcal{R}}$ is the smallest binary relation for a pair of ground terms $s, t \in T(\mathcal{F}, \emptyset)$, where there is a position p, a ground constructor substitution σ and a rewrite rule $l \to r \Leftarrow c$ such that $s|_p = l\sigma$, $s[r\sigma]_p = t$ and, for each condition $(l_i \to r_i) \in c$, $l_i\sigma \to^*_{\mathcal{R}} r_i\sigma$.

3.2 Conditional Constructor Systems

In this study, we focus on a subclass of CTRSs we call *conditional constructor term rewriting systems*. These systems are both input to and output from the semi-inversion algorithm. They are also referred to as pure constructor CTRSs in the literature [16] and are a subset of 4-CTRSs [22]. They can model first-order functional programs, logic programs, and functional logic programs [5] and are suitable for observing and discussing common problems arising from inversion without considering different language specifications.

The purpose of these systems is to describe relations f from n ground constructor terms to m ground constructor terms, that is,

$$f(s_1, \ldots, s_n) \ \to^*_{\mathcal{R}} \ \langle t_1, \ldots, t_m \rangle.$$

We assume that the signature includes special constructors $\langle\rangle/m$ intended to contain the m output and function symbols of the form $\texttt{mul}/2/1$ and $\texttt{mul}_{\{2\}\{1\}}/2/1$.

Definition 1 (CCS). *A* conditional constructor term rewriting system $\mathcal{R}$*, abbreviated* CCS*, is a CTRS over* $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ *if each rule in* $\mathcal{R}$ *is of the form*

$$l_0 \to r_0 \Leftarrow l_1 \to r_1 \wedge \ldots \wedge l_k \to r_k,$$

where each $l_i \to r_i$ $(0 \leq i \leq k)$ *is of the form* $f^i(p^i_1, \ldots, p^i_{n_i}) \to \langle q^i_1, \ldots, q^i_{m_i} \rangle$ *such that* $f^i/n_i/m_i \in \mathcal{D}$*,* $\langle\rangle/m_i \in \mathcal{C}$*, and* p^i_j *and* $q^i_{j'}$ *are constructor terms.*

We shall only consider the associated ground constructor-based rewrite relation [19] described in Sect. 3.1. The reductions $f(s_1, \ldots, s_n) \to^*_{\mathcal{R}} \langle t_1, \ldots, t_m \rangle$, where all s_i and t_j are ground constructor terms, specified by a CCS, can only be 1-step reductions, that is, $f(s_1, \ldots, s_n) \to_{\mathcal{R}} \langle t_1, \ldots, t_m \rangle$. The left- and right-hand side of the rules are not unifiable, prohibiting 0-step reductions; there is one function symbol in the initial term, and each rule-application removes exactly one function symbol, prohibiting reductions with more than one step. This

also simplifies the correctness proof of the semi-inversion algorithm, which can be proven by induction over the depth of the rewrite steps [22, Def. 7.1.4].

The next example defines a rewrite relation by overlapping rules.

Example 2. This CCS defines a one-to-many rewrite relation `perm`/1/1 between a list and all its permutations, *e.g.*, `perm([1|[2|[]]])` $\to \langle$`[1|[2|[]]]`$\rangle$ and `perm([1|[2|[]]])` $\to \langle$`[2|[1|[]]]`$\rangle$. It has two defined function symbols `perm`/1/1 and `del`/1/2 and a set of constructors, including two list constructors, `[]`/0 and `[·|·]`/2, and two special output constructors, $\langle\cdot\rangle$/1 and $\langle\cdot,\cdot\rangle$/2. The defined function symbol `perm` depends on `del`, which removes an arbitrary element from a list and returns the removed element and the remaining list. The nondeterministic relation is caused by the overlapping rules `r3` and `r4`.

`r1:` `perm([])` $\to \langle$`[]`$\rangle$
`r2:` `perm(x)` $\to \langle$`[y|z]`$\rangle \Leftarrow$ `del(x)` $\to \langle$`y,u`$\rangle \wedge$ `perm(u)` $\to \langle$`z`$\rangle$
`r3:` `del([x|y])` $\to \langle$`x,y`$\rangle$
`r4:` `del([x|y])` $\to \langle$`z, [x|u]`$\rangle \Leftarrow$ `del(y)` $\to \langle$`z,u`$\rangle$

A CCS can be nondeterministic by overlapping rules, as in Example 2, and by what we call *extra variables*[4], i.e., variables occurring on the right-hand side r of a rule but neither in its left-hand l side nor in its conditions c, i.e., $\mathcal{V}ar(r) \setminus (\mathcal{V}ar(l) \cup \mathcal{V}ar(c))$. In case a system has no extra variables, we call it *extra-variable free*, abbreviated *EV-free*. EV-free CCSs are a subset of 3-CTRSs [22], and pcDCTRSs [20] are a subset of EV-free CCSs.

The extra variables cause infinite branching in the ground constructor-based rewrite relation; for example, a rule $f() \to \langle x\rangle$ represents an infinite ground constructor-based rewrite relation $\{f() \to_{\mathcal{R}} \langle a\rangle, f() \to_{\mathcal{R}} \langle b\rangle, \ldots\}$. Intuitively, these variables can be interpreted as *logic variables* subsuming all possible ground constructor terms. Extra variables require efficient implementations that do not naively produce the entire ground constructor-based rewrite relation. *Narrowing* is a well-established rewriting method, where matching is replaced by unification; see, e.g., [5] for further details, [3, 12] for a survey, and [18] for the use in partial inversion.

3.3 Semi-Inverse

The reader has already seen an example of semi-inversion in Sect. 2. Next, we define semi-inversion formally and illustrate it with examples of the algorithm.

Definition 2 (semi-inverse). *Let $\mathcal{R}$ and $\underline{\mathcal{R}}$ be CCSs over $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ and $\underline{\mathcal{F}} = \underline{\mathcal{C}} \uplus \underline{\mathcal{D}}$, respectively, with $f/n/m \in \mathcal{D}$ and $\underline{f}_{IO}/\underline{n}/\underline{m} \in \underline{\mathcal{D}}$, where $I = \{i_1, \ldots, i_a\}$ and $O = \{o_1, \ldots, o_b\}$ are index sets such that $\underline{n} = a + b$ and $\underline{m} = m + n - \underline{n}$. Then, $\underline{\mathcal{R}}$ is a* semi-inverse of $\mathcal{R}$ w.r.t. f, I, and O *if for all ground constructor terms $s_1, \ldots, s_n$, $t_1, \ldots, t_m \in T(\mathcal{C} \setminus \{\langle\rangle\}, \emptyset)$,*

$$f(s_1, \ldots, s_n) \to_{\mathcal{R}} \langle t_1, \ldots, t_m\rangle \quad \Leftrightarrow$$
$$\underline{f}_{IO}(s_{i_1}, \ldots, s_{i_a}, t_{o_1}, \ldots, t_{o_b}) \to_{\underline{\mathcal{R}}} \langle s_{i_{a+1}}, \ldots, s_{i_n}, t_{o_{b+1}}, \ldots, t_{o_m}\rangle$$

[4] In this case, we follow the terminology of [18] —these are not to be confused with "extra variables" as defined by Ohlebusch [22], i.e., $(\mathcal{V}ar(r) \cup \mathcal{V}ar(c)) \setminus \mathcal{V}ar(l)$.

where divisions $\{i_1,\ldots,i_a\} \uplus \{i_{a+1},\ldots,i_n\} = \{1,\ldots,n\}$ *and* $\{o_1,\ldots,o_b\} \uplus \{o_{b+1},\ldots,o_m\} = \{1,\ldots,m\}$. *We assume that the name and the parameters of* $\underline{f}_{IO}$ *are ordered according to* $<$*-order on the indices.*

The reason the semi-inversion algorithm produces a CCS and not an EV-free CCS lies in the nature of full-inversion, i.e., the most specific inversion problem, as demonstrated by the next example.

Example 3. Full inversion of the EV-free CCS $\mathtt{fst(x,y)} \to \langle \mathtt{x} \rangle$ unavoidably creates a CCS with extra variables, namely, $\underline{\mathtt{fst}}_{\emptyset,\{1\}}\mathtt{(x)} \to \langle \mathtt{x,y} \rangle$.

Sometimes the semi-inverted system and its original system define the same rewrite relation but are defined differently, as in the following examples.

Example 4 (Ex. 2, continued). The semi-inverse of `perm` w.r.t. index sets $I = \emptyset$ and $O = \{1\}$, i.e., a full inversion, is a CCS that defines the same permutation relation by different rules. Here, $\underline{\mathtt{del}}_{\emptyset,\{1,2\}}$ inserts an element randomly into a list, whereas the original `del` removes an arbitrary element from the list.

$\underline{\mathtt{r1}}\colon \underline{\mathtt{perm}}_{\emptyset,\{1\}}\mathtt{([])} \to \langle \mathtt{[]} \rangle$
$\underline{\mathtt{r2}}\colon \underline{\mathtt{perm}}_{\emptyset,\{1\}}\mathtt{([y|z])} \to \langle \mathtt{x} \rangle \Leftarrow \underline{\mathtt{perm}}_{\emptyset,\{1\}}\mathtt{(z)} \to \langle \mathtt{u} \rangle \wedge \underline{\mathtt{del}}_{\emptyset,\{1,2\}}\mathtt{(y,u)} \to \langle \mathtt{x} \rangle$
$\underline{\mathtt{r3}}\colon \underline{\mathtt{del}}_{\emptyset,\{1,2\}}\mathtt{(x,y)} \to \langle \mathtt{[x|y]} \rangle$
$\underline{\mathtt{r4}}\colon \underline{\mathtt{del}}_{\emptyset,\{1,2\}}\mathtt{(z,[x|u])} \to \langle \mathtt{[x|y]} \rangle \Leftarrow \underline{\mathtt{del}}_{\emptyset,\{1,2\}}\mathtt{(z,u)} \to \langle \mathtt{y} \rangle$

3.4 Modeling Programming Languages and Evaluation Strategies

EV-free CCSs are suitable for modeling logic languages such as Prolog, as seen in the next example, where predicates are modeled by function symbols with co-arity 0. In general, logic programs require narrowing, as we shall see below.

Example 5. The classic predicate append can be modeled by the two rules $\mathtt{app([],y,y)} \to \langle\rangle$ and $\mathtt{app([h|t],y,[h|z])} \to \langle\rangle \Leftarrow \mathtt{app(t,y,z)} \to \langle\rangle$.

The evaluation order of the conditions, i.e., the *strategy*, does not affect the correctness of a rewriting, but the conditions and their order may require narrowing. Instead of describing when there exists an evaluation order, which would only require the faster term rewriting, it is standard to fix the order to be from left to right and to define for which systems there is an order that would only require term rewriting. We follow [16] and define these properties for CCSs, and not only EV-free CCSs as in [22]. For a rule $l \to r \Leftarrow l_1 \to r_1 \wedge \ldots \wedge l_k \to r_k$, a variable x in a condition l_i, i.e., $x \in \mathcal{V}ar(l_i)$, is *known* if $x \in \mathcal{V}ar(l, r_1, \ldots, r_{i-1})$, and *unknown* otherwise. The rule is *left-to-right deterministic*[5] if all variables on the left-hand sides of the conditions are known, i.e., $\mathcal{V}ar(l_i) \subseteq \mathcal{V}ar(l, r_1, \ldots, r_{i-1})$, and a CCS is *left-to-right deterministic* if all its rules are left-to-right deterministic.

A left-to-right deterministic and EV-free CCS does *not* require narrowing, and it is desirable for a semi-inverter to produce such systems [22, Sect. 7.2.5].

[5] Left-to-right determinism is referred to as "determinism" in term rewriting literature, but we make a rather clear distinction between this property, deterministic computations, and deterministic input-output relations, i.e., functions.

In addition, these systems provide a good basis for modeling functional programs [5]. However, other requirements include *orthogonal* rules, i.e., non-overlapping rules and left-linearity. These requirements will not be a part of our design focus for the semi-inversion algorithm, but we will comment on where to check for such paradigm-specific properties in the algorithm in Sect. 4.

Similarly, an EV-free CCS can model reversible languages by ensuring right-orthogonality and non-deletion. Nishida et al. [20] performed a reversibilization of a pcDCTRS[6] by labeling each right-hand side of a rule with a unique constructor, i.e., right-orthogonality, and recording all deleted values in a trace, i.e., non-deletion. Thus, their resulting systems are reversible.

4 The Semi-Inversion Algorithm

The polyvariant semi-inverter is presented in a modular way, including the local inversion and a heuristic to improve the semi-inversion by reordering the conditions. The algorithm semi-inverts a CCS w.r.t. a given function symbol and a pair of input-output index sets into a new CCS. It terminates and yields correct semi-inverse systems, as shown at the end of this section.

The semi-inverter labels all function symbols with two index sets, I and O, that contain the indices of the known terms of the left-hand and right-hand sides of a rule, respectively, and locally inverts every rule after reordering the conditions such that all known variables given by the index sets occur on the left-hand side and the rest occur on the right-hand side of the new rule.

4.1 Control of Rule Generation

The recursive semi-inversion algorithm (Fig. 3) controls the local inversion (Fig. 4) of the conditional rewrite rules. Given a rewrite system $\mathcal{R}$, an initial function symbol f and the initial input-output index sets I and O, the algorithm produces the semi-inverse rewrite system $\underline{\mathcal{R}_{f_{IO}}}$. It keeps track of the function symbols that have been semi-inverted (in set `Done`) and those that are pending semi-inversions (in set `Pend`) to address circular dependencies between rules.

A pending task $(f, I, O) \in$ `Pend` is selected, and each of the rules defining f in $\mathcal{R}$ is semi-inverted, which may lead to new semi-inversion tasks. The auxiliary procedure `getdep` collects all function symbols and their input-output index sets on which the conditions of a set of inverted rules depend. This procedure helps determine new reachable tasks after semi-inverting the rules. Using reachability for semi-inversion reduces the risk of exponentially increasing the size of $\underline{\mathcal{R}_{f_{IO}}}$; semi-inversion is *polyvariant inversion* of $\mathcal{R}$ in that it may produce several semi-inversions of the same function symbol, namely, one for each input-output index set. Eventually, all reachable semi-inverses are generated then no pending task exist and the algorithm returns the self-contained semi-inverted system $\underline{\mathcal{R}_{f_{IO}}}$.

At this point, as an add-on, the type of the new rewrite system can be syntactically checked. For example, if none of the semi-inverted rules contains an extra

[6] Equivalent to left-to-right deterministic EV-free CCSs with orthogonal rules.

```
seminv(Pend, Done) =
  if Pend = ∅ then ∅ else
  // choose a pending task for semi-inversion
  (f,I,O) ∈ Pend;
  // semi-invert all rules of f with index sets I and O
  f-Rules_original := { ρ | ρ : l → r ⇐ c ∈ R, root(l) = f };
  f-Rules_inverted := { localinv(ρ,I,O) | ρ ∈ f-Rules_original };
  // update the pending and done sets
  NewDep := getdep(f-Rules_inverted) \ Done;
  f-Rules_inverted ∪ seminv((Pend ∪ NewDep) \ {(f,I,O)}, Done ∪ {(f,I,O)});
getdep(Rules) =
  { (f,I,O) | l → r ⇐ c ∈ Rules, l_i → r_i ∈ c, root(l_i) = f_IO };
```

Fig. 3. Recursive semi-inversion algorithm.

variable, then the system is marked as EV-free, and if all function symbols in the CSS are defined by orthogonal rules, then this system corresponds to a first-order functional program. This is the strength of using conditional term rewriting systems as a foundation for studying semi-inversion: they smoothly model inversion problems across a range of different important programming paradigms.

The algorithm *terminates* for any (f,I,O) because the numbers of function symbols and their possible index sets are finite for any given $\mathcal{R}$. In each recursion, a task is semi-inverted and moved from `Pend` to `Done`. Eventually, no more tasks can be added to `Pend` that are not already in `Done`, and the algorithm terminates.

Invocation of the semi-inverter in Fig. 3 is done by $\mathsf{seminv}(\{(f,I,O)\},\ \emptyset)_{\mathcal{R}}$, where the read-only $\mathcal{R}$ is global for the sake of simplicity. A new system $\underline{\mathcal{R}}_{\underline{f}_{IO}}$ with all semi-inverted functions *reachable* from the initial task (f,I,O) is returned.

Definition 3 (semi-inverter). *Given a CCS $\mathcal{R}$, a defined function symbol $f/n/m \in \mathcal{D}$, and two index sets $I \subseteq \{1,\ldots,n\}$ and $O \subseteq \{1,\ldots,m\}$, the semi-inverter in Fig. 3 yields the CCS*

$$\underline{\mathcal{R}}_{\underline{f}_{IO}} = \mathsf{seminv}(\{(f,I,O)\},\emptyset)_{\mathcal{R}}.$$

Note that if the initial pending set contains two or more tasks, they are semi-inverted together by seminv, which may be useful for practical reasons.

$$\underline{\mathcal{R}}_{\underline{f}_{IO}} \cup \underline{\mathcal{R}}_{\underline{g}_{I'O'}} = \mathsf{seminv}(\{(f,I,O)\} \cup \{(g,I',O')\},\emptyset)_{\mathcal{R}}.$$

4.2 Local Semi-Inversion of Conditional Rules

The form of the rules with a rule head followed by a sequence of flat conditions considerably simplifies the local inversion. Given the index sets I and O, a rule

$$(l \to r \Leftarrow l_1 \to r_1 \wedge \ldots \wedge l_k \to r_k)$$

is locally semi-inverted into a *left-to-right deterministic* rule

$$(l' \to r' \Leftarrow l'_1 \to r'_1 \wedge \ldots \wedge l'_k \to r'_k),$$

i.e., satisfying $\mathcal{V}ar(l', r'_1, \ldots, r'_{i-1}) \supseteq \mathcal{V}ar(l'_i)$ for all $1 \leq i \leq k$.

```
localinv(f(p1,...,pn) → ⟨q1,...,qm⟩ ⇐ c, {i1,...,ia}, {o1,...,ob}) =
  // semi-invert the rule head and label the function symbol
  {ia+1,..., in}  := {1,...,n} \ {i1,...,ia}
  {ob+1,..., om}  := {1,...,m} \ {o1,...,ob}
  lhs := f_{i1,...,ia}{o1,...,ob}(pi1,...,pia,qo1,...,qob)
  rhs := ⟨pia+1,...,pin,qob+1,...,qom⟩
  // locally invert the conditions after reordering
  Var := Var(lhs)
  c'  := heuristic(c, Var)     // reorder conditions of rule
  c'' := localinvc(c',Var)     // reorder terms in conditions
  // return the inverted rule
  lhs → rhs ⇐ c''
localinvc(c, Var) = case c of
  // if no condition, then return the empty condition
  ϵ  =>  ϵ
  // else invert the left-most condition
  f(p1,...,pn) → ⟨q1,...,qm⟩ ∧ Restc  =>
    // build the index sets
    {i1,  ..., ia}  := {i | i ∈ {1,...,n}, Var(pi) ⊆ Var}
    {ia+1,..., in}  := {1,...,n} \ {i1,...,ia}
    {o1,  ..., ob}  := {o | o ∈ {1,...,m}, Var(qo) ⊆ Var}
    {ob+1,..., om}  := {1,...,m} \ {o1,...,ob}
    // locally invert and label the left-most condition
    lhs := f_{i1,...,ia}{o1,...,ob}(pi1,...,pia,qo1,...,qob)
    rhs := ⟨pia+1,...,pin,qob+1,...,qom⟩
    // return the inverted conditions
    lhs → rhs ∧ localinvc(Restc, Var ∪ Var(rhs))
```

Fig. 4. Local inversion of a conditional rule.

Local inversion (Fig. 4) first generates the new head $l' \to r'$ by rearranging the terms of l and r according to I and O as required for semi-inversion (Def. 2). To ensure that the new rule is left-to-right deterministic, we transform the conditions $l_1 \to r_1 \wedge \ldots \wedge l_k \to r_k$ from left to right such that all terms in the ith condition ($l_i \to r_i$) that depend only on the already *known variables* $\mathcal{V}ar(l, r_1, \ldots, r_{i-1})$ are moved to the new left-hand side l'_i and all other terms are moved to the new right-hand side r'_i (ordered by increasing index). Therefore, l'_i contains terms that depend on known variables (including ground terms), and r'_i contains all the other terms. At the same time, the function symbol $\underline{f_i}$ at the root of l'_i is labeled with the corresponding input-output sets. This transformation is repeated recursively from left to right for each condition while updating the set of known variables. In this way, the semi-inverted rule becomes left-to-right deterministic, and each new condition $l'_i \to r'_i$ uses the maximum number of known terms in l'_i. This step is not necessary for correctness but reduces the search space of the intended reduction strategy.

4.3 A Heuristic Approach to Reordering Conditions

We have chosen a greedy heuristic to reorder the conditions in a rule before semi-inverting them, which works surprisingly well. The procedure heuristic (shown in

```
heuristic(c, KnownVar) = case c of
  // if no condition, then return the empty condition
  ϵ  =>  ϵ
  // else find condition with highest percentage of known parameters
  l1 → r1  ∧ ...∧ lk → rk   =>
    // determine percentages
    Percent1  := percent(l1 → r1, KnownVar)
    ...          ...
    Percentk  := percent(lk → rk, KnownVar)
    (Pi,i)    := maxPercent((Percent1,1),...,(Percentk,k))
    // select condition, reorder remaining conditions in updated variable set
    li → ri  ∧ heuristic(c \ (li → ri), KnownVar ∪ Var(li,ri))

percent(f(p1, ..., pn) → ⟨q1, ..., qm⟩, KnownVar) =
  // determine index sets of known parameters
  I := {i | i ∈ {1, ..., n}, Var(pi) ⊆ KnownVar }
  O := {j | j ∈ {1, ..., m}, Var(qj) ⊆ KnownVar }
  // return percentage of known parameters
  (|I| + |O|) / (m + n)        // known = |I| + |O|, total = m + n
```

Fig. 5. A greedy heuristic for reordering conditions.

Fig. 5) reorders the conditions according to the *percentages of known parameters* such that the condition with the highest percentage comes first. This procedure dynamically updates the set of known variables each time a condition is moved to the head of the sequence and recursively applies the reordering to the remaining conditions. Clearly, different sets of known variables can lead to different orders of the conditions. The intention with this heuristic is to syntactically exploit as much known information as possible without having to rely on an extra analysis.

Other reordering methods could be used instead. The algorithm by Mogensen [14], which semi-inverts non-overlapping rules in a guarded equational language without extra variables, searches through *all possible semi-inversions* and uses additional semantic information about primitive operators. An exhaustive search will find better orders than a local heuristic, but the search will take more time. This is the familiar trade-off between the accuracy and run time of a program analysis. The heuristic always finds a reordering (perhaps leading to extra variables), while the semi-inverter [14] may halt with no answer due to the limitations of the language —a later inverter [15] allows functional parameters.

There is no fixed order of conditions that avoids extra variables for all possible semi-inversions of a given rewrite system. Our experiments show that the heuristic usually improves the resulting semi-inversion, but it can also be deceived, as shown in the following example.

Example 6. Given a system consisting of `r1`–`r3`, the semi-inversion of `test` w.r.t. $I = \{1\}$ and $O = \{2\}$ yields the system `s1`–`s3`, which has an extra variable in function $\underline{\texttt{fst}}_{\{1\}\{1\}}$, whereas one produced without the heuristic would be EV-free.

```
r1: test(x,y) → ⟨w,z⟩     ⇐ copy(x,y) → ⟨w,z⟩ ∧ fst(x,y) → ⟨z⟩
r2: fst(x,y) → ⟨x⟩           r3: copy(x,y) → ⟨x,y⟩

s1: test{1}{2}(x,z) → ⟨y,w⟩ ⇐ fst{1}{1}(x,z) → ⟨y⟩ ∧ copy{1,2}{2}(x,y,z) → ⟨w⟩
s2: fst{1}{1}(x,x) → ⟨y⟩       s3: copy{1,2}{2}(x,y,y) → ⟨x⟩
```

4.4 Correctness of the Semi-Inversion Algorithm

The correctness of the semi-inversion algorithm is proven by first defining $\underline{\mathcal{R}}_{all}$ by semi-inverting all possible semi-inversion tasks for the function symbols in $\mathcal{R}$.

Definition 4 ($\underline{\mathcal{R}}_{all}$). *Let $\mathcal{R}$ be a CCS, and let $P = \{(f, I, O) \mid f/n/m \in \mathcal{D}, I \subseteq \{1, \ldots, n\}, O \subseteq \{1, \ldots, m\}\}$ be the pending set consisting of all semi-inversion tasks of all function symbols defined in $\mathcal{R}$. Then, we define*

$$\underline{\mathcal{R}}_{all} = \mathsf{seminv}(P, \emptyset)_{\mathcal{R}}.$$

The following theorem can be proven in two steps: First, the rewrite steps of f are in $\mathcal{R}$ if and only if the rewrite steps of its semi-inverse $\underline{f}_{IO}$ are in $\underline{\mathcal{R}}_{all}$, and secondly, the rewrite steps of $\underline{f}_{IO}$ are in $\underline{\mathcal{R}}_{all}$ if and only if they are in $\underline{\mathcal{R}}_{\underline{f}_{IO}}$. The proofs are omitted due to lack of space.

Theorem 1. *Let $\mathcal{R}$ be a CCS, and let $\underline{\mathcal{R}}_{\underline{f}_{IO}} = \mathsf{seminv}(\{(f, I, O)\}, \emptyset)_{\mathcal{R}}$ for a function symbol $f/n/m \in \mathcal{D}$ and index sets $I \subseteq \{1, \ldots, n\}$ and $O \subseteq \{1, \ldots, m\}$. Then, $\underline{\mathcal{R}}_{\underline{f}_{IO}}$ is a semi-inverse of $\mathcal{R}$ w.r.t. f, I, and O.*

5 Application of the Semi-Inverter

The semi-inverter has been fully implemented (in Prolog), and in the following, we will present the results: a series of semi-inversions of a discrete simulation of a free fall, each solving a different problem, a decrypter from a simple encrypter, and the inversion of an inverter for a reversible programming language.

5.1 Discrete Simulation of a Free Fall

We consider a discrete simulation of an object that falls through a vacuum [4] and use semi-inversion to generate four new programs, each solving a different aspect. The simulation is defined by the equations $v_t = v_{t-1} + g$ and $h_t = h_{t-1} - v_t + g/2$, where $g \approx 10\, m/s^2$ is the approximate gravitational acceleration. The following system $\mathtt{fall}_0$ yields the object's velocity v_t ($\mathtt{v}$) and height h_t ($\mathtt{h}$) at time t, given t ($\mathtt{t}$) and initial velocity v_0 ($\mathtt{v_0}$) and height h_0 ($\mathtt{h_0}$).

$\mathtt{fall_0(v,h,0)} \rightarrow \langle \mathtt{v,h} \rangle$

$\mathtt{fall_0(v_0,h_0,s(t))} \rightarrow \langle \mathtt{v,h} \rangle \Leftarrow$
$\quad \mathtt{add(v_0, s^5(0))} \rightarrow \langle \mathtt{v_n} \rangle \wedge$
$\quad \mathtt{height(h_0,v_n)} \rightarrow \langle \mathtt{h_n} \rangle \wedge$
$\quad \mathtt{fall_0(v_n,h_n,t)} \rightarrow \langle \mathtt{v,h} \rangle$

$\mathtt{height(h_0,v_n)} \rightarrow \langle \mathtt{h_n} \rangle \Leftarrow$
$\quad \mathtt{add(h_0,s^5(0))} \rightarrow \langle \mathtt{h_{temp}} \rangle \wedge\ \mathtt{sub(h_{temp},v_n)} \rightarrow \langle \mathtt{h_n} \rangle$

Original	$\mathtt{fall}_0\colon\ (\mathtt{v_0}, \mathtt{h_0}, \mathtt{t}) \rightarrow (\mathtt{v}, \mathtt{h})$
Full inv.	$\mathtt{fall}_1\colon\ (\mathtt{v}, \mathtt{h}) \rightarrow (\mathtt{v_0}, \mathtt{h_0}, \mathtt{t})$
Partial inv.	$\mathtt{fall}_2\colon\ (\mathtt{t}, \mathtt{v}, \mathtt{h}) \rightarrow (\mathtt{v_0}, \mathtt{h_0})$
Semi-inv. #1	$\mathtt{fall}_3\colon\ (\mathtt{v_0}, \mathtt{t}, \mathtt{h}) \rightarrow (\mathtt{h_0}, \mathtt{v})$
Semi-inv. #2	$\mathtt{fall}_4\colon\ (\mathtt{v_0}, \mathtt{t}, \mathtt{v}) \rightarrow (\mathtt{h_0}, \mathtt{h})$

The system fall_0 is geared towards solving the 'forward' problem, while finding a solution to the 'backward' problem of determining the origin of a fall may be equally interesting, it requires a new set of rules. Full inversion algorithms can transform fall_0 into a new system fall_1 solving the backward problem, while other problems require partial or semi-inversion.

These four programs $\mathtt{fall}_1$ to $\mathtt{fall}_4$ are successfully generated by the semi-inversion algorithm as shown in Fig. 6 (dependency functions are omitted for clarity). The difference between the order of the conditions in the four inversions indicates that the heuristic has taken action.

$\texttt{fall}_1\texttt{(v,h)} \rightarrow \langle \texttt{v,h,0} \rangle$
$\texttt{fall}_1\texttt{(v,h)} \rightarrow \langle \texttt{v}_0\texttt{,h}_0\texttt{,s(t)} \rangle \Leftarrow$
$\quad \texttt{fall}_1\texttt{(v,h)} \rightarrow \langle \texttt{v}_n\texttt{,h}_n\texttt{,t} \rangle \wedge$
$\quad \underline{\texttt{add}}_{\{2\}\{1\}}\texttt{(s}^{10}\texttt{(0),v}_n\texttt{)} \rightarrow \langle \texttt{v}_0 \rangle \wedge$
$\quad \underline{\texttt{height}}_{\{2\}\{1\}}\texttt{(v}_n\texttt{,h}_n\texttt{)} \rightarrow \langle \texttt{h}_0 \rangle$

$\texttt{fall}_2\texttt{(0,v,h)} \rightarrow \langle \texttt{v,h} \rangle$
$\texttt{fall}_2\texttt{(s(t),v,h)} \rightarrow \langle \texttt{v}_0\texttt{,h}_0 \rangle \Leftarrow$
$\quad \texttt{fall}_2\texttt{(t,v,h)} \rightarrow \langle \texttt{v}_n\texttt{,h}_n \rangle \wedge$
$\quad \underline{\texttt{add}}_{\{2\}\{1\}}\texttt{(s}^{10}\texttt{(0),v}_n\texttt{)} \rightarrow \langle \texttt{v}_0 \rangle \wedge$
$\quad \underline{\texttt{height}}_{\{2\}\{1\}}\texttt{(v}_n\texttt{,h}_n\texttt{)} \rightarrow \langle \texttt{h}_0 \rangle$

$\texttt{fall}_3\texttt{(v,0,h)} \rightarrow \langle \texttt{h,v} \rangle$
$\texttt{fall}_3\texttt{(v}_0\texttt{,s(t),h)} \rightarrow \langle \texttt{h}_0\texttt{,v} \rangle \Leftarrow$
$\quad \texttt{add(v}_0\texttt{,s}^{10}\texttt{(0))} \rightarrow \langle \texttt{v}_n \rangle) \wedge$
$\quad \texttt{fall}_3\texttt{(v}_n\texttt{,t,h)} \rightarrow \langle \texttt{h}_n\texttt{,v} \rangle \wedge$
$\quad \underline{\texttt{height}}_{\{2\}\{1\}}\texttt{(v}_n\texttt{,h}_n\texttt{)} \rightarrow \langle \texttt{h}_0 \rangle$

$\texttt{fall}_4\texttt{(v,0,v)} \rightarrow \langle \texttt{h,h} \rangle$
$\texttt{fall}_4\texttt{(v}_0\texttt{,s(t),v)} \rightarrow \langle \texttt{h}_0\texttt{,h} \rangle \Leftarrow$
$\quad \texttt{add(v}_0\texttt{,s}^{10}\texttt{(0))} \rightarrow \langle \texttt{v}_n \rangle \wedge$
$\quad \texttt{fall}_4\texttt{(v}_n\texttt{,t,v)} \rightarrow \langle \texttt{h}_n\texttt{,h} \rangle \wedge$
$\quad \underline{\texttt{height}}_{\{2\}\{1\}}\texttt{(v}_n\texttt{,h}_n\texttt{)} \rightarrow \langle \texttt{h}_0 \rangle$

Fig. 6. The $\texttt{fall}_0$, its full inversion $\texttt{fall}_1$, the partial inversion $\texttt{fall}_2$ and the two semi-inversions $\texttt{fall}_3$ and $\texttt{fall}_4$.

5.2 Encrypter and Decrypter

The automatic generation of decrypters is fascinating. The following encrypter is a modification of a simple encryption method suggested by Mogensen. It produces an encrypted text `z:zs` given a text formed as an integer list `x:xs` and a key `key`. Encryption cleans the key by mod 4, adds the new value to the first character, and repeats the process recursively for the rest of the text (the modification is that we use mod 4 instead of mod 256, as mod 256 consists of 257 rules). The encrypter is given in Fig. 7 (on the left), and the decrypter is a partial inversion of it with respect to $I = \{2\}$ and $O = \{1\}$, as shown in Fig. 7 (on the right). The decrypter ($\underline{\texttt{encrypt}}_{\{2\}\{1\}}$) produces the decrypted text `x:xs` given the key `key` and the encrypted text `z:zs`. Note that `mod4` is equivalent to $\underline{\texttt{mod4}}_{\{1\}\emptyset}$.

Original encrypter:	Generated decrypter:
`encrypt(nil,key)` $\rightarrow \langle$`nil`$\rangle$	$\underline{\texttt{encrypt}}_{\{2\}\{1\}}$`(key,nil)` $\rightarrow \langle$`nil`$\rangle$
`encrypt(x:xs,key)` $\rightarrow \langle$`z:zs`$\rangle \Leftarrow$	$\underline{\texttt{encrypt}}_{\{2\}\{1\}}$`(key,z:zs)` $\rightarrow \langle$`x:xs`$\rangle \Leftarrow$
`mod4(key)` $\rightarrow \langle$`y`$\rangle$,	$\underline{\texttt{mod4}}_{\{1\}\emptyset}$`(key)` $\rightarrow \langle$`y`$\rangle$,
`add(x,y)` $\rightarrow \langle$`z`$\rangle$,	$\underline{\texttt{add}}_{\{2\}\{1\}}$`(y,z)` $\rightarrow \langle$`x`$\rangle$,
`encrypt(xs,key)` $\rightarrow \langle$`zs`$\rangle$	$\underline{\texttt{encrypt}}_{\{2\}\{1\}}$`(key,zs)` $\rightarrow \langle$`xs`$\rangle$
`mod4(0)` $\rightarrow \langle$`0`$\rangle$	$\underline{\texttt{mod4}}_{\{1\}\emptyset}$`(0)` $\rightarrow \langle$`0`$\rangle$
`mod4(s(0))` $\rightarrow \langle$`s(0)`$\rangle$	$\underline{\texttt{mod4}}_{\{1\}\emptyset}$`(s(0))` $\rightarrow \langle$`s(0)`$\rangle$
`mod4(s`^{2}`(0))` $\rightarrow \langle$`s`^{2}`(0)`$\rangle$	$\underline{\texttt{mod4}}_{\{1\}\emptyset}$`(s`^{2}`(0))` $\rightarrow \langle$`s`^{2}`(0)`$\rangle$
`mod4(s`^{3}`(0))` $\rightarrow \langle$`s`^{3}`(0)`$\rangle$	$\underline{\texttt{mod4}}_{\{1\}\emptyset}$`(s`^{3}`(0))` $\rightarrow \langle$`s`^{3}`(0)`$\rangle$
`mod4(s`^{4}`(x))` $\rightarrow \langle$`w0`$\rangle \Leftarrow$ `mod4(x)` $\rightarrow \langle$`w0`$\rangle$	$\underline{\texttt{mod4}}_{\{1\}\emptyset}$`(s`^{4}`(x))` $\rightarrow \langle$`w0`$\rangle \Leftarrow$ $\underline{\texttt{mod4}}_{\{1\}\emptyset}$`(x)` $\rightarrow \langle$`w0`$\rangle$

Fig. 7. Inversion of a simple encrypter into a decrypter.

5.3 Inverted Inverter

The Janus language is a reversible language [23] where functions can be both called (executed in a forward direction) and uncalled (executed in a backward direction). An inverter for the Janus language creates procedures that are equal to uncalling the original procedure. Since Janus is a reversible language, the full inversion of such a Janus-inverter is equivalent to itself. A Janus-inverter can be described as a left-to-right deterministic EV-free CCS, and in Fig. 8 (on the left), we have given such an inverter for a subset of the language.

A full inversion of the inverter for the reversible Janus language is equivalent to itself. The result of the semi-inversion algorithm is the Janus inverter shown in Fig. 8 (on the right). If we assume that `invName` and its full inversion $\underline{\texttt{invName}}$ are equivalent, then the produced and original rules are equivalent up to the variable naming and order of the parameters.

Janus-Inverter	Fully inverted Janus-inverter
$\texttt{inv(proc(name,progr))} \rightarrow \langle \texttt{proc(u,v)} \rangle \Leftarrow$ $\quad \texttt{invName(name)} \rightarrow \langle \texttt{u} \rangle \wedge \texttt{inv(progr)} \rightarrow \langle \texttt{v} \rangle$	$\underline{\texttt{inv}}\texttt{(proc(u,v))} \rightarrow \langle \texttt{proc(name,progr)} \rangle \Leftarrow$ $\quad \underline{\texttt{invName}}\texttt{(u)} \rightarrow \langle \texttt{name} \rangle \wedge \underline{\texttt{inv}}\texttt{(v)} \rightarrow \langle \texttt{progr} \rangle$
$\texttt{inv(+=(x,y))} \rightarrow \langle \texttt{-=(x,y)} \rangle$ $\texttt{inv(-=(x,y))} \rightarrow \langle \texttt{+=(x,y)} \rangle$ $\texttt{inv(<=>(x,y))} \rightarrow \langle \texttt{<=>(x,y)} \rangle$	$\underline{\texttt{inv}}\texttt{(-=(x,y))} \rightarrow \langle \texttt{+=(x,y)} \rangle$ $\underline{\texttt{inv}}\texttt{(+=(x,y))} \rightarrow \langle \texttt{-=(x,y)} \rangle$ $\underline{\texttt{inv}}\texttt{(<=>(x,y))} \rightarrow \langle \texttt{<=>(x,y)} \rangle$
$\texttt{inv(if(}x_1\texttt{,}y_1\texttt{,}y_2\texttt{,}x_2\texttt{))} \rightarrow \langle \texttt{if(}x_2\texttt{,}z_1\texttt{,}z_2\texttt{,}x_1\texttt{)} \rangle \Leftarrow$ $\quad \texttt{inv(}y_1\texttt{)} \rightarrow \langle u_1 \rangle \wedge \texttt{inv(}y_2\texttt{)} \rightarrow \langle v_2 \rangle$ $\texttt{inv(loop(}x_1\texttt{,}y_1\texttt{,}y_2\texttt{,}x_2\texttt{))} \rightarrow \langle \texttt{loop(}x_2\texttt{,}z_1\texttt{,}v_2\texttt{,}x_1\texttt{)} \rangle \Leftarrow$ $\quad \texttt{inv(}y_1\texttt{)} \rightarrow \langle z_1 \rangle \wedge \texttt{inv(}y_2\texttt{)} \rightarrow \langle z_2 \rangle$	$\underline{\texttt{inv}}\texttt{(if(}x_2\texttt{,}z_1\texttt{,}z_2\texttt{,}x_1\texttt{))} \rightarrow \langle \texttt{if(}x_1\texttt{,}y_1\texttt{,}y_2\texttt{,}x_2\texttt{)} \rangle \Leftarrow$ $\quad \underline{\texttt{inv}}\texttt{(}z_1\texttt{)} \rightarrow \langle y_1 \rangle \wedge \underline{\texttt{inv}}\texttt{(}z_2\texttt{)} \rightarrow \langle y_2 \rangle$ $\underline{\texttt{inv}}\texttt{(loop(}x_2\texttt{,}z_1\texttt{,}z_2\texttt{,}x_1\texttt{))} \rightarrow \langle \texttt{loop(}x_1\texttt{,}y_1\texttt{,}y_2\texttt{,}x_2\texttt{)} \rangle \Leftarrow$ $\quad \underline{\texttt{inv}}\texttt{(}z_1\texttt{)} \rightarrow \langle y_1 \rangle \wedge \underline{\texttt{inv}}\texttt{(}z_2\texttt{)} \rightarrow \langle y_2 \rangle$
$\texttt{inv(call(name))} \rightarrow \langle \texttt{call(u)} \rangle \Leftarrow$ $\quad \texttt{invName(name)} \rightarrow \langle \texttt{u} \rangle$ $\texttt{inv(uncall(name))} \rightarrow \langle \texttt{uncall(u)} \rangle \Leftarrow$ $\quad \texttt{invName(name)} \rightarrow \langle \texttt{u} \rangle$	$\underline{\texttt{inv}}\texttt{(call(u))} \rightarrow \langle \texttt{call(name)} \rangle \Leftarrow$ $\quad \underline{\texttt{invName}}\texttt{(u)} \rightarrow \langle \texttt{name} \rangle$ $\underline{\texttt{inv}}\texttt{(uncall(u))} \rightarrow \langle \texttt{uncall(name)} \rangle \Leftarrow$ $\quad \underline{\texttt{invName}}\texttt{(u)} \rightarrow \langle \texttt{name} \rangle$
$\texttt{inv(sequence(x,y))} \rightarrow \langle \texttt{sequence(u,v)} \rangle \Leftarrow$ $\quad \texttt{inv(y)} \rightarrow \langle \texttt{u} \rangle \wedge \texttt{inv(x)} \rightarrow \langle \texttt{v} \rangle$ $\texttt{inv(skip)} \rightarrow \langle \texttt{skip} \rangle$	$\underline{\texttt{inv}}\texttt{(sequence(u,v))} \rightarrow \langle \texttt{sequence(x,y)} \rangle \Leftarrow$ $\quad \underline{\texttt{inv}}\texttt{(u)} \rightarrow \langle \texttt{y} \rangle \wedge \underline{\texttt{inv}}\texttt{(v)} \rightarrow \langle \texttt{x} \rangle$ $\underline{\texttt{inv}}\texttt{(skip)} \rightarrow \langle \texttt{skip} \rangle$

Fig. 8. The Janus-inverter `inv` expressed as CCS rules, and its full inverse $\underline{\texttt{inv}}$.

6 Conclusion

We have shown that polyvariant semi-inversion can be performed for conditional constructor systems (CCSs) that are highly useful for modeling several language paradigms, including logic, functional, and reversible languages. Notably, we have shown that local inversion and a straightforward heuristics suffice for performing the general form of inversion with interesting results. This approach can be used for transformation problems ranging from the inversion of a simple encryption algorithm or a physical discrete-event simulation to a program inverter for a reversible language. We have also implemented the algorithm and shown its correctness. The algorithm makes use of a simple syntactic heuristic that produces good results in our experiments. In full inversion, some auxiliary functions may be partially inverted [17, p. 145], that is, their inversion may also benefit from the algorithm in this paper. Furthermore, the structure of the algorithm is modular such that the heuristic can be easily replaced. In regard to future work, it could be interesting to vary the heuristics with respect to the type of inversion considering that reversing conditions suffices for full inversion; e.g., by parameterization, we might capture other inversion algorithms in this framework.

Acknowledgment The authors wish to thank the anonymous reviewers for their insightful comments. Support by the EU COST Action IC1405 is acknowledged.

References

1. S. M. Abramov, R. Glück. The universal resolving algorithm and its correctness: inverse computation in a functional language. *Science of Computer Programming*, 43(2-3):193–229, 2002.
2. J. M. Almendros-Jiménez, G. Vidal. Automatic partial inversion of inductively sequential functions. In Z. Horváth, et al. (eds.), *Implementation and Application of Functional Languages. Proceedings*, LNCS 4449, 253–270. Springer, 2007.
3. S. Antoy. Programming with narrowing: A tutorial. *Journal of Symbolic Computation*, 45(5):501–522, 2010. (version: June 2017, update).
4. H. B. Axelsen, R. Glück, T. Yokoyama. Reversible machine code and its abstract processor architecture. In V. Diekert, M. V. Volkov, A. Voronkov (eds.), *Computer Science – Theory and Applications. Proceedings*, LNCS 4649, 56–69. Springer, 2007.
5. F. Baader, T. Nipkow. *Term Rewriting and All That*. Cambridge Univ. Press, 1998.
6. E. W. Dijkstra. Program inversion. In F. L. Bauer, M. Broy (eds.), *Program Construction: International Summer School*, LNCS 69, 54–57. Springer-Verlag, 1978.
7. D. Eppstein. A heuristic approach to program inversion. In A. K. Joshi (ed.), *IJCAI-85. Proceedings*, Vol. 1, 219–221. Morgan Kaufmann, Inc., 1985.
8. R. Glück, M. Kawabe. A program inverter for a functional language with equality and constructors. In A. Ohori (ed.), *Programming Languages and Systems. Proceedings*, LNCS 2895, 246–264. Springer, 2003.
9. R. Glück, M. Kawabe. A method for automatic program inversion based on LR(0) parsing. *Fundamenta Informaticae*, 66:367–395, 2005.
10. R. Glück, M. Kawabe. Revisiting an automatic program inverter for Lisp. *SIGPLAN Notices*, 40(5):8–17, 2005.
11. R. Glück, V. F. Turchin. Application of metasystem transition to function inversion and transformation. In *ISSAC. Proceedings*, 286–287. ACM Press, 1990.
12. M. Hanus. The integration of functions into logic programming: From theory to practice. *The Journal of Logic Programming*, 19-20(Suppl. 1):583–628, 1994.
13. J. McCarthy. The inversion of functions defined by Turing machines. In C. Shannon, J. McCarthy (eds.), *Automata Studies*, 177–181. Princeton Univ. Press, 1956.
14. T. Æ. Mogensen. Semi-inversion of guarded equations. In R. Glück, M. Lowry (eds.), *GPCE. Proceedings*, LNCS 3676, 189–204. Springer, 2005.
15. T. Æ. Mogensen. Semi-inversion of functional parameters. In *PEPM. Proceedings*, 21–29. ACM, 2008.
16. M. Nagashima, M. Sakai, T. Sakabe. Determinization of conditional term rewriting systems. *Theoretical Computer Science*, 464:72–89, 2012.
17. N. Nishida. *Transformational Approach to Inverse Computation in Term Rewriting*. PhD thesis, Graduate School of Engineering, Nagoya University, 2004.
18. N. Nishida, M. Sakai, T. Sakabe. Partial inversion of constructor term rewriting systems. In J. Giesl (ed.), *RTA. Proceedings*, LNCS 3467, 264–278. Springer, 2005.
19. N. Nishida, G. Vidal. Program inversion for tail recursive functions. In *RTA. Proceedings*, *LIPIcs*, Vol. 10, 283–298. Schloss Dagstuhl, 2011.
20. N. Nishida, G. Vidal. Characterizing compatible view updates in syntactic bidirectionalization. In *RC. Proceedings*, LNCS 11497, 67–83. Springer, 2019.
21. E. Ohlebusch. Transforming conditional rewrite systems with extra variables into unconditional systems. In H. Ganzinger, et al. (eds.), *Logic Programming and Automated Reasoning. Proceedings*, LNCS 1705, 111–130. Springer, 1999.
22. E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002.
23. T. Yokoyama, R. Glück. A reversible programming language and its invertible self-interpreter. In *PEPM. Proceedings*, 144–153. ACM, 2007.

Natural language compositionality in Coq*

Erkki Luuk

Institute of Computer Science, University of Tartu
erkkil@gmail.com

Abstract. The paper presents a new formalism (a minimalist type system) for modeling natural language compositionality over the three levels of compositional semantics, syntax and morphology. The formalism reduces compositionality to a single level by using a subclass of compound types (syntactic compounds of multiple types or their terms) and types for morphemes. The paper includes a detailed comparison with earlier similar approaches and an overview of the implementation of the formalism on a structurally diverse fragment of English in the Coq proof assistant.

Keywords: natural language · compositionality · type theory · Coq.

1 Introduction

The last 70 years or so have seen many concentrated efforts of natural language (NL) formalization, many of which have involved the use of types and functions, either on paper [16,20] or, starting from this century, also as computer code [3,5,26]. In this paper, we introduce a new formalism, a minimalist type system for modeling NL compositionality. We will also present an implementation of the formalism in the Coq proof assistant. The paper is organized as follows. Section 2 gives a motivating example, followed by an overview of the formalism and its novel contributions in section 3. Section 4 introduces the Coq development of NLC on a fragment of English. An extensive comparison of NLC with existing formalisms is in section 5. The paper concludes with a short discussion in the eponymous section.

2 The quest for a parsimonious representation

Traditionally, NL semantics has been modeled in nth order logic, for $n \leq 4$. To understand why this should be so, one should recall the model-theoretic definition of the meaning of a declarative sentence as its truth conditions (conditions on which the sentence would be true) [8]. The main problem with the definition is that it cannot be readily extended to other syntactic categories, such as e.g. morphemes, words, phrases and interrogative sentences, all of which have

* This work has been supported by IUT20-56 and European Regional Development Fund through CEES.

meaning. For example, the notion of truth-conditions of a morpheme, word or phrase can be scarcely upheld, unless we arbitrarily conflate the meanings of e.g. *cat*, *a cat* and *there is a cat* as $\exists x.\,\mathrm{cat}(x) := \mathrm{True}$. Clearly, the latter signals a failure of model-theoretic semantics to distinguish between the meanings of the constructions. This aside, the notation is awkward, which becomes clear if we compare the following:

(1) $\exists x.\,\texttt{red}(x) \wedge \texttt{ball}(x)$
(2) `a (red ball)`
(3) $\exists x.\,\texttt{veryred}(x) \wedge \texttt{ball}(x)$
(4) $\exists x.\,\texttt{very}(\texttt{red}(x)) \wedge \texttt{ball}(x)$
(5) `a ((very red) ball)`

First, as mentioned above, (1) fails to capture the semantic differences between *red ball*, *a red ball* and *there is a red ball*. (For (3)–(4), the argument is similar.) Second, as soon as some complexity is added, we need a higher-order logic (4) or must deal with a lossy compression in the first-order logic (3) (the example is lossy as it does not even distinguish between different words)[1]. Third, nothing in *red ball* justifies the existential quantification (at least not in the way it seems justified in *a red ball*). By contrast, the formulas (2) and (5) are free from these problems **and** have a near-optimal correspondence to NL.

Another staple of a (not only) type-theoretical NL modeling has been a **separate** representation of all three levels of a modality-independent NL description (i.e. morphology[2], syntax and semantics). In most (all?) cases, this either means omitting some of the levels or deploying an impressive array of theories, formalisms and interfaces. Thus, in one extreme, we have works dealing only with e.g. semantics [1,17,2], while in the other there is a variety of approaches ranging from the categorial tradition [16,27,21,22] to an entire high-level programming language with dependent types [26]. As compared to these alternatives, the key novelty of the present approach is an integrated and parsimonious modeling of morphology, syntax and semantics. The modeling features types for morphemes, polymorphism and a particular subclass of compound types (syntactic compounds of multiple types or their terms), resulting in a representation with a close correspondence to NL ((2) and (5) are examples). A detailed comparison with related approaches is in section 5.

3 NLC

Let us call our formalism NLC (for natural language code, compositionality or C). NLC is a *type system for modeling semantic, syntactic and morphological compositionality* of NL. Hence, its scope is quite wide as compared to dedicated systems (type-driven or otherwise) for modeling one of those three levels, while

[1] The possibilities are not mutually exclusive (see e.g. [20]).
[2] Linguistic morphology studies (the combination of) morphemes, the smallest signs (form-meaning units) in language.

being narrower than that of "maximal" formalisms such as (e.g.) Head-Driven Phrase Structure Grammar (HPSG) that model also word and morpheme orders and lexical semantics. Thus, NLC handles *all* compositionality a NL formalism is expected to, but does not cover syntax, semantics and morphology in their entirety, as constituent orders and lexicon are omitted. The omission is neither accidental nor strictly a deficiency, as it makes sense to model the compositionality together across all levels, since the compositionality is, in a weaker sense, cross-linguistically universal, while lexicon and constitutent orders are language-specific. By the "weaker sense" I mean "universal in all languages that have the (compositional) categories", such as noun, verb, adjective, nominative, accusative, etc., as I do not claim that all languages have all these categories. However, in all languages that have them, adjectives are functions over nouns and not vice versa, adverbs are functions over adjectives and not vice versa, etc., i.e. the general logic of compositionality is universal.

There are ways to test this. First, I suggest the very general principle

(6) A formula must derive a well-typed standalone NL expression at every stage of derivation,

where by a "stage of derivation" is meant a composition. Thus, from the hypothetical parses of *a very red ball*:

(7) `a (very (red ball))`,
(8) `a (very red) ball`,
(9) `a ((very red) ball)`,

(8) is ruled out by (6) already, as *a very red* is not a well-typed standalone NL expression. Second, a type-theoretic test. One observes that *a very red ball*, *a red ball* and *a ball* are all well-typed standalone NL expressions. Thus, `a` : `X` $\rightarrow$ `XP`, and *very red ball*, *red ball*, *ball* : `X`, where `X` and `XP` are at this point arbitrary labels. Then by (7), `very` : `X` $\rightarrow$ `X`, which does not compute, since *very ball* is ill-typed. Given the lack of sensible alternatives, (9) must be correct. The third test is that of modification or "semantic contribution": e.g. *very* modifies (or contributes to) the semantics of *red*, not e.g. vice versa, and adopting the (very sensible) convention of taking the modifier to be a function over that what it modifies, we get `very red` instead of `red very`, `a ball` instead of `ball a`, etc. Notice that the circumstance of English word order aligning with the function application syntax is somewhat accidental, as all possible orders of {subject, object, verb} are present in the world's languages (although not all are equally common) [11].

Returning to the topic of universality, it is easy to observe that the well-typedness is semantically motivated. This is significant, as NL semantics is generally assumed to be universal, owing to the possibility of translation from any one language to another. Thus, with a slightly more general (or abstract) notation, one could even approximate a Universal Grammar with the formulas (a possibility not pursued here).

Finally, since we are dealing with a (weakly) universal, semantically motivated phenomenon, using types to model it is not a long shot, given the common definition of 'type' as a category of semantic value. A remarkable feature of NLC is modeling syntactic and morphological compositionality without specialized notations like trees, phrase structure rules or (as in HPSG) attribute value matrices. In fact, the "look and feel" of NLC is very similar to NL.

As a type system, NLC requires only four rules:

$$\frac{a \in \mathcal{M}^k}{a : T^k} \text{ AT-Intro,}$$

$$\frac{e_0 : T_0^k,\ ...,\ e_m : T_m^k \quad a : T_0^k \to ... \to T_m^k \to T_{m+1}^k}{a(e_0, ..., e_m) : T_{m+1}^k} \text{ FT-Elim,}$$

$$\frac{e_0 : T_0^k,\ ...,\ e_m : T_m^k \quad \vdash \quad a(e_0, ..., e_m) : T_{m+1}^k}{((e_0, ..., e_m) \ \mapsto \ a(e_0, ..., e_m)) \ : T_1^k \to ... \to T_m^k \to T_{m+1}^k} \text{ FT-Intro,}$$

$$\frac{B : T^k \qquad c_0 : C_0^k,\ ...,\ c_{n+1} : C_{n+1}^k \qquad B \mapsto c_0, ..., B \mapsto c_{n+1}}{B : C_0^k..C_{n+1}^k} \text{ LT-Intro.}$$

AT-Intro (atomic type introduction) generates atomic terms of NLC and introduces type variables for them. The rule says that all morphemes have types in NLC (technically: "if a is a morpheme of language k then a has type T^k", where T^k is a proper type variable[3]). FT-Elim (function type elimination) is a computation rule, with $a(e_0, ..., e_m)$ an application and $T_0^k \to ... \to T_{m+1}^k$ the right-associative function type. FT-Intro (function type introduction) is standard and derived from FT-Elim. LT-Intro (lump type introduction) is, together with AT-Intro, NLC-specific. B is a term constant and $C_0^k, ..., C_{n+1}^k$ type constants in NLC, $x \mapsto y$ is a function interpreting x as y, and $C_0^k..C_{n+1}^k$ the notation for a lump type (comprising types C_0^k through C_{n+1}^k, i.e. there must be at least two). The interpretations must not preclude each other[4]. All indices are natural numbers.

Lump types are compound types that satisfy LT-Intro. Compound type is defined as a syntactic compound of multiple types or their terms. For lump types, the exact principle of compounding is irrelevant, as long as it supports function types over the compound types defined by the principle. For example, Appendix B models a tiny NL fragment with lump types implemented as application, record, product or Π-types[5]. Since intersection types are, by definition, a small subclass of lump types, it is semantically natural to encode lump

[3] A proper type is a type that is not a universe.

[4] The interpretations that preclude each other (e.g. the interpretations of *run* as a noun and verb) are dealt with polymorphism instead.

[5] The file is online at `https://gitlab.com/jaam00/nlc/blob/master/compound.v`

types as intersection types in the relatively few languages that have them (e.g. TypeScript, Flow...).

In NLC, functions must be derived manually from NL expressions and the abovementioned logic of composition. In general, if there is a function $f : A \to B$ in NLC, there is no function $g : B \to A$. This follows from observing NL expressions, from the functions of linguistic categories like noun, adjective, etc., and from the general principle of asymmetry in NL composition[6]. But whence the principle? A necessary explanation for it is the fact that function composition is highly non-commutative. Another one is the ease of interpretation brought about by the avoidance of functional symmetry (e.g. while flexibles[7] are ambiguous, I am not aware of one "flexing" between f and g).

3.1 Lump types

The use of lump types is linguistically motivated by most stems belonging to several types, e.g. *john* is a proper name, in nominative case, male, a physical, sentient, limbed, etc. entity, while *run* is a flexible, a noun in nominative or a verb taking a limbed argument. Flexibles are tricky, as they exhibit a true polymorphism, with different readings (e.g. the nominal and verbal readings of *run*) precluding one another. When different typings always coincide, as with e.g. *john* or *nut*, it is far more convenient to pack them together into a lump type than to keep track of n distinct typings. Similarly, it may be preferable to index the case (e.g. nominative) on the lump type rather than to make it a modifer over an argument, as in `NOM john`. Thus, it is desirable to have both lump types and polymorphism. Below are examples of *john* typed as a lump type[8]:

```
Check john: XP0 (hu _) NOM SG (Pn Ml).
Check john: XP0 Phy NOM SG (Pn Ml).
Check limbed john: XP0 Lim NOM SG PN+.
Fail Check limbed john: XP0 Phy NOM SG (Pn Ml).
Fail Check limbed (john: XP0 Phy NOM SG (Pn Ml)).
```

This tells that *john* is of type `XP`[9], human, in nominative, singular, proper name, and male. `XP`s are arguments of verbs and flexibles, and proper names are `XP0`s because there are different ways of deriving an `XP` (e.g. zero-derivation in proper names or prepending an article as in *the man*). Humans have specific selectional restrictions[10] such as physical, sentient, bilogical, animate, limbed, etc. The argument of `hu` tells which one is active in the context. For example, the adjective

[6] There is a lot of dedicated research on this, see e.g. the contents of volumes [9]–[10]

[7] Flexibles are word classes that "flex" between different word classes (e.g. *walk* is a noun and a verb) [19].

[8] The full Coq code for the fragment is at `https://gitlab.com/jaam00/nlc/blob/master/frag.v`

[9] Frequently also reffered to as NP or DP.

[10] Selectional restrictions are the ontological restrictions imposed by a relation on its arguments' types.

limbed and the flexible *throw* require a limbed entity for their first (and in case of the adjective, only) argument. `(hu H_Phy)` reduces to `Phy`. `PN+` is the notation for a proper name with additional (gender) information.

The encoding of lump types as application types has the benefit of (at least visual) simplicity. In Coq, one can usually concoct a simpler notation for a type, but it would be difficult to get a simpler one than this.

While LT-Intro has not been defined before, it is still possible that lump types have been used implicitly. The closest examples I have found are from Coq [3,4] and Grammatical Framework (GF) [26]. GF's set of compound types contains e.g. function, record and table types. The use of function types is computational, i.e. they are not used for storing values (as would be the case with lump types), while records and tables engage concrete (i.e. linearized) syntax and morphology. At least superficially, there is no good match with NLC, which has only abstract morphosyntax. Below is a relevant example of a GF's record and table[11]:

(10)
```
noun "vino" "vini" Masc : {table {Sg => vino ; Pl => vini} :
 Number => Str ; _ : Gender}
```

A table is a finite function[12], in this case from number to string, while the record type has two fields (for the table and gender, respectively). Thus, an Italian noun's type is indexed by number, two strings and gender. Below are relevant examples of records in the Coq implementations:

```
Record Irishdelegate: CN := mkIrishdelegate { c :> delegate; _ : Irish c }.
Record Book: Set := mkBook {Arg:> PhyInfo; Qualia: BookQualia Arg}.
```

The first is a rendering of *Irish delegate* as a record type [3], the second of *book* as another record type [4], a physical/informational object with Pustejovsky's [25] qualia structure. The structure specifies `Qualia` as a human-made physical/informational object (`PhyInfo` and `BookQualia` are also record types).

Of the various indices to the right of `:`, the record `noun "vino" "vini" Masc` could be said to have only type `Gender`, i.e. it does not qualify as a lump type by LT-Intro. In particular, as it represents both `Sg` and `Pl`, it lacks number; similarly, it represents both strings (and a record itself is not a string of the object language). The Coq examples fare better in this respect, as an Irish delegate could be sensibly typed as a delegate and as something Irish (and likewise for a book).

The conclusion is that, differently from `{table {...} ...}`, `Book` and `Irishdelegate` are indeed lump types, although very different from `XP0 _ _ _ _`. Perhaps the main difference is that `Book` and `Irishdelegate` model only semantics[13], while `XP0 _ _ _ _` models syntax, semantics and morphology. Another difference is in their implementations. An obvious advantage of record types is that they occur

[11] Taken from `http://www.grammaticalframework.org/doc/tutorial/gf-tutorial.html#toc64`

[12] A function with a finite enumeration of all argument-value pairs.

[13] I assume that `Irish c` represent something Irish rather than a phrase *Irish* ....

in many programming languages, but there are several alternatives, such as Σ-, Π-, list, application or Cartesian product types (some of which — but not all — can be used for defining record types in some languages [6]).

4 Implementation

NLC has been implemented for an English fragment containing stems, nouns, verbs, flexibles, proper names, pronouns, XPs, adjectives, sentential, adjectival and generic adverbs, determiners, demonstratives, quantifiers, tense-aspect-mood, gender, number and nonfinite markers, cases, adpositions, sentences (both simple and complex), connectives, connective phrases (for substantives, adjectives, adverbs and sentences), copulas, complementizers and selectional restrictions (for physical, informational, limbed, animate, biological and sentient entities). The linguistic categories not formalized in the fragment are gerunds, participles, auxiliary verbs, interrogatives, numerals, negation, mass/count distinction and unspecified selectional restrictions (and possibly others). These are left as a future work.

In general, the implementation has few representatives of each category, as the goal was to model a structurally rich rather than quantitatively extensive fragment. For a structurally rich fragment, a diverse set of correctly composing types must be defined manually. A quantitative extension of this set should be generated semi-automatically, combining resources (tagged corpora or dictionaries) with standalone scripts or machine learning.

4.1 Base types

The implementation was done in Coq (ver. 8.9). We start by defining some categories[14]:

```
Inductive NUM := SG | PLR. (*number: singular, plural*)
Inductive GN := Ml | Fm. (*gender*)
Inductive ST := F | N | PN | PR | Nn (x:GN) | Pn (x:GN) | Pr (x:GN).
(*argument stems: flexible, noun, proper name, pronoun, also w/ gender*)
Parameter NF S: Prop. (*nonfinite, sentence*)
Inductive CA := NOM | ACC | GEN | LAT | LOC | DIR.
(*case/adposition: nominative, accusative, genitive, lative, locative...*)
```

Coq has inductive types, and there are some reasons to prefer them over variables (defined with e.g. `Parameter` in Coq). First, Gallina's (Coq's specification language's) pattern matching works only over inductive types, so we should use them whenever we want it (Coq's official tactic language Ltac[15] has a general syntactic pattern matching). Second, if we want to define proof schemes (e.g. the

[14] The code is taken from Supplement I: https://gitlab.com/jaam00/nlc/blob/master/frag.v

[15] https://coq.inria.fr/distrib/current/refman/proof-engine/ltac.html

Boolean equality) over a type, the type must also be inductive. Finally, differently from inductive types, variables have no term resolution, so inductives are preferable for a better type inference. To give an example, `Inductive NU := SF | PF (x:DD)` declares an inductive type `NU: Type` with two constructors (i.e. canonical terms) `SF: NU` and `PF: DD → NU` (`Type` is the universe of types).

While not much implementationally, the following line is a central one, as it defines our lump types **and** (for exposition purposes, a minimal) phrase architecture:

```
Parameter STM PLU XP: SR -> CA -> NUM -> ST -> Type. (*stem, plural, XP*)
```

Thus, `STM _ _ _ _`, `PLU _ _ _ _`, etc., are our lump types (in Coq, `_` is a placeholder for any admissible type or term). The comments `(*...*)` explain what types of phrases they are: the first the stem phrase, the second plural phrase, etc. Thus we get e.g. `car: STM _ _ _ _` and `PL car: PLU _ _ _ _`, where `PL` is a plural marker (such as *-s* in English).

NL relations (adjectives, determiners, etc.) frequently take different types of phrases as arguments. So we need a device to generalize over n types. For this, we will use canonical structures[16] (canonical instances of record types):

```
(*(in|out)put for ADJ, input to PL...*)
Structure CSTM := cs {gs: SR -> CA -> NUM -> ST -> Type}.
Canonical Structure cs_s: CSTM := cs STM.
Canonical Structure cs_p: CSTM := cs PLU.
```

Now, whenever we need to refer to e.g. the general input/output type for `ADJ`, we write `gs _`. This is known as notation overloading (and will, in this case, reduce to `STM` or `PLU`).

4.2 Selectional restrictions

Selectional restrictions are semantic (or ontological) restrictions imposed by NL relations on their arguments; e.g. *throw* selects for the first argument that is a limbed and the second that is a physical entity. It is straightforward to type something which has an elementary selectional restriction (e.g. *hut*, a physical entity). But what about words like *he* or *john*, which are (at least) physical, sentient, informational, animate, biological and limbed entities? One way to do this is to rely on notation overloading; this is the approach taken in Supplement I. In the more recent Supplement II[17] we take a shorter way, and define selectional restrictions (SRs) as

[16] `https://coq.inria.fr/distrib/current/refman/addendum/canonical-structures.html`. We could also use type classes but canonicals are more minimalist.

[17] `https://gitlab.com/jaam00/nlc/blob/master/cop.v`. Primarily a testbed for new design choices, Supplement II is the more experimental and smaller fragment of the two. Both Supplements have some unique features while being fully compatible otherwise.

```
(*selectional restrictions*)
Inductive SRH := Phy | Sen | Inf | Ani | Bio. (*humans*)
Inductive SRV := Phy_v | Ani_v. (*vehicles*)
Inductive SR := srh (x:SRH) | hum_sr (x:SRH):> SR | veh_sr (x:SRV):> SR.
Notation "' x" := (srh x) (at level 8). (*general*)
Notation "> x" := (hum_sr x) (at level 8). (*humans (coerced to general)*)
Notation "'>v' x" := (veh_sr x) (at level 8). (*vehicles (coerced)*)
```

Humans have a distinct type of SRs, and the general restrictions are obtained by parametrizing on this type (the circumstance that general restrictions are limited to those of humans is coincidental). Due to the coercion (`:>`), `SRH` is a subtype of `SR`, while a term of `SR` is not a term of `SRH`. The notations are added for convenience. Vehicles are dealt with likewise. Now we may declare

```
Definition ADJ {x y z w u} := gs x y z w u -> gs x y z w u.
Parameter red: forall {x y z w}, @ADJ x Phy y z w.
Parameter ill: forall {x y z w}, @ADJ x Bio y z w.
```

and get the desired result: `ill` takes and returns stem phrases of biological and `red` of physical entities[18].

We have also implemented an on/off switch for SRs, enabling us to respect and ignore them with minimal changes to the code (commenting out an import of one of the two modules specifying differentiated and uniform SRs, respectively). The modules themselves are implemented differently in Supplements I and II; in the latter, the implementation is simpler, lacking some of the features of Supplement I[19]. When selectional restrictions are switched off, the code is checked only for morphosyntactic well-typedness.

4.3 Sentences

There are at least two ways to define sentences, taken in Supplements I and II, respectively. Since the latter is more recent, shorter, and seemingly also more parsimonious, we will review the latter approach only. After the base types and selectional restrictions have been defined (in sections 4.1-4.2), we give the following minimal (and almost standalone) fragment for exposition purposes:

```
Structure CPX := cpx {gp: SR -> CA -> NUM -> ST -> Type}.
Canonical Structure cp_p := cpx PLU.
Canonical Structure cp_x := cpx XP.
```

[18] Arguments in braces are implicit (i.e. implicitly applied), and `@ADJ` makes all arguments of `ADJ` explicit.

[19] If SRs are switched on, then e.g. `red (limbed man)` does not type check, since the phrase `limbed man` has restriction `Lim` while `red` expects `Phy`. To circumvent the limitation, Supplement I employs a special notation `[...]`. In particular, `red [limbed ...]` type checks there if SRs are off; if SRs are on, e.g. `red [limbed john]` is well- and `red [limbed book]` ill-typed. A more elegant, notationless solution is being developed for Supplement II.

```
Parameter hut: STM 'Phy (acc acc_n) SG N.
Parameter car: forall {x:SRV}, STM x (acc acc_n) SG N.
Parameter man: forall {x:SRH}, STM x (acc acc_n) SG N.
Parameter a: forall {x y z w}, gs x y z SG w -> gp cp_x y z SG w.
Parameter he: forall {x:SRH}, XP x NOM SG (Pr M1).
Parameter him: forall {x:SRH}, XP x ACC SG (Pr M1).

Parameter TAM: forall {x:Type}, (NF -> x) -> x.  (*tense-aspect-mood*)
Structure ANI := ani0 {ani:SR}.  (*animate entities*)
Canonical Structure ani_a := ani0 'Ani. (*default*)
Canonical Structure ani_b {x} := ani0 > x.  (*human*)
Canonical Structure ani_c {x} := ani0 >v x.  (*vehicle*)
Parameter hit: forall {w u k d m n f}, NF ->
gp cp_x (ani f) NOM w u -> gp cp_x k (acc d) m n -> S.

Check TAM hit (a man) (a hut). (*tests*)
Fail Check TAM hit (a hut) (a man). (*wrong selectional restriction*)
Fail Check TAM hit man (a man). (*"man" is not XP*)
Check TAM hit (a man) (a man).
Fail Check TAM hit (a man) he. (*wrong case*)
Check TAM hit (a man) him.
Check TAM hit he him.
Fail Check TAM hit him him. (*wrong case*)
Fail Check TAM hit (red he) him. (*"he" is not STM*)
Fail Check TAM hit (red man) him. (*"red man" is not XP*)
Check TAM hit (a (red man)) him.
Check TAM hit (a car) him.
```

We define the canonical structure `CPX` for accessing the elements of the set `{PLU, XP}` via notation overloading, and declare the variables `a`, `he`, etc. The variables correspond to morphemes, and all the argument variables (`he`, `man`, etc.) have lump types, implemented as function applications. `acc _` reduces to accusative or false accusative, the latter being defined as nominative. We need this for words such as `hut`, `man` and `car`, where nominative and accusative are indistinguishable. The definition of false accusative is omitted for space considerations.

To use a relation like *hit* on its arguments, it must be finitized with a tense-aspect-mood marker (e.g. `TAM` or `PAST`). A convenient way to do this is to declare an infinitive marker `NF` and prepend `NF →  ` to the relation's function type. As `NF` is an empty type, a function of type `NF → _` behaves like an infinitive, i.e. cannot be applied to anything. However, we can declare a function of type `∀ {x:Type}, (NF → x) → x`, and apply this to the infinitive to make it finite.

We are using the notation overloading trick again for `hit`, for it to accept various kinds of animate entities (in the fragment, humans and vehicles, even if they happen to have e.g. selectional restrictions `> Inf` and `>v Phy_v`).

4.4 Appendices and tests

For space considerations, longer (but still interesting) examples are at the end of the paper in the Appendices (the full code is in the online Supplements). Below is a short overview of the contents of the Appendices.

The examples we have seen are not truth-functional; however, sometimes a truth-functional semantics may be preferred. Appendix A introduces an optional truth-functionality module from Supplement I. Appendix B illustrates some ways of defining lump types (namely, as application, record, Cartesian product and Π-types). Appendix C shows some tests from the Supplements. A handful of the most revealing ones are below:

```
Check PRES know (several (PL man)) (a (few (PL man))).
(* several men know a few men *)
(* "john threw madly blue stones at the hut
and red limbed boys" has 2 parses: *)
Check madly (PAST throw) john (blue (-s stone))
(at (and (the hut) (red [limbed (-s boy)]))): S.
Check PAST throw john (madly blue (-s stone))
(at (and (the hut) (red [limbed (-s boy)]))): S.
(* Supplement I uses notation "[...]" to make
e.g. a limbed entity into a physical one *)

Check PRES throw (all (the (-s boy))) (every ball) (to him): S.
(* "all the boys throw every ball to him" is a sentence *)
Fail Check PAST throw john (all (-s stone)) (at he).
(* "john threw all stones at he": wrong case *)
Fail Check PRES throw (a walk) john. (* "a walk throws john": wrong SR *)
Check PRES throw john (-s stone) (at (all (madly (madly red)
(red (-s hut))))): S.
(* "john throws stones at all madly, madly red, red huts" is a sentence *)
Check PRES throw john (-s stone) (at (all (madly (madly red)
(red [limbed [-s hut]])))): S.
(* "john throws stones at all madly, madly red, red, limbed huts"
checks only if SRs are off *)
Check all ((and madly madly) red
(red (red [and blue limbed [-s john]]))): QU2 _ _ _ _.
(* "all madly and madly red, red, red, blue
and limbed johns" is a quantifier phrase *)
```

5 Related work

The tradition of using type theory for modeling NL started with Lambek calculus [16] in the pre-existing tradition of categorial grammar, invented by Ajdukiewicz in 1935 and developed (unbeknowst to Lambek) in the beginning of 1950s by Bar-Hillel [23]. The categorial tradition of NL modeling that followed [16] is rich but discontinuous (see [23] for an overview). Combinatory Categorial Grammar

(CCG) [27] and multimodal type-logical grammars [21] are some representatives of the tradition, one of the later additions to which is the Grail parser/automated theorem prover for multimodal categorial grammars [22].

It is convenient to compare all these (and other) categorial formalisms (CFs) to NLC together. A major similarity is that both NLC and the CFs model syntax and semantics together, and since both are type-driven, syntax may be even dismissed as a separate level of description[20]. Another major similarity is that both model NL compositionality with functions. There are more similarities, but with this the major ones seem to end.

A few major differences are as follows. The formalisms are categorial, NLC not. The CFs have diverse (and sometimes quite complex) underpinnings, while NLC is just a minimalist type system, i.e. very simple in comparison. The third difference is that NLC's types are syntactic, morphological and/or semantic, while CFs have only syntactic (and some, e.g. ACG [12] and its generalization AACG [15], also semantic) types. The fourth is that CFs allow to model constituent order, NLC not. The fifth is that (some?) CFs (e.g. CCG) can yield multiple formulas for a single NL expression independently of interpretation ambiguity, which is not the case in NLC.

Besides the categorial tradition, there are other typed approaches to NL to consider. The historical starting point for them is Montague Grammar of the 1970s, e.g. [20] has the primitive types of truth-values and entities, and diverse function types. More relevant for NLC — and especially its implementation — is the work that has followed in a richly typed setting, i.e. using dependent and/or polymorphic types. Methodologically, the closest approaches to NLC are Grammatical Framework (GF) [26] and a series of typed formalisations on different levels of abstraction [7,18,3,17], some of them implemented in Coq [4,5,18].

We start with GF, which is richly typed, uses record types and models NL, so is somewhat similar to NLC. However, NLC's difference from GF is profound. GF is a *formal language for writing NL grammars*, while NLC is a *type system for modeling semantic, syntactic and morphological compositionality* of NL. In particular, the "grammar" in the name of GF implies syntax and morphology but not semantics. It is only recently that parts of Abstract Meaning Representation and FrameNet libraries for GF have been implemented[13,14]. And the use of record types in GF, although pervasive, is very different from their possible use in NLC and does not follow LT-Intro (see section 3.1).

The rest of the above-cited works model only semantics [1,4,17,2] or semantics with some syntax [3,5], and while some of them use record types [7,18,4,3], and some even (implicit) lump types [4,3] (cf. section 3.1), none does so in the way suggested by NLC. Another novelty, both in this group and for Coq, is the scope of NLC's implementation, which is so far the broadest (both in terms of the number of linguistic categories and levels of description involved).

[20] E.g. "syntactic structure is merely the characterization of the process of constructing a logical form, rather than a representational level..." [27], p. xi.

In contrast to all other formalisms, NLC integrates syntactic, morphological, and compositional semantic information in a single level of type. For example, a modern typed formalism like AACG [15] requires (besides using Categorial Grammar and category theory in addition to type theory) simply-typed lambda-calculus, two separately typed syntaxes (abstract and concrete), typed semantics, syntax-semantics and abstract-concrete syntax interfaces, and does not even cover compositional semantics, i.e. selectional restrictions. An advantage of (A)ACG [15,12], HPSG [24], GF [26] and many other formalisms is that they support language-specific constituent orders; in addition, ACG and AACG have truth-functional semantics. The first can be implemented with a language-specific function from formulas to strings, the second has been implemented (with an Ltac tactic, which is only one possibility for this) in Appendix A.

6 Discussion

We presented a new formalism, a type system NLC, as well as its implementation on a structurally diverse fragment of English. As compared to the usual three-level representation of compositional semantics, syntax and morphology, NLC reduces NL compositionality to a single level by using lump types and types for morphemes. This main novelty of the formalism is captured in rules AT-Intro and LT-Intro.

The implementation was done in Coq and, in hindsight, the choice of the programming language seems somewhat accidental. Quite possibly, it would have been easier to do this in some other language. The prime suspects are languages with type systems of roughly the same level of complexity, e.g. OCaml, Agda and Haskell. However, perhaps NLC could be also encoded in a qualitatively simpler type system. As we can have e.g. intersection types without dependent and polymorphic types, it is clear that the latter are not required for lump types. Also, subtyping polymorphism is attainable in several languages without dependent and polymorphic types. However, without dependent or polymorphic types, getting useful function types will be more tricky (if not altogether impossible).

As a representative fragment of NL morphology and syntax has not been directly formalized in a general-purpose programming language before[21], at the time of starting the project (in 2016), I was unaware of both the goal's feasibility and the eventual form it might take. Generally speaking, choosing a proof assistant for such work makes sense — if it is (theoretically) possible to formalize all mathematics in it, it must be also possible to formalize a substantial fragment of NL (provided there is a mathematical representation of it). There were some intuitions and ideas about the suitable representation to start with but, in the end, Coq "corrected" them profoundly. From the very beginning, as a methodological guideline, there was the idea to produce code that would not only represent but *look like* NL. The motivation behind this was an elementary

[21] It has been directly formalized in a specialized programming language (GF), and indirectly in a general-purpose programming language (Haskell, in which GF is written).

theory of NL expression as function (or perhaps relation) application. In the end, to get that kind of code, my problems mostly reduced to getting Coq's type inference to work. Coq has several (and sometimes quite involved) devices for this — custom notations, Ltac, type classes, canonical structures and more — but it may well be that the implementation-specific struggle would have been alleviated in some other language. On the other hand, there is no dedicated language for this kind of work[22] and Coq, even if an overkill, certainly qualifies as something that should get the work done. However, after giving all credit to Coq's complexity and relatively advanced type inference, I encourage the interested reader to experiment along these lines not only in Coq but a programming language of their choice.

References

1. Asher, N.: Selectional restrictions, types and categories. Journal of Applied Logic **12**(1), 75–87 (2014). https://doi.org/10.1016/j.jal.2013.08.002
2. Bekki, D., Asher, N.: Logical polysemy and subtyping. In: Motomura, Y., Butler, A., Bekki, D. (eds.) New Frontiers in Artificial Intelligence, pp. 17–24. Springer, Berlin, Heidelberg (2013)
3. Chatzikyriakidis, S., Luo, Z.: Natural language inference in Coq. Journal of Logic, Language and Information **23**(4), 441–480 (Dec 2014). https://doi.org/10.1007/s10849-014-9208-x
4. Chatzikyriakidis, S., Luo, Z.: Individuation criteria, dot-types and copredication: A view from modern type theories. In: Proceedings of the 14th Meeting on the Mathematics of Language (MoL 2015). pp. 39–50. Association for Computational Linguistics (2015). https://doi.org/10.3115/v1/W15-2304, `http://www.aclweb.org/anthology/W15-2304`
5. Chatzikyriakidis, S., Luo, Z.: Proof assistants for natural language semantics. In: Amblard, M., de Groote, P., Pogodalla, S., Retoré, C. (eds.) Logical Aspects of Computational Linguistics. Celebrating 20 Years of LACL (1996–2016). pp. 85–98. Springer, Berlin, Heidelberg (2016), `http://www.cs.rhul.ac.uk/~zhaohui/LACL16PA.pdf`
6. Constable, R.L.: Recent results in type theory and their relationship to Automath. In: Kamareddine, F.D. (ed.) Thirty Five Years of Automating Mathematics, pp. 37–48. Springer Netherlands, Dordrecht (2003)
7. Cooper, R.: Records and record types in semantic theory. Journal of Logic and Computation **15**(2), 99–112 (2005). https://doi.org/10.1093/logcom/exi004
8. Davidson, D.: Truth and meaning. Synthese **17**(3), 304–323 (1967), `http://www.jstor.org/stable/20114563`
9. Di Sciullo, A.M. (ed.): Asymmetry in Grammar: Volume 1: Syntax and semantics. John Benjamins (2003), `https://www.jbe-platform.com/content/books/9789027296801`
10. Di Sciullo, A.M. (ed.): Asymmetry in Grammar: Volume 2: Morphology, phonology, acquisition. John Benjamins (2003), `https://www.jbe-platform.com/content/books/9789027296795`

[22] GF does not count, as it does not allow to choose between mathematical formalisms.

11. Dryer, M.S.: Order of Subject, Object and Verb. In: Dryer, M.S., Haspelmath, M. (eds.) The World Atlas of Language Structures Online. Max Planck Institute for Evolutionary Anthropology, Leipzig (2013), http://wals.info/chapter/81
12. de Groote, P.: Towards abstract categorial grammars. In: Proceedings of 39th Annual Meeting of the Association for Computational Linguistics. pp. 252–259. Association for Computational Linguistics, Toulouse, France (Jul 2001). https://doi.org/10.3115/1073012.1073045, https://www.aclweb.org/anthology/P01-1033
13. Gruzitis, N., Dannélls, D.: A multilingual FrameNet-based grammar and lexicon for controlled natural language. Lang Resources & Evaluation **51**(1), 37–66 (2017). https://doi.org/10.1007/s10579-015-9321-8
14. Gruzitis, N.: Abstract meaning representation (amr) (2019), https://github.com/GrammaticalFramework/gf-contrib/tree/master/AMR
15. Kiselyov, O.: Applicative abstract categorial grammar. In: Kanazawa, M., Moss, L.S., de Paiva, V. (eds.) NLCS'15. Third Workshop on Natural Language and Computer Science. EPiC Series in Computing, vol. 32, pp. 29–38. EasyChair (2015). https://doi.org/10.29007/s2m4, https://easychair.org/publications/paper/RPN
16. Lambek, J.: The mathematics of sentence structure. The American Mathematical Monthly **65**(3), 154–170 (1958)
17. Luo, Z.: Type-theoretical semantics with coercive subtyping. In: Semantics and Linguistic Theory. vol. 20, pp. 38–56. Vancouver (2010)
18. Luo, Z.: Contextual analysis of word meanings in type-theoretical semantics. In: Pogodalla, S., Prost, J.P. (eds.) Logical Aspects of Computational Linguistics. pp. 159–174. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
19. Luuk, E.: Nouns, verbs and flexibles: implications for typologies of word classes. Language Sciences **32**(3), 349–365 (2010). https://doi.org/10.1016/j.langsci.2009.02.001
20. Montague, R.: The proper treatment of quantification in ordinary English. In: Portner, P., Partee, B.H. (eds.) Formal Semantics: The Essential Readings, pp. 17–34. Blackwell, Oxford (2002)
21. Moortgat, M.: Categorial type logics. In: van Benthem, J., ter Meulen, A. (eds.) Logic and Language, pp. 95–179. Elsevier, Amsterdam, North-Holland (2011)
22. Moot, R.: The Grail Theorem Prover: Type Theory for Syntax and Semantics. In: Luo, Z., Chatzikyriakidis, S. (eds.) Modern Perspectives in Type-Theoretical Semantics, Part III, vol. Studies in Linguistics and Philosophy, pp. 247–277. Springer (2017). https://doi.org/10.1007/978-3-319-50422-3_10, https://hal-lirmm.ccsd.cnrs.fr/lirmm-01471644
23. Morrill, G.: A chronicle of type logical grammar: 1935–1994. Research on Language and Computation **5**, 359–386 (09 2007). https://doi.org/10.1007/s11168-007-9034-2
24. Pollard, C., Sag, I.A.: Head-Driven Phrase Structure Grammar. University of Chicago Press, Chicago (1994)
25. Pustejovsky, J.: The Generative Lexicon. MIT Press, Cambridge, MA. (1995)
26. Ranta, A.: Grammatical Framework: a type-theoretical grammar formalism. The Journal of Functional Programming **14**(2), 145–189 (2004). https://doi.org/10.1017/S0956796803004738
27. Steedman, M.: The Syntactic Process. MIT Press, Cambridge, MA, USA (2000)

A Appendix

A.1 Truth-functionality

The fragment we have introduced so far is not truth-functional, i.e. we cannot compute truth values on it. In a sense, this does not matter, as we clearly do not need truth-functionality for compositional semantics, syntax and morphology. However, truth-functionality is certainly something that comes in handy in natural language inference (not to mention model-theoretic semantics, of what it is the foundation). For the latter reason, truth-functionality is a quality of NL formalisms that philosophers of language and students of Montague, in particular, have always insisted on. Fortunately, it is relatively easy to turn our fragment truth-functional.

The following is an optional extension of (our implementation of) NLC. For degenerate models (in which all sentences are uniformly true, false or undecidable) one only needs to add a coercion such as

```
Parameter s_prop:> S -> Prop. (* all S-s undecidable *)
```

For nontrivial models, a convenient solution is to define an Ltac match, e.g.

```
Parameter s_prop:> S -> Prop. (*coerce S to Prop*)
Notation ":>> x" := (s_prop (s1s2 (s0s1 x))) (at level 179).
Notation "$ x" := ltac:(let u := constr:(x:Prop) in
lazymatch u with
  | :>> (PAST _ _) => exact True
  | s_prop (s1s2 (however (PAST _ _))) => exact True
  | :>> (PRES sleep john) => exact True
  | :>> (variad_S0 _ _ _ (PRES walk (-s boy))) => exact True
  | :>> (PRES _ _) => exact False
  | :>> (variad_S0 _ _ _ (PRES _ _)) => exact False
  | :>> (and (PAST _ _) (PAST _ _)) => exact True (*sleep..*)
  | :>> (and (PAST _ _) (variad_S0 _ _ _ (PAST _ _)))  => exact True (*walk..*)
  | :>> (and (PAST _ _) (PRES sleep john)) => exact True
  | :>> (and (PAST _ _) (variad_S0 _ _ _ (PRES walk (-s boy))))  => exact True
  | :>> (and ?m ?n) => idtac "unanalyzed:" m n
  | s_prop (s1s2 (however ?n)) => idtac "unanalyzed:" n
  | x => idtac "unanalyzed:" x": Prop"
  (*leave some potentially t-functional constructs unanalyzed*)
  end) (at level 199).
```

to get the axioms we want[23], and then proceed by theorem proving based on them:

```
(*a trivial proof that "boys don't sleep" and "john sleeps"*)
Theorem pres: (~ ($ (PRES sleep (-s boy)))) /\ ($ (PRES sleep john)).
Proof. firstorder. Qed.
```

[23] Notice that we use the `let` construct to get `x:Prop`.

B Appendix

```
(* Compound types for NL modeling: application, record, product and pi-types; written and
best viewed in Coq 8.9 --- https://gitlab.com/jaam00/nlc/blob/master/compound.v *)

Module Types. (*shared types*)

Inductive ST := F | N | PN. (*stem: flexible, noun, proper name*)
Inductive NUM := SG | PLR. (*number: singular, plural*)
Inductive SR := Phy | Sen | Inf | Lim. (*selectional restriction: physical, sentient..*)
Inductive CA := NOM | GEN | LAT | DIR | LOC. (*case/adposition: nominative, genitive..*)

End Types.

Module Application. (*----------------------*)
Export Types.

Parameter STM PLU XP: SR -> CA -> NUM -> ST -> Type.
Parameter hut: STM Phy NOM SG N.
Parameter thought: STM Inf NOM SG N.

Structure CSTM := cx {g: SR -> CA -> NUM -> ST -> Type}.
Canonical Structure cx_x: CSTM := cx STM.
Canonical Structure cx_x': CSTM := cx PLU.

Parameter red: forall {x y z w}, g x Phy y z w -> g x Phy y z w.
Parameter a: forall {x y z w}, g x y z SG w -> XP y z SG w.
Parameter the: forall {x y z u w}, g x y z u w -> XP y z u w.
Parameter PL: forall {x y z w}, g x y z SG w -> PLU y z PLR w.

End Application.

Module Record. (*----------------------*)
Export Types.

Inductive CAT := STM | PLU | XP. (*stem, plural, XP*)
Record POS := mkPOS { cw:CAT; cw0:SR;
cw1:CA; cw2:NUM; cw3:ST }.
Parameter co:> POS -> Set.
Parameter hut: mkPOS STM Phy NOM SG N.
Parameter thought: mkPOS STM Inf NOM SG N.

Structure CSTM := cx {g: CAT}.
Canonical Structure cx_x: CSTM := cx STM.
Canonical Structure cx_x': CSTM := cx PLU.

Parameter red: forall {x y z w}, mkPOS (g x) Phy y z w -> mkPOS (g x) Phy y z w.
Parameter a: forall {x y z w}, mkPOS (g x) y z SG w -> mkPOS XP y z SG w.
Parameter the: forall {x y z u w}, mkPOS (g x) y z u w -> mkPOS XP y z u w.
```

```
Parameter PL: forall {x y z w}, mkPOS (g x) y z SG w -> mkPOS PLU y z PLR w.

End Record.

Module Product. (*----------------------*)
Export Types.

Inductive CAT := STM | PLU | XP.
Open Scope type.
Definition POS := CAT*SR*CA*NUM*ST.
Parameter co:> POS -> Set.
Definition pos x y z w u := (x, y, z, w, u): POS.
Parameter hut: pos STM Phy NOM SG N.
Parameter thought: pos STM Inf NOM SG N.

Structure CSTM := cx {g: CAT}.
Canonical Structure cx_x: CSTM := cx STM.
Canonical Structure cx_x': CSTM := cx PLU.

Parameter red: forall {x y z w}, pos (g x) Phy y z w -> pos (g x) Phy y z w.
Parameter a: forall {x y z w}, pos (g x) y z SG w -> pos XP y z SG w.
Parameter the: forall {x y z u w}, pos (g x) y z u w -> pos XP y z u w.
Parameter PL: forall {x y z w}, pos (g x) y z SG w -> pos PLU y z PLR w.

End Product.

Module Pi. (*----------------------*)
Export Types.

Inductive CAT := STM | PLU | XP.
Parameter POS: forall (cw:CAT)(cw0:SR)
(cw1:CA)(cw2:NUM)(cw3:ST), Type.
Parameter hut: POS STM Phy NOM SG N.
Parameter thought: POS STM Inf NOM SG N.

Structure CSTM := cx {g: CAT}.
Canonical Structure cx_x: CSTM := cx STM.
Canonical Structure cx_x': CSTM := cx PLU.

Parameter red: forall {x y z w}, POS (g x) Phy y z w -> POS (g x) Phy y z w.
Parameter a: forall {x y z w}, POS (g x) y z SG w -> POS XP y z SG w.
Parameter the: forall {x y z u w}, POS (g x) y z u w -> POS XP y z u w.
Parameter PL: forall {x y z w}, POS (g x) y z SG w -> POS PLU y z PLR w.

End Pi.

Module Test. (*comment out imports as required*)
Import (* Application *) (* Record *) Product (* Pi *).

Check red hut.
```

```
Check a thought.
Fail Check red thought.
Check a (red hut).
Check the (red hut).
Check the (red (PL hut)).
Fail Check red (a hut).
Fail Check a (red (PL hut)).
Fail Check a (the (red hut)).
Fail Check the a.

End Test.
```

C Appendix

```
(* The following examples are generously commented and should be self-explanatory *)
(*from https://gitlab.com/jaam00/nlc/blob/master/frag.v*)

Check PAST throw john. (*"John threw" type checks..*)
Fail Check PAST throw john: S. (*..but not as a sentence.*)
Check PAST throw john (-s stone): S.
Check PAST throw john (-s stone) (at (the hut)): S.
(*"At the hut" can be the 3rd argument of "throw" (above),
but not the 2nd or 1st one:*)
Fail Check PAST throw john (at (the hut)).
Fail Check PAST throw (a hut).
Check PAST throw [a hut]. (*..except when in brackets (works only if SRs are off).*)

Fail Check PAST throw john (-s stone) (in (the hut)).
(*"In the hut" cannot be an argument of "throw"..*)
Check in (the hut) (PAST throw john (-s stone)): S.
(*..but it can be a sentence modifier..*)
Check at (every hut) (PAST throw john (-s stone)): S.
(*..and so can "at every hut"..*)
Check however (PAST throw john (-s stone)): S.
(*..and sentential adverbs like "however".*)
Fail Check and (PAST throw john (a stone)) john.
(*Connectives cannot range over a sentential
and nominal argument, but can range over nominal..*)
Check and (every john) (all (the (-s boy))).
Check and (PAST walk (-s boy) (to (all (the (-s hut)))))
(PAST sleep john). (*..or sentential arguments..*)
Check and (PAST walk john) (PAST sleep john).
(*..(also w/ optional arguments omitted).*)
Fail Check in (all (the hut)). (*ungrammatical?*)
Check in (the (entire hut)). (*grammatical*)
Check madly ((in (the (entire hut))) ((PAST walk) (all (the (-s boy))))): S.
(*"All the boys walked madly in the entire hut" is a sentence*)

(*Examples of compound types:*)
```

```
Check john: XP0 (hu H_Phy) NOM SG (Pn Ml).
(*"John" is an XP, human, proper name, male,
in nominative, singular, a physical entity..*)
Check john: XP0 (hu H_Lim) NOM SG PN+. (*..and a limbed entity.*)
Check john: XP0 Lim NOM SG (Pn Ml). (*The normalized version of the above.*)
Check the john: XP2 _ NOM SG (Pn Ml). (*"The John" is another type of XP.*)
Check and (a (good throw)) (entire (-s throw)).
(*"Throw" is a flexible (a function or argument (as above))..*)
Check and (PRES stone john (a boy)) (PAST throw john (-s stone)
((at (-s stone)))). (*..and "stone" likewise.*)
(*So "at stones" is an XP, flexible in lative,
plural and a physical entity:*)
Check at (-s stone): XP2 Phy LAT PLR F.
(*As prescribed by "red", "red John" is a physical entity:*)
Check red john: XP0 Phy NOM SG (Pn Ml).
Check [red john]: XP0 Lim NOM SG PN+.
(*..but we can make it into a limbed one w/ our bracket notation (above).
"PN+" means proper name w/ gender info*)
Check hut: STM Phy _ _ _.
(*"Hut" is a stem and always a physical entity..*)
Check [hut]: STM Lim _ _ _.
(*..so we can make it into a limbed one only if SRs are off
(the above line fails if SRs are on).*)
Check and (the (entire hut)) (all (-s john)): XP2 Phy NOM _ _.
(*"The entire hut and all Johns" is an XP
and physical entity in nominative..*)
Check and (the (entire hut)) (all (-s john)): XP2 Phy ACC' _ _.
(*..or pseudo-accusative (by zero-derivation)..*)
Check [and (the (entire hut)) (all (-s john))]: XP2 Sen ACC' _ _.
(*..which can be made into a sentient entity if SRs are off.*)

(*Arguments of verbs and flexibles must be proper XPs:*)
Fail Check PAST throw (the boy) (entire stone) (to (him)).
Check PAST throw (the boy) (the (entire stone)) (to (him)): S.
(*Arguments of V and F must have correct
cases and selectional restrictions:*)
Check PAST throw he (all (-s stone)) (at (john)): S.
Check PAST throw john (all (the (-s stone))) (at (him)): S.
Fail Check PAST throw him (all (-s stone)) (at john). (*wrong case*)
Fail Check PRES throw john (a walk). (*wrong case and SR*)

(*from https://gitlab.com/jaam00/nlc/blob/master/cop.v*)

Fail Check are (PL man) (PL hut). (*"men are huts" won't do*)
Fail Check is i me. (*fails as desired (because gp.. doesn't match XP..)*)
Fail Check PRES know i (who (is ill ill)). (*"i know ill who is ill" won't do*)
Check TAM know i (who (are ill (PL man))).
(*"i know men who are ill" (TAM is tense-aspect-mood marker)*)
Check PRES know i (who (PRES COP ill (the man))). (*i know the man who is ill*)
Check TAM know i (who (are ill (the (PL man)))). (*i know the men who are ill*)
```

```
Fail Check TAM know i (who (are ill (a (PL man)))).
(*"i know a men who are ill" won't do*)
Check PAST know i (who (TAM COP ill (a man))). (*i knew a man who is ill*)
Check TAM know i (who (PAST COP ill (the man))). (*i know the man who is ill*)

Fail Check PRES PRES know i me.
Fail Check PAST (PAST know) i me.
Check PAST know i (the hut).
Check TAM know (a man) (the hut). (*a man knows the hut*)

Fail Check PRES know i the.
Fail Check PRES know i i.
Fail Check know i me. (*TAM required, know is nonfinite*)
Check PAST know i me.
Check PRES know i me.
Check TAM know i (who (is (the man) me)). (*i know the man who is me*)
Fail Check PRES know i (who (is (the man) i)).  (*wrong case*)
```