30th International Symposium on
Logic-based Program Synthesis and Transformation

LOPSTR 2020

7-9 September 2020

Preliminary Proceedings

Edited by Maribel Fernández.

LOPSTR 2020 is part of BOPL
Bologna Federated Conference on Programming Languages
Preface

This volume contains the papers presented at LOPSTR 2020 (30th International Symposium on Logic-Based Program Synthesis and Transformation), held 7-9 September 2020.

The aim of the LOPSTR series is to stimulate and promote international research and collaboration on logic-based program development. Topics of interest cover all aspects of logic-based program development (including in domain-specific languages), all stages of the software life-cycle, and issues of both programming-in-the-small and programming-in-the-large, including: synthesis; transformation; specialisation; composition; optimisation; specification; analysis and verification; testing and certification; program and model manipulation; inversion; machine learning for program development; transformational techniques in SE; applications and tools.

This year LOPSTR is part of the Bologna Federated Conference in Programming languages (together with PPDP, WFLP and Microservices), which has been organised as a virtual conference. Previous editions of LOPSTR were held in Porto, Namur, Edinburgh, Siena, Canterbury, Madrid, Leuven, Odense, Hagenberg, Coimbra, Valencia, Lyngby, Venice, London, Verona, Uppsala, Madrid, Paphos, London, Venice, Manchester, Leuven, Stockholm, Arnhem, Pisa, Louvain-la-Neuve, Manchester and Frankfurt.

The Programme Committee consists of:

– Elvira Albert, Complutense University of Madrid, Spain
– María Alpuente, Universitat Politècnica de València, Spain
– Mauricio Ayala-Rincón, University of Brasilia, Brazil
– Clara Bertolissi, University Aix-Marseille, France
– Emanuele De Angelis, CNR Inst. for Systems Analysis and Computer Science, Italy
– Maribel Fernández, King’s College London, UK (chair)
– Mario Florido, University of Porto, Portugal
– Maurizio Gabbrielli, University of Bologna, Italy
– Robert Glück, University of Copenhagen, Denmark
– Gopal Gupta, University of Texas at Dallas, US
– Michael Hanus, Kiel University, Germany
– Delia Kesner, Université de Paris, France
– Andy King, University of Kent, UK
– Temur Kutsia, RISC J. Kepler University of Linz, Austria
Seventeen papers were selected for presentation at LOPSTR 2020. In addition the programme includes invited talks by Philipp Rümmer (Uppsala University, Sweden), Ekaterina Komendantskaya (Heriot-Watt University, UK) and José Meseguer (University of Illinois at Urbana-Champaign, US).

The Final Proceedings of LOPSTR 2020 will appear in LNCS (Lecture Notes in Computer Science), published by Springer.

More information about the conference can be found on the conference web page:

https://nms.kcl.ac.uk/maribel.fernandez/LOPSTR2020/

We would like to thank all those who contributed to LOPSTR 2020, particularly the invited speakers, authors, programme committee members and local organisers. We are grateful to Springer for sponsoring the best paper awards and to EasyChair for providing support to deal with the reviewing process.
Runtime Complexity Analysis of Logically Constrained Rewriting

Sarah Winkler and Georg Moser

University of Verona, Italy, and University of Innsbruck, Austria

Abstract. Logically constrained rewrite systems (LCTRSs) are a versatile and efficient model of rewriting that can be used to model programs from various programming paradigms, as well as simplification systems in compilers and SMT solvers. In this paper, we investigate techniques to analyse the worst-case runtime complexity of LCTRSs. For that, we exploit synergies between previously developed decomposition techniques for standard term rewriting by Avanzini et al.
in conjunction with alternating time and size bound approximations for integer programs by Brockschmidt et al. and adapt these techniques suitably to LCTRSs. Furthermore, we provide novel modularization techniques to exploit loop bounds from recurrence equations which yield sublinear bounds. We have implemented the method in TCT to test the viability of our method.

1 Introduction

Rewriting with constraints over background theories is a highly versatile model of computation and tool for analysis. While user-defined data types are modelled by free function symbols, arbitrary decidable theories can be incorporated, such as integer or bit-vector arithmetic, lists, or array theory. Constraints over these theories can be effectively handled by SMT solvers. Different rewrite formalisms capture this idea. Here we use the recent notion of logically constrained term rewrite systems (LCTRSs for short), due to Kop et al.

LCTRSs can abstract programs in a variety of paradigms, comprising imperative, functional, and logic languages. They also subsume integer transition systems (ITSs), which constitute a frequently used program abstraction, but do—in contrast to LCTRSs—not support (non-tail) recursion. On the other hand, LCTRSs can also model simplification routines for expressions, which are crucial procedures in compilers or SMT solvers. For all of these application areas, LCTRSs offer a uniform toolset to analyse termination (or non-termination), reachability, uniqueness, or program equivalence.

However, techniques for resource analysis of LCTRSs are so far lacking. This is despite the fact that in their application domains (program analysis, simplification systems), execution time is crucial. As a remedy, this paper investigates methods to analyse (worst-case) innermost runtime complexity of logically constrained rewrite systems. To this end, we unify and generalise the complexity framework for standard rewriting by Avanzini and Moser with the approach.
Contributions. We present a novel resource analysis framework for logically constrained rewrite systems (Sect. 4) coached in the modular processor framework of TCT [7]. Precisely,

1. we present the first fully-automated runtime complexity analysis of LTCRSs;
2. we unify the complexity framework for standard (innermost) rewriting by Avanzini and Moser [6] and the alternating time and size bound approximations for ITSs by Brockschmidt et al. [9],
3. generalising this, we introduce a novel modularisation processor, the splitting processor;
4. we present a novel processor, dubbed loop processor to derive sublinear bounds based on recurrences as described by the Master Theorem;
5. we illustrate the viability of our method by providing a prototype implementation as a dedicated module tct-lctrs in TCT.

In the remainder of the section, we highlight potential application areas of LTCRSs to emphasise their versatility. In the next section (Sect. 2) we give a high-level account of our technical achievements, providing a step-by-step explanation how the runtime complexity of a natural representation of mergesort can be optimally analysed in our framework. In this section, we also discuss to what extent our results can be applied to the below given examples. In Sect. 3 we summarise the foundations of LTCRSs, while in Sect. 4 we detail the complexity framework used. The novel processors are introduced in Sect. 6 and implementational choices are detailed in Sect. 7. Finally, in Sect. 8 we conclude.

Logically Constrained Rewrite Systems. We emphasise motivational examples from three different domains, focusing on imperative and logic programs, as well as compiler optimisations.

Example 1. The following recursive ITS $R_1$, due to Albert et al. [1] represents an imperative mergesort implementation. The rewrite system is naturally coached into the LTCRS framework.

\begin{align*}
(1) \text{init}(x, y, z) & \rightarrow m(x, y, z) & (2) \quad m_3(x, y, z) & \rightarrow \text{merge}(y, z, z) \\
(3) \quad m_1(x, y, z) & \rightarrow m(y, y, z) & (4) \quad \text{merge}(x, y, z) & \rightarrow \text{merge}(x - 1, y, z) \ [x \geq 1 \land y \geq 1] \\
(5) \quad m_0(x, y, z) & \rightarrow \text{split}(x, y, z) & (6) \quad \text{split}(x, y, z) & \rightarrow \text{split}(x - 2, y, z) \ [x \geq 2] \\
(7) \quad m_2(x, y, z) & \rightarrow m(z, y, z) & (8) \quad \text{merge}(x, y, z) & \rightarrow \text{merge}(x, y - 1, z) \ [x \geq 1 \land y \geq 1] \\
(9) \quad m(x, y, z) & \rightarrow \langle m_0(x, u, v), m_1(x, u, v), m_2(x, u, v), m_3(x, u, v) \rangle \ [x \geq 2 \land u \geq 0 \land v \geq 0 \\
\qquad & \land x + 1 \geq u + 2v \land 2u + 2v \geq x \land x \geq 2v + 1 \geq x]
\end{align*}

Here a rule of the form $\ell \rightarrow r [c]$ means that an instance of $\ell$ is replaced by the respective instance of $r$ provided that the instance of $c$ is satisfied.

Similarly, (constraint) logic programs can be nicely suited to LTCRSs.
Example 2. Consider the following simple Prolog program from the benchmarks considered by Mesnard and Neumerkl in [31].

\[
\text{max\_length}(Ls,M,Len) :- \text{max1}(Ls,0,M), \text{len}(Ls,Len).
\]

\[
\begin{align*}
\text{len}([H|T],L) & :\rightarrow \text{len}(T,LT), L \leftarrow LT + 1. \\
\text{len}([],0). \\
\text{max1}([H|T],N,M) & :\rightarrow H \leq N, \text{max1}(T,N,M). \\
\text{max1}([],M,M). \\
\text{max1}([H|T],N,M) & :\rightarrow H > N, \text{max1}(T,H,M).
\end{align*}
\]

Assuming an instantiated list \( Ls \), \text{max\_length}/3 is deterministic and returns the maximal value of the list entries and the length of the list. This function becomes representable as the following LCTRS \( R_2 \).

\[
\text{max\_length}(ls,m,l) \rightarrow \langle \text{max}(ls,0,m), \text{len}(ls,l) \rangle
\]

\[
\begin{align*}
\text{len}(xs,l) & \rightarrow \text{len}(t,l-1) \ [xs \approx h : t] \\
\text{len}([],0) & \rightarrow \langle \rangle. \\
\text{max}(xs,n,m) & \rightarrow \text{max}(t,n,m) \ [h \leq n \land xs \approx h : t] \ 	ext{max}([],m,m) \rightarrow \langle \rangle. \\
\text{max}(xs,n,m) & \rightarrow \text{max}(t,h,m) \ [h > n \land xs \approx h : t]
\end{align*}
\]

Here, \( :: \) denotes the cons operator and \( \langle \cdot, \cdot \rangle, \langle \rangle \) are additional constructor symbols to collect the recursive calls of a rule. Conceptually LCTRSs appear as a good fit to express constraint logic programs as well, making use of the fact that constraints are natively supported.

It is perhaps important to emphasise, that LCTRSs are not confined to static program analysis. This is stressed by our final example, which is concerned with program optimisation.

Example 3. The Instcombine pass in the LLVM compilation suite performs peephole optimisations to simplify expressions in the intermediate representation. The current optimisation set contains over 1000 simplification rules to e.g. replace multiplications by shifts or perform bitwidth changes. About 500 of them have recently been translated into the domain-specific language Alive [30], and subsequently into LCTRSs [40], resulting in rules of the following shape:

\[
\begin{align*}
\text{add}(x,x) & \rightarrow \text{shift\_left}(x,\#x01) \\
\text{add}(\text{add}(\text{xor}(\text{or}(x,a),y),\#x01),w) & \rightarrow \text{sub}(w,\text{and}(x,b)) \ [a \approx \sim b] \\
\text{add}(\text{xor}(x,a),z) & \rightarrow \text{sub}(a+z,x) \ \text{is\_Power\_2}(a+\#x01) \land \ldots .
\end{align*}
\]

These rules are expressed over the background theory of bit-vectors. Naturally, as a compiler pass this simplification suite is a performance-critical routine, hence an automated complexity analysis is of great interest.

2 Step by Step to an Optimal Bound

Consider the rewrite system \( R_1 \) from Ex. [1] and a derivation starting with an instance of \text{init}(x_0,y_0,z_0). Below we sketch the steps to obtain an upper bound

\[1 \text{ See } \text{http://www.complang.tuwien.ac.at/cti/bench/}\]
on the runtime complexity of $R_1$, expressed in $|x_0|$, $|y_0|$, and $|z_0|$, where $|\cdot|$ is a suitable size measure like the absolute value.

An automated runtime complexity analysis of mergesort is notoriously difficult: For this example, PUBS [1] and CoFloCo [17,16] can only derive a quadratic bound, while KoAT [9] (as well as AProVE [22]) even proposes an exponential bound. Due to the work presented in this paper, our complexity analyser TCT can automatically prove the optimal $O(n \cdot \log(n))$ upper bound. This is obtained by the following recipe.

1. A dependency graph approximation is computed to estimate computation paths, where the numbers refer to (dependency tuples of) rules in Ex. 1:

![Dependency Graph]

2. Next, we derive bounds on the size of variables in left hand sides of rules, in terms of the sizes of (instances of) the variables in the initial term init$(x_0, y_0, z_0)$.

For example, it is easy to check that for rule (9), $|x|$, $|y|$, and $|z|$ are bounded by $|x_0|$, $|y_0|$, and $|z_0|$, respectively, and all variables in other rules are bounded by $|x_0|$. This is established by the size bounds processor (Lemma 3). Formally, we adapt techniques developed for ITSs for that purpose [9].

3. We first derive time bounds for the SCCs $\{2, 4, 8\}$ and $\{6\}$ separately (Lem. 2). Thus, re-using the size bounds from above, and suitable interpretations [6] (also called polynomial ranking functions [9]) for LCTRSs, one can derive linear runtime bounds $2|x_0| + 1$ and $|x_0|$ for these subproblems, respectively.

4. In order to analyse the SCC $\{3, 7, 9\}$, we first apply chaining to combine rule (9) with (3) and (7), respectively (eliminating symbols $m_1$ and $m_2$).

5. With respect to the modified rule (9) and the derived subproblem bounds, we exploit the loop processor (Lemma 5) to observe that its runtime can thus be overestimated by the following recurrence equations.

$$f(|x|, |y|, |z|) = 2 \cdot f(|x|/2, |x|/2, |x|/2) + 3|x| + 1 \quad f(1, |y|, |z|) = 0 \quad (1)$$

Solving the recurrences by the Master Theorem, implies an overall runtime complexity of $O(|x_0| \cdot \log(|x_0|))$ for $R_1$, as $|x|$ in rule (9) is bound by $|x_0|$.

Wrt. $R_2$ from Example 3 we can fully automatically infer an (asymptotic) optimal linear bound on the runtime complexity for the given instantiation. Here, we take an instance of $\text{max_length}(xs, z, l)$ as initial term. Note that the corresponding logic program cannot be handled by the dedicated variant of AProVE [22], which also fails to prove termination. Termination can only be shown by the most recent version of AProVE [22]. However, in general our approach is restricted to logic programs with instantiation patterns that ensure determinism and avoid failure. Finally, Ex. 3 cannot yet be handled, as a successful
analysis requires the extension of the proposed framework to (innermost) derivational complexity (i.e., the setting of arbitrary starting terms that may contain nested defined symbols). This is subject to future work. However, we conceive the work established in this paper as a solid first step towards the automated analysis of such systems.

3 Logically Constrained Term Rewriting

We assume familiarity with term rewriting \[386\], but briefly recapitulate the notion of logically constrained rewriting \[28,20\] that our approach is based on.

We consider an infinite, sorted set of variables \( \mathcal{V} \) and a sorted signature \( \mathcal{F} = \mathcal{F}_T \uplus \mathcal{F}_L \) such that \( \mathcal{T}(\mathcal{F}, \mathcal{V}) \) denotes the set of terms over this disjoint signature. Symbols in \( \mathcal{F}_T \) are called term symbols, while \( \mathcal{F}_L \) contains theory symbols. A term in \( \mathcal{T}(\mathcal{F}_L, \mathcal{V}) \) is a theory term. For a non-variable term \( t = f(t_1, \ldots, t_n) \), we write \( \text{root}(t) \) to obtain the top-most symbol \( f \). A position \( p \) is an integer sequence used to identify subterms. The subterm of \( t \) at position \( p \) is defined as \( t|_p = t \), and if \( t = f(t_1, \ldots, t_n) \) then \( t|_p = t_i|_p \). We write \( t \uplus u \) if \( u \) is a subterm of \( t \). The set of positions in a term \( t \) is denoted \( \text{Pos}(t) \), and for a set of function symbols \( \mathcal{F}' \subseteq \mathcal{F} \), \( \text{Pos}_{\mathcal{F}'}(t) \) are those positions \( p \in \text{Pos}(t) \) such that \( t_p \) is rooted by a symbol in \( \mathcal{F}' \). A substitution \( \sigma \) is a mapping from variables to terms with finite domain \( \text{Dom}(\sigma) \). We write \( t\sigma \) to denote the application of \( \sigma \) to a term \( t \).

Theory terms \( \mathcal{T}(\mathcal{F}_L, \mathcal{V}) \) have a fixed semantics: we assume a mapping \( \mathcal{I} \) that assigns to every sort \( i \) occurring in \( \mathcal{F}_L \) a carrier set \( \mathcal{I}(i) \). Moreover, we assume that for every element \( a \in \mathcal{I}(i) \) in the carrier set there is exactly one constant symbol \( c_a \in \mathcal{F}_L \), called a value. The set of all value symbols of sort \( i \) is denoted \( \mathcal{Val}_i \), and we write \( \mathcal{Val} \) for \( \bigcup_i \mathcal{Val}_i \). For instance, if the sort of integers occurs in \( \mathcal{F}_L \) then there is a subset \( \mathcal{Val}_{\mathbb{Z}} \subseteq \mathcal{F}_L \) containing a value \( c_i \) for every \( i \in \mathbb{Z} \).

Moreover, we assume a fixed interpretation \( \mathcal{J} \) that assigns to every symbol \( f \in \mathcal{F}_L \) a function \( f_{\mathcal{J}} \) of appropriate sort, and such that \( (c_a)_{\mathcal{J}} = a \) for value symbols \( c_a \), i.e., value symbols are interpreted as the represented element. The interpretation \( \mathcal{J} \) naturally extends to theory terms without variables by setting \( [f(t_1, \ldots, t_n)]_{\mathcal{J}} = f_{\mathcal{J}}([t_1]_{\mathcal{J}}, \ldots, [t_n]_{\mathcal{J}}) \). In particular, we assume a sort \( \text{bool} \) such that \( \mathcal{I}(\text{bool}) = \{\top, \bot\} \) with values \( \mathcal{Val}_{\text{bool}} = \{\text{true}, \text{false}\} \) such that \( \text{true}_{\mathcal{J}} = \top \), and \( \text{false}_{\mathcal{J}} = \bot \). We also assume that \( \mathcal{F}_L \) contains equality symbols \( \approx_i \) for every theory sort \( i \), and a symbol \( \land \) interpreted as logical conjunction.

Theory terms of sort \text{bool} are called constraints, and a constrained term is a pair \((t, \varphi)\) of a term \( t \) and a constraint \( \varphi \). A substitution \( \gamma \) is a valuation if its range is a subset of \( \mathcal{Val} \). A constraint \( \varphi \) is valid, denoted \( \models \varphi \), if \( [\varphi]_{\mathcal{J}} = \top \) for all valuations \( \gamma \), and satisfiable if \( [\varphi]_{\mathcal{J}} = \top \) for some valuation \( \gamma \).

Logically Constrained Rewriting. A constrained rewrite rule is a triple \( \ell \rightarrow r [\varphi] \) where \( \ell, r \in \mathcal{T}(\mathcal{F}, \mathcal{V}), \ell \not\in \mathcal{V}, \varphi \) is a constraint, and \( \text{root}(\ell) \in \mathcal{F}_T \). If \( \varphi = \text{true} \) then the constraint is omitted. For a rule \( \rho : \ell \rightarrow r [\varphi] \) we use \( \text{lhs}(\rho) = \ell \) and \( \text{rhs}(\rho) = r \) to denote its left- and right-hand sides, respectively. A set of constrained rewrite rules is called a logically constrained term rewrite system (LCTRS for short). For
an LCTRS $R$, its defined symbols $F_D$ are all root symbols of left-hand sides, that is, $F_D = \{\text{root}(\ell) \mid \ell \to r \ [\varphi] \in R\}$. In the remainder we assume that LCTRSs are left-linear, that is, all variables $x$ occur at most once in the left-hand side $\ell$ of a rule $\ell \to r \ [\varphi]$. An LCTRS $R$ is a transition system if all rules in $R$ are of the form $f(\ell_1, \ldots, \ell_n) \to g(r_1, \ldots, r_m) \ [\varphi]$ such that $f, g \in F_D$, all $\ell_i \in V$, and all $r_j$ are in $T(F_L, V)$; if moreover the background theory associated with $F_L$ is the theory of integers then $R$ is an integer transition system (ITS).

The fixed TRS $R_{\text{calc}}$ is given by the (infinite) set of rules $f(s_1, \ldots, s_n) \to u$ such that $f \in F_L \setminus \text{Val}$, $s_i \in \text{Val}$ for all $1 \leq i \leq n$, and $u \in \text{Val}$ is the value symbol of $[f(s_1, \ldots, s_n)]_T$. A rewrite step using $R_{\text{calc}}$ is called a calculation step and denoted $\to_{\text{calc}}$. A rule step $s \to^\rho r$ using a rule $\rho: \ell \to r \ [\varphi]$ satisfies $s = C[\ell\sigma], t = C[r\sigma]$, and $\sigma$ respects $\varphi$; where a substitution $\sigma$ is said to respect a constraint $\varphi$ if $\varphi\sigma$ is valid and $\sigma(x) \in \text{Val}$ for all $x \in \text{Var}(\varphi)$. The substitution in the notation $\to^\rho_\sigma$ is mostly omitted, and a rule step simply denoted $\to^\rho$. For an LCTRS $R$, we denote the relation $\to_{\text{calc}} \cup \{\to_\rho \}_{\rho \in R}$ by $\to_R$. The above rewrite step is innermost, denoted $s \to^1 r$, if all proper subterms of $\ell \sigma$ are in normal form wrt. $\to_R$. The subscript $R$ is dropped if clear from the context. Given binary relations $R$ and $S$, we write $R/S$ for $S^* \cdot R \cdot S^*$. For LCTRSs $R$ and $S$ we abbreviate $\to_R / \to_S$ by $\to_{R/S}$, and $\to_R / \to_{\text{calc}}$ by $\to_{R/\text{calc}}$.

Example 4 (continued from Example 3). The LCTRS $R_2$ indicated in Example 2 expressing the predicate $\text{max\_length}/3$, makes use of the sorts $\text{int}$, $\text{list}$ and $\text{bool}$. Further, $F_L$ consist of symbols $::$ and $[]$ for lists, $\cdot$, $+$, $-$, $\leq$, and $\geq$ as well as values $n$ for all $n \in \mathbb{Z}$, with the usual interpretations on $\mathbb{Z}$ and lists of integers. Then $R$ admits the following rewrite steps:

$$\text{len}([1,2],2) \to \text{len}([2],2-1) \to_{\text{calc}} \text{len}([2],1) \to \text{len}([],1-1) \to_{\text{calc}} \text{len}([],0) \to c_0$$

Note that in LCTRS rewriting, calculation steps like the subtractions in Ex. 4 are explicit in the $\to_{\text{calc}}$ relation, in contrast to ITSs or related formalisms [32], where simplification is implicit. Explicit calculation steps have advantages for certain analysis tasks [40][20].

By the definition of a rewrite step wrt. an LCTRS, variables in constraints need to be substituted by values. Hence non-innermost steps are only possible if nested redexes occur below unconstrained variables. For instance, in a term $f(f(2))$ only the inner $f$ call constitutes a redex for the rule $f(x) \to x \ [x > 0]$. Therefore innermost rewriting is a natural restriction for LCTRSs.

Algebras. We assume mappings $|\cdot|_\iota : \mathcal{I}(\iota) \to \mathbb{N}$ for every sort $\iota$, playing the role of norms to measure size. For instance, one might take the absolute values for integers, the size of arrays, and the unsigned integer value for bit-vectors. The subscript $\iota$ is omitted if the sort of the argument is clear from the context.

\footnote{This is only a very mild restriction: non-left-linear rules are rare in practice; and moreover further occurrences of $x$ in $\ell$ can be substituted by a fresh variable $x'$, adding $x \approx x'$ to $\varphi$. Though this means that $x$ can only be substituted by theory terms in rewrite sequences, for innermost evaluation this is not a limitation.}
We consider well-founded algebras $\mathcal{A}$ over the natural numbers and the Booleans, with interpretation functions $f^A$ for all $f \in F_T \cup F_L$, cf. [8,38]. An interpretation $f^A$ is weakly monotone if $a \geq_N b$ implies $f^A(\ldots , a_{i-1}, a, a_{i+1}, \ldots) \geq_N f^A(\ldots , a_{i-1}, b, a_{i+1}, \ldots)$ for all $f \in F_T$. An algebra is weakly monotone if all its interpretation functions are weakly monotone. An assignment $\alpha$ maps variables to values in $\mathbb{N}$. By $t^A$ we denote the interpretation of a term $t$ based on $\mathcal{A}$, and by $[\alpha]_A(t)$ the interpretation of $t$ based on $\mathcal{A}$ and assignment $\alpha$. In order to bound complexity, we need algebras that comply with the given complexity measures.

**Definition 1.** A measure interpretation is given by an algebra $\mathcal{M}$ with carrier $\mathbb{N}$, and measures $| \cdot |$, for all sorts $\imath$. The interpretation $t^\mathcal{M}$ of a term $t$ is $|t|$, if $t \in \mathcal{V}$ has sort $\imath$, and $f^\mathcal{M}(t_1^\mathcal{M}, \ldots, t_n^\mathcal{M})$ if $t = f(t_1, \ldots, t_n)$. In addition, we demand that $f^\mathcal{M}(t_1^\mathcal{M}, \ldots, t_n^\mathcal{M}) \geq \sup[f(t_1, \ldots, t_n)]^\mathcal{M}$ for all values $t_1, \ldots, t_n$.

In the following we suit interpretations (aka ranking functions) to LCTRSs. The ternary relation $\geq^\mathcal{M}_\imath$ is defined as $s \geq^\mathcal{M}_\imath t$ if and only if $[\alpha]_\mathcal{M}(s) \geq [\alpha]_\mathcal{M}(t)$ is satisfied for all assignments $\alpha$ that respect $\varphi$. Similarly, $s \geq^\mathcal{M}_\varphi t$ if and only if $[\alpha]_\mathcal{M}(s) \geq [\alpha]_\mathcal{M}(t)$ holds for all assignments $\alpha$ that respect $\varphi$.

**Definition 2.** We call an LCTRS $\mathcal{R}$ weakly compatible with a measure interpretation $\mathcal{M}$ if $\ell \geq^\mathcal{M}_\varphi r$ for all $\ell \rightarrow r [\varphi] \in \mathcal{R}$, and strictly compatible if $\mathcal{R}$ is weakly compatible and in addition $\ell >^\mathcal{M}_\varphi r$ for some $\ell \rightarrow r [\varphi] \in \mathcal{R}$.

If $\mathcal{M}$ is clear from the context, $\mathcal{R}_\geq$ is the set of $\ell \rightarrow r [\varphi] \in \mathcal{R}$ such that $\ell >^\mathcal{M}_\varphi r$.

**Example 5.** Consider the measure interpretation $\mathcal{M}$ such that $m^\mathcal{M}_\imath(x, y, z) = y$, $m^\mathcal{M}_\varphi(x, y, z) = z$, $x +^\mathcal{M} y = x +_N y$, $x -^\mathcal{M} y = \max(x -_N y, 0)$, $\geq^\mathcal{M}_\imath$ is $\geq_N$, and $v^\mathcal{M} = \max(v, 0)$ for all $v \in \mathbb{Z}$. The LCTRS $\mathcal{R}'$ consisting of the rules (2), (4), and (8) from Ex. [3] is strictly compatible with $\mathcal{M}$, since the rules (2) and (8) are weakly decreasing, while (4) is strictly decreasing.

## 4 Complexity Framework

An LCTRS $\mathcal{R}$ is terminating if $\rightarrow_\mathcal{R}$ is well-founded. In applications like static analysis, termination of a program is often not enough and more precise resource guarantees are needed. In this section we propose suitable runtime complexity notions for LCTRSs.

Following common notions in complexity analysis [6], the derivation height of a term $t$ wrt. a binary relation $\rightarrow$ is defined as follows: $\text{dh}(t_0, \rightarrow) := \sup \{ k \mid \exists t_1, \ldots, t_k, t_0 \rightarrow \cdots \rightarrow t_k \}$. Without loss of generality, we assume that an LCTRS $\mathcal{R}$ has a unique initial term $t_0$ such that $t_0 = \text{init}(\mathbf{x})$ for a vector of input variables $\mathbf{x} = (x_1, \ldots, x_n)$, and the function symbol init does not occur on any right-hand side. Below, we will consider a more general initial state $(t_0, \varphi_0)$, which consists of a term $t_0$ and a constraint $\varphi_0$. The intention is that we consider...
Definition 3. For an LCTRS $R$ and a constrained term $(t, \varphi)$ such that $\forall t \in \text{Var}(t)$, the (innermost) runtime complexity $r_{\text{CT}}(\varphi) = \sup \{dh(t, t_0) | |\varphi| \leq n, \text{for some } \sigma \text{ such that } t \sigma = \varphi|\sigma\}$.

Thus, the runtime complexity of an LCTRS is the maximal number of innermost rule steps in a rewrite sequence that starts with a size-bounded instance of the initial state; calculation steps are not counted. This is common in cost analysis, but it also corresponds to the runtime complexity of a program or ITS, where the number of transitions are counted, but not simplifications of expressions like replacing $1 + 3 \cdot 2$ by $7$.

Definition 4. Consider a rule $\rho: \ell \rightarrow r[\varphi]$ such that $\text{Pos}_{\text{DP}}(r)$ is sorted as $p_1, \ldots, p_k$ with respect to a fixed order on positions. Then the dependency tuple $D_T(\rho)$ of $\rho$ is the constrained rule $\ell^\# \rightarrow \langle (r_1^\#)^\#, \ldots, (r_k^\#)^\# \rangle^k[\varphi]$. For an LCTRS $R$, $D_T(R) = \bigcup_{\rho \in R} D_T(\rho)$.

Here $(\ldots)_k$ is a fresh tuple symbol for every arity $k$ (but the subscript will be dropped for simplicity). For an initial state $s_0 = (t_0, \varphi_0)$, let $s_0^\#$ denote $(t_0^\#, \varphi_0)$.

Definition 5 (Dependency Graph). Let $R$ be an LCTRS and $D \subseteq D_T(R)$. The $(D, R)$-dependency graph (DG) is the directed graph with node set $D$ and edges from $s^\# \rightarrow \langle v_1^\#, \ldots, v_n^\# \rangle[\varphi]$ to $u^\# \rightarrow v[\psi]$ if there is some $i$ such that $t_i^\# \sigma \rightarrow R^u^\# \tau$, for some substitutions $\sigma$ and $\tau$.

The DG is not computable in general, but techniques to obtain approximations are well-known. For instance, the graph in Sect. 2 constitutes a dependency graph approximation for the LCTRS from Ex. 1. Following Noschinski et al. [34], we assume particular interpretation functions for the tuple operators $(\ldots)$. To this end, let a $D_T$-measure interpretation $\mathcal{M}$ be a measure interpretation that interprets $(t_1, \ldots, t_k)^\mathcal{M} = t_1 + \cdots + t_k$, for all $k$.

We consider a set of (upper) bound expressions UB, that is inductively defined as follows: (i) $|x|_\iota \in \text{UB}$ for $x \in V$ of sort $\iota$, (ii) $\omega \in \text{UB}$, (iii) if $p, q \in \text{UB}$
then \( p + q, pq, \) and \( \max(p, q) \) are in UB, and (iv) if \( p \in UB \) and \( k \in \mathbb{N} \) then \( k^p \in UB \) and \( \log_k(p) \in UB \). Given \( p, q \in UB \), we write \( p \leq q \) if \( \alpha[\alpha]_\alpha(p) \leq \alpha[\alpha]_\alpha(q) \) for all substitutions \( \alpha: V \rightarrow \mathbb{N} \). For an initial term \( t_0 = \text{init}(\overline{\alpha}) \) where \( \overline{\alpha} \in V^n \), a vector \( \overline{m} \in \mathbb{N}^n \), and \( p \in UB \), we also denote \( p[m_i/x_i]_{1 \leq i \leq n} \) by \( p(\overline{m}) \).

A triple \( P = ((t, \varphi), \mathcal{D}, \mathcal{R}) \) of a constrained term \( (t, \varphi) \), a set of DTs \( \mathcal{D} \) and an LCTRS \( \mathcal{R} \) is called a (complexity) problem. Following Brockschmidt et al. [9], we next define time and size bound approximations.

**Definition 6.** For a complexity problem \( ((t, \varphi), \mathcal{D}, \mathcal{R}) \) with \( \overline{\alpha} = \text{Var}(t) \), a function \( T: \mathcal{D} \rightarrow UB \) is a runtime approximation if, for all \( \rho \in \mathcal{D} \) and \( \overline{m} \in \mathbb{N}^n \),

\[
T(\rho)(\overline{m}) \geq \sup \{ \text{dh}(t\sigma, \overline{i}^{(\rho)}/\mathcal{D} \cup \mathcal{R}) \mid \|\overline{\alpha}\| \leq_n \overline{m} \text{ and } \sigma \text{ respects } \varphi \}
\]

In words, a runtime approximation \( T(\rho) \) over-approximates how often a rule \( \rho \in \mathcal{D} \) can be used in a derivation starting from the initial term of a problem. For instance, consider Ex. [1] and let \( n^\# \) be the DT corresponding to rule \( n \). Then the function \( T \) such that \( T(1^\#) = 1 \) and \( T(\rho)(\|x_0\|, \|y_0\|, \|z_0\|) = \|x_0\|^2 \) for all other DTs \( \rho \in \mathcal{D} \) is a valid (though not optimal) runtime approximation.

For a complexity problem \( ((t, \varphi), \mathcal{D}, \mathcal{R}) \), the set of transition variables TV is given by tuples denoted \((\rho, y)\) such that \( \rho \in \mathcal{D} \), and \( y \in \text{Var}(\text{lhs}(\rho)) \).

**Definition 7.** For a complexity problem \( ((t, \varphi), \mathcal{D}, \mathcal{R}) \) with \( \overline{\alpha} = \text{Var}(t) \), a function \( S: TV \rightarrow UB \) is a size approximation if

\[
S(\rho, y)(\overline{m}) \geq \sup \{ \|y\| \mid \exists \sigma, u, t\sigma \overrightarrow{\mathcal{R} \cup \mathcal{D} \cup \mathcal{R}} u, \|\overline{\alpha}\| \leq_n \overline{m} \}
\]

for \((\rho, y) \in TV \) such that substitution \( \sigma \) respects \( \varphi \), and \( \overline{m} \in \mathbb{N}^n \).

A size approximation over-approximates how large an instantiation of a variable in the left-hand side of a rule in \( \mathcal{D} \) can get, in a derivation from the initial state.

A tuple \((T, S)\) is a bound approximation for \( P = ((t, \varphi), \mathcal{D}, \mathcal{R}) \) if \( T \) and \( S \) are runtime and size approximations for \( P \). We next define a complexity framework in the spirit of Avanzini and Moser [4].

**Definition 8.** Given a complexity problem \( P = ((t, \varphi), \mathcal{D}, \mathcal{R}) \), a (complexity) judgement is a statement \( \vdash P: (T, S) \), for functions \( T: \mathcal{D} \rightarrow UB \) and \( S: TV \rightarrow UB \). The judgement is valid if \((T, S)\) is a bound approximation for \( P \).

A complexity processor \( \text{Proc} \) is an inference rule on complexity judgements:

\[
\vdash P_1: (T_1, S_1), \ldots, \vdash P_k: (T_k, S_k) \quad \text{Proc} \quad \vdash P: (T, S)
\]

which is sound if \( \vdash P: (T, S) \) is valid whenever all \( \vdash P_i: (T_i, S_i) \) are valid.

For \( P = ((t_0, \varphi_0), \mathcal{D}, \mathcal{R}) \) a DT \( \ell \rightarrow r \) \([\psi] \in \mathcal{D} \) is initial if \( t_0 = \ell \sigma \) for some \( \sigma \) such that \( \varphi_0 \land \psi \sigma \) is satisfiable, and \( \text{root}(t_0) \) does not occur on any right-hand side in \( \mathcal{D} \). The initial processor for a problem \( P = ((t_0, \varphi_0), \mathcal{D}, \mathcal{R}) \) is given by

\[
\vdash P: (T, S_\omega) \quad \text{Initial}
\]
where $T(\rho_1) = 1$ for all initial $\rho_1 \in D$, and $T(\rho) = \omega$ otherwise; and $S_x(\rho, x) = \omega$ for all $(\rho, x) \in TV$. The processor Initial is clearly sound. For instance, the DT $\text{init}^\#(x, y, z) \rightarrow m^\#(x, y, z)$ originating from rule (1) is initial in Ex. 1.

We sometimes write $\vdash ((t, \varphi), D, R)$: $C$ to abbreviate that there is a judgement $\vdash ((t, \varphi), D, R): (T, S)$ for some pair $(T, S)$ such that $C = \sum_{\rho \in D} T(\rho)$.

The following result states that valid judgements bound the runtime complexity of LCTRSs. It can be proven similar to [6, Theorem 6], relying on the properties of dependency tuples for innermost rewriting.

**Theorem 1.** If an LCTRS $R$ with initial state $(t, \varphi)$ admits the valid judgement $\vdash ((t^\#, \varphi), DT(R), R \cup R_{\text{calc}}): (T, S)$ then $rc^{(t^\#, \varphi)}_R \leq \sum_{\rho \in DT(R)} T(\rho)$ holds.

## 5 Processors

This section presents processors that implement the complexity framework from Sect. 4, in particular showing how the respective ITS techniques [9] carry over.

**Interpretation Processors.** Compatible interpretations are a standard tool in resource analysis, cf. [9,10,11].

**Lemma 1.** Let $P = ((t_0, \varphi_0), D, R)$ and $M$ a DT-measure interpretation with which $R$ is weakly, and $D$ is strictly compatible. Then the following processor is sound, where $T'(\rho) = (t_0)^M$ for all $\rho \in D_>$, and $T'(\rho) = T(\rho)$ otherwise:

$$\vdash P: (T, S) \quad \text{Interpretation}$$

For instance, for Ex. 1 one can take the interpretation $M$ such that $\text{split}^M = 0$ and $f^M = 1$ for all other $f \in F_T$, and $F_L$ is interpreted as in Ex. 5. $R_1$ is strictly compatible since all rules are weakly and rule (5) is strictly decreasing. This justifies updating a given runtime approximation by $T((5)) = 1 = \text{init}^\#(\varphi)^M$.

Next, we adapt [9, Theorem 3.6] to our setting, by which runtime bounds can be obtained using an interpretation that orients the given LCTRS partially.

For a $(D, R)$-dependency graph $G$ and some $D' \subseteq D$, the set $\text{pre}_G(D')$ consists of all edges $(\rho_1, \rho_2)$ in $G$, such that $\rho_1 \in D \setminus D'$ and $\rho_2 \in D'$.

**Lemma 2.** Suppose $P = (s_0, D, R)$ is a complexity problem such that $D' \subseteq D$ has no initial DTs and $R$ is weakly, and $D'$ is strictly compatible with a weakly monotone DT-measure interpretation $M$. Then the following processor is sound:

$$\vdash P: (T, S) \quad \text{TimeBounds}$$

such that $T'(\rho) = \sum_{(\gamma, \delta) \in \text{pre}_G(D')} T(\gamma) \cdot \ell_2^M(\delta_\delta)$ for all $\rho \in D_>$, where $\delta : \ell_2 \rightarrow r_2 [\varphi_2]$ and $\delta_\delta$ the vector $S(\delta, y_1), \ldots, S(\delta, y_k)$ for $y_1, \ldots, y_k$ the variables in $\ell_2$. Otherwise, for all $\rho \in D \setminus D'$ set $T'(\rho) = T(\rho)$.
**Size Bounds.** Size bound estimation was developed for ITs [9], and is in general harder to achieve for LCTRSs. However, in many practical examples, a sufficient approximation is feasible. To that end, we consider the following transition variable graph (corresponding to a result variable graph for ITs [9]).

First, we define a notion to capture that two variables influence each other. Let \( y, z \) be variables in \( \rho: \ell \to r [\varphi] \). Then \( y \) depends on \( z \) if \( y = z \), or there is a variable \( y' \) in \( \rho \), such that both \( y', y \) appear in a common conjunct of (a CNF of) \( \varphi \), and \( y' \) depends on \( z \). Given a term \( t \), let \( \text{lcap}(t) \) be \( f(\text{lcap}(t_1), \ldots, \text{lcap}(t_n)) \) if \( t = f(t_1, \ldots, t_n) \) and \( f \notin \mathcal{F}_D \), and a fresh variable otherwise.

**Definition 9.** A transition variable graph \( G_{TV} \) for \((t_0, \varphi_0), (\mathcal{D}, \mathcal{R})\) with \( DG \) has as nodes \( TV(D) \), and there is an edge from \((\rho_1, y)\) to \((\rho_2, z)\), for \( \rho_1: \ell_1 \to r_1 [\varphi_1] \) and \( \rho_2: \ell_2 \to r_2 [\varphi_2] \), if \( \ell_2|\rho_2 = z \) and

- in \( G \) there is an edge from \( \rho_1 \) to \( \rho_2 \),
- there is a variable \( y' \) and a position \( p_1 \in \text{Pos}(r_1) \) such that \( r_1|_{p_1} = y' \) and \( y' \) depends on \( y \),
- the maximal position \( q_1 \in \text{Pos}(\text{lcap}(r_1)) \) with \( q_1 \leq \rho_1 \) and the maximal position \( q_2 \in \text{Pos}(\text{lcap}(\ell_2)) \) with \( q_2 \leq \rho_2 \) are not parallel.

This edge is labelled by \( \alpha \in \text{UB} \) such that for all rewrite sequences of the form \( t_0 \sigma \rightarrow^{*}_{\mathcal{D}, \mathcal{R}} r_1 \rightarrow^{*}_{\mathcal{R}} \tau u \) for some \( \sigma, \tau \) such that \( \sigma \) respects \( \varphi_0 \), it holds that \( z \tau \in \mathcal{T}(\mathcal{F}_L, \mathcal{V}) \) and \( |z\tau| \leq \alpha \). If no such \( \alpha \) exists, the edge is labelled by \( \omega \).

**Example 6.** Consider again Ex. 1. We first apply chaining, a standard technique in termination analysis [15], to compress the cycles \((9) - (3) - (9)\) and \((9) - (7) - (9)\) into single-step cycles, such that \((9)\) is replaced by

\[
\psi = x \geq 2 \land u \geq 0 \land v \geq 0 \land x + 1 \geq 2u \land 2u \geq x \land x \geq 2v \land 2v + 1 \geq x
\]

Then we obtain the following transition variable graph:

```
1.x 1.y 1.z → 0.x 0.y 0.z → 2.x 2.y 2.z → 1.x 1.y 1.z
```

where a triple arrow \( a \rightarrow b \) means that there are arrows from \((a, x)\) to \((b, x)\), \((a, y)\) to \((b, y)\), and \((a, z)\) to \((b, z)\), respectively. All edges from transition variables \((a, u)\) can be labeled \(|u|\) in this example.

Next, we use the transition variable graph \( G_{TV} \) to obtain size bound refinements, following the approach of [9]. To that end, we define two processors \( S_{triv} \) and \( S_{triv} \) that refine bounds for trivial and non-trivial SCCs in \( G_{TV} \), respectively. Here, an SCC is trivial if it consists of a single node without an edge to itself.

---

[1] This notion of dependence is rather strong; more precise notions are future work.
Definition 10. Given a size bound $S$, we define $S_{\text{triv}}$ as follows: (i) $S_{\text{triv}}(\rho, y) = |y|$ if $\rho$ is initial; (ii) $S_{\text{triv}}(\rho, y) = \max\{\alpha \mid (\rho', z) \rightarrow^\alpha (\rho, y) \text{ in } G_{TV}\}$, if $(\rho, y)$ does not occur in a non-trivial SCC of $G_{TV}$; (iii) otherwise $S_{\text{triv}}(\rho, y) = S(\rho, y)$.

We distinguish three types of edges in $G_{TV}$: for an edge labelled $\alpha$, (i) $\alpha \in E_\approx$ if $\alpha = a, a \in \mathbb{N}$ or $\alpha = |x|$ for some $x \in V$; (ii) $\alpha \in E_+$ if $|x| + a_\alpha \geqslant \alpha$ for some $x \in V$ and $a_\alpha \in \mathbb{N}$; (iii) $\alpha \in E_\times$ if $c + \sum_{x \in \mathcal{X}} a_x |x| \geqslant \alpha$ for $c, a_x \in \mathbb{N}$ and $\mathcal{X} \subseteq V$.

For an SCC $C$ in $G_{TV}$, let $C_\alpha$ denote the set of edge labels $\alpha$ of edges in $C$. For a transition variable graph $G_{TV}$, let $\text{pre}(\rho, y)$ be the set of all direct predecessors of $(\rho, y)$ in $G_{TV}$. The following bounds correspond to those developed in [9] for the ITS setting. The case for $E_\times$ is omitted for reasons of space, but carries over in a similar way.

Definition 11. Let $C$ be a non-trivial SCC in $G_{TV}$. Given runtime and size bounds $(T, S)$, let $S_{\text{sec}}$ be defined as

- if $C_\alpha \subseteq E_\approx$ then $S_{\text{sec}}(\rho, y) = \max\{\alpha \mid \alpha \in C_\alpha\}$,
- if $C_\alpha \subseteq E_+$ then let $\alpha_{\text{pre}} = \max\{S(\rho', z) \mid (\rho', z) \in \text{pre}(\rho, y) \setminus C\}$ and $S_{\text{sec}}(\rho, y) = \max\{\{\alpha_{\text{pre}}\} \cup \{a_\alpha \mid \alpha \in C_\alpha\}\} + \sum_{\rho \in D} T(\rho) \cdot \max\{a_\alpha \mid \alpha \in C \setminus E_\approx\}$

and $S_{\text{sec}}(\rho, y) = S(\rho, y)$ otherwise, for all $\rho \in C$ and $(\rho, y) \in TV$.

Lemma 3. The following processors are sound:

\[
\vdash (s_0, D, R): (T, S) \quad \vdash (s_0, D, R): (T, S_{\text{triv}}) \quad \vdash (s_0, D, R): (T, S_{\text{sec}}) \quad \text{Size Bounds}
\]

6 Processors for Splitting and Loop Summary

In this section we present two new processors that allow for decomposition of a problem into subproblems, as well as loop analysis based on recurrence relations.

Splitting. A subgraph is forward closed if it is closed under successors. For a DT $\rho: \ell \rightarrow r [\varphi]$, let $\text{constr}(\rho)$ be a constraint $\psi$ such that $\text{Var}(\psi) \subseteq \text{Var}(\ell)$ and $\psi \tau$ is valid for all $\sigma, \tau$ such that $I_0^\# \sigma \rightarrow^{\varphi}_{D, \mathcal{R}, \text{calc}} \tau \ldots$. While not computable in general, $\text{constr}(\rho)$ can be approximated by a reachability analysis in the DG.

Definition 12. For a given $(s_0, D, R)$, suppose $D = D_1 \cup D_2$ such that there is a $(D, R)$-dependency graph $G$ for $R$ with subgraphs $G_1$ and $G_2$ with node sets $D_1$ and $D_2$, respectively, all initial DTs are in $D_1$, and $G_2$ is forward closed. Then $(D_1, D_2)$ constitutes a splitting for $(s_0, D, R)$ and $G$ with respect to $(G_1, G_2)$.

Lemma 4. Let $P = (s_0, D, R)$ be a problem with DG $G$ which admits the splitting $(D_1, D_2)$ with respect to $(G_1, G_2)$. Then the following processor is sound:

\[
\vdash P: (T, S) \quad \vdash (s_0, D_1, R): C_0 \wedge_{\delta_i \in \text{pre}(D_2)} \vdash ((\ell_i, \psi_i), D_2, R): C_i \quad \text{Split}
\]

where $\text{pre}(D_2) = \{\delta_1, \rho_1\}, \ldots, \{\delta_m, \rho_m\}$, $\ell_i = \text{lhs}(\rho_i)$, $\psi_i = \text{constr}(\rho_i)$ and $S_{\rho_i}$ is $(S(\rho_i, y_1), \ldots, S(\rho_i, y_{\ell_i}))$ such that $y_{j1}, \ldots, y_{j\ell_i} = \text{Var}(\ell_i)$, for $1 \leqslant i \leqslant m$. 

12
Summarising Loops. We next propose a technique to analyse (sub)problems with self-loops whose DG is of the following shape:

![Diagram of a loop with self-loops](image)

To simplify the presentation, we only consider cycles formed by a single DT, as indicated in the picture above. The result generalizes to longer cycles, but chaining can often reduce these cases to the simpler situation discussed here.

Definition 13. A complexity problem $P = ((t, \varphi), \mathcal{D}, \mathcal{R})$ with $(\mathcal{D}, \mathcal{R})$-dependency graph $G$ is cyclic if the following conditions are satisfied:

- there is some $\rho \in \mathcal{D}$ not reachable from $\mathcal{D} \setminus \{\rho\}$ in $G$ of the form

$$\rho : f(\overline{x}) \rightarrow (f(\overline{r}_1), \ldots, f(\overline{r}_m), h_1(\overline{t}_1), \ldots, h_p(\overline{t}_p)) \quad [\psi]$$ (2)

modulo the order of arguments of in $\ldots$, such that $t = f(\overline{x})$, $m > 0$, $x_i \in \mathcal{V}$ and $r_{ij} \in T(\mathcal{F}_L, \mathcal{V})$, for all $1 \leq i \leq m$ and $1 \leq j \leq k$;

- $\varphi \models \overline{x} \succ r_i \forall 1 \leq i \leq k$, where $\overline{x} \succ r_i$ means $\bigwedge_j x_j \geq r_{ij} \land (\bigvee_j x_j > r_{ij})$;

- there is a vector of values $\overline{b}$ in $(\mathbb{N} \cup \{-\infty\})^k$ called termination condition such that $\psi \models \overline{x} \geq \overline{b}$ and $\varphi \models \overline{x} \geq \overline{b}$, and

- for all $P_i = ((h_i(\overline{t}_i), \varphi \land \psi), \mathcal{D}, \mathcal{R})$, the judgement $\vdash P_i : C_i$ is valid.

Lemma 5. Let $P = ((t, \varphi), \mathcal{D}, \mathcal{R})$ be a cyclic complexity problem of the form (2) with termination condition $\overline{b}$ and subproblems $((h_i(\overline{t}_i), \varphi \land \psi), \mathcal{D}, \mathcal{R})$, for $1 \leq i \leq p$. Then the following processor is sound:

$$\vdash \bigwedge (h_i(\overline{t}_i), \varphi \land \psi), \mathcal{D}, \mathcal{R}) : (T_i, S_i)$$

$$\vdash (s_0, \mathcal{D}, \mathcal{R}) : (\lambda \rho. F(\overline{x}), \lambda \rho. \max, S_i(\rho))$$

Recurrence

for all $F$, $\hat{f}$ such that $F(\overline{x}) \geq \hat{f}(\overline{x}) \forall \overline{x}$ and $\hat{f}$ is a solution to the recurrence

$$f(\overline{x}) = f(\overline{r}_1) + \ldots + f(\overline{r}_m) + H(\overline{x}) \quad f(\overline{b}) = 0$$ (3)

and where $H(\overline{x}) \geq \sum_{i=1}^{p} C_i(\overline{x})$ for all $\overline{x}$.

Example 7. Consider Ex. 1 with chaining as applied in Ex. 6. For instance, $C_0((\text{s}_0(x, u, v), \psi), \mathcal{D}, \mathcal{R}_1) = x + 1$ and $C_3((\text{s}_3(x, u, v), \psi), \mathcal{D}, \mathcal{R}_1) = u + v + 1$ are valid complexity approximations for subproblems, so we can set $H(x, u, v) = x + u + v + 1$, and since $u, v \geq x/2$, for $/$ being integer division, we can set $H'(x, u, v) = 2|x| + 1$. Thus, we solve the recurrence (1) given in Sect. 2. According to one of the cases of the Master Theorem, (1) has a solution in $\mathcal{O}(|x| \cdot \log(|x|))$ which is a complexity approximation according to Lem. 5.

7 Implementation

To evaluate the viability of the presented framework, we prototyped our approach in the complexity analyser $\text{TCT}$ [7], adding the module $\text{tct-lctrs}$ to the
tool suite. Several subroutines require an SMT solver, for which TCT interfaces Yices \cite{14} and Z3 \cite{13}. Currently, only the theory of integers is supported. TCT also computes asymptotic bounds, rather than precise expressions.

All the processors described in this paper are implemented, using the modular processor framework of TCT. These processors are currently arranged in a strategy as follows, where the loop indicates repetition until no further change. Below, we mention some implementation aspects that seem noteworthy.

- Initially, TCT computes dependency tuples and a DG approximation
- The simp processor in the strategy schema above comprises a number of obvious simplification processors: unsatisfiable paths and unreachable rules are eliminated, leaves in the DG obtain their time bound from their predecessors.
- Suitable algebras instantiating the interpretation and time bounds processors (Lems. \ref{lem:interpretation} and \ref{lem:timebounds}) are searched for by means of an SMT encoding, as was already done in the ITS module of TCT previously.
- In order to apply the loop processor (Lem. \ref{lem:loop}), TCT first applies chaining to obtain loops that involve only a single DT (see Appendix B for details). It is then checked whether certain recursion patterns apply. To that end, TCT first attempts to solve subproblems corresponding to the functions $h_1, \ldots, h_p$ separately, obtaining complexities $C_1, \ldots, C_p$ (see the notation of Lem. \ref{lem:loop}). Then, it is checked whether a function $H$ corresponding to one of the known recursion patterns satisfies $H(\pi) \geq \sum_i C_i$ using an SMT call.
- The splitting processor (Lem. \ref{lem:split}) leaves a lot of choice to the implementation where to split. We currently use it to enable the loop processor, which requires a very particular problem shape.

This strategy lets TCT also derive logarithmic bounds, e.g. for the problems divideByTwo and direct_n_log_n \cite{9}. The TCT branch and examples are available.\footnote{\url{http://cl-informatik.uibk.ac.at/users/swinkler/lctrs_complexity/}}

8 Conclusion

This paper presented the first complexity framework for LCTRSs. We conclude by relating to earlier work in the area, before indicating leads for future research.

Related work. In the last decades there has been significant progress in the area of fully automated resource analysis, showing that it can be both practicable and scalable, see e.g. \cite{38,1,24,3,35,39,5,18,26,32}. In the following, we indicate related work that directly influenced our framework, or employed similar methods.

Our framework differs from earlier work by Avanzini and Moser \cite{6} in three important respects: first, constraints over arbitrary background theories are supported, second, complexity is not expressed in terms of the size of the initial term
but in terms of measure functions, and third, sublinear bounds can be derived.
While innermost rewriting is a rather natural restriction for LCTRSs, *call by need* strategies could be considered in the future for LCTRSs, too.

LCTRSs generalise ITSs, the complexity analysis of which is subject to a comprehensive line of research [9,34]. Our approach gracefully extends the alternating time and size bound technique by Brockschmidt et al. [9], as the ITS case is fully covered. In addition, we can obtain sublinear bounds, and support further modularization. Moreover, LCTRSs offer native support for full recursion.

Sublinear bounds are beyond the scope of this earlier work, but can be inferred by some other tools. Albert et al. [1] apply refinements to linear ranking functions and support sufficient criteria for divide-and-conquer patterns. This allows the tool PUBS to recognize logarithmic and $O(n \log(n))$ bounds for some problems, though the *mergesort* example [1] cannot be handled. Chatterjee et al. [10] use synthesis ranking functions extended by logarithmic and exponential terms, making use of an insightful adaption of Farkas’ and Handelman’s lemmas. The approach is able to handle examples such as *mergesort*. In contrast to our work this amounts to a whole-program analysis. Further, extensibility to a constraint formalism as LCTRS is unclear. Wang et al. [37] present an ML-like language with type annotations, also using the Master Theorem to handle divide-and-conquer-like recurrences.

**Future work.** We see exciting directions for future work both on a theoretical and an application level. Various additional processors can be conceived for our complexity framework, for instance forms of dependency pairs for non-innermost rewriting [31,25], knowledge propagation and narrowing [34].

Simplification systems as, for instance, employed in compiler toolchains (cf. Ex. 3) or SMT solvers constitute a highly relevant application domain, since these routines operate in performance-critical contexts. In order to tackle such systems, techniques for *derivational* complexity of LCTRSs need to be developed.

On the application level, LCTRSs constitute a natural backend for complexity analysis of constraint logic programs, since constraints can be natively expressed. Our experiments with logic programs did not take backtracking into account, but suitably adapting the transformational frameworks as e.g. established by Giesl et al. [23] or Albert et al. [2] to LCTRSs, this is not a showstopper. We thus plan to support CLP as a frontend of our analysis, possibly taking into account *labelling strategies* that control the instantiation of query terms. We furthermore plan to support C programs as a frontend. C programs with integers, as considered in the Termination Competition [5] can be expressed as ITSs. LCTRSs offer more flexibility and can support also strings and floats, as the respective theories are supported by SMT solvers. Just like for the case of CLP, this requires the development of suitable complexity-reflecting transformations. Extensive experiments are planned to evaluate our method on ITS benchmarks, (constrained) logic programs [31], and problems from the software competition [6].

---

References


11. Ş. Ciobăcă and D. Lucanu. A coinductive approach to proving reachability properties in logically constrained term rewriting systems. In *Proc. 9th IJCAR*, volume 10900, pages 295–311, 2018. [https://doi.org/10.1007/978-3-319-94205-6_20](https://doi.org/10.1007/978-3-319-94205-6_20)


16. A. Flores-Montoya. Upper and lower amortized cost bounds of programs expressed as cost relations. In *Proc. 21st FM*, volume 9995 of *LNCS*, pages 254–273, 2016. [https://doi.org/10.1007/978-3-319-48989-6_16](https://doi.org/10.1007/978-3-319-48989-6_16)


A Proofs

Lemma 1 1 Suppose $P = ((t_0, \varphi_0), D, R)$ is a complexity problem and $\mathcal{M}$ a DT-measure interpretation with which $R$ is weakly, and $D$ is strictly compatible.

Then the following processor is sound:

$$\vdash P : (T, S) \quad \text{Interpretation}$$

where $T'(\rho) = (t_0^\#)^{\mathcal{M}}$ for all $\rho \in D_>$, and $T'(\rho) = T(\rho)$ otherwise.

Proof. By assumption, $\mathcal{R}$ has a unique initial state $(t_0, \varphi_0)$. Let $\mathcal{T} = \text{Var}(t_0)$, and $\rho \in D_>$. We show that for $\rightarrow_T = \rightarrow_{(D \cup \mathcal{R} \cup \text{calc}) \setminus \{\rho\}}$, $(t_0^\#)^{\mathcal{M}}(\overline{m}) \geq \sup \{k \mid \exists \sigma. t_0^\# \sigma \xrightarrow{k} u, |\mathcal{T}\sigma| \leq_n \overline{m} \text{ and } \sigma \text{ respects } \varphi_0\}$

for all $\overline{m} \in \mathbb{N}^n$. Consider a substitution $\sigma$ such that $|\mathcal{T}\sigma| \leq_n \overline{m}$, and a (potentially infinite) rewrite sequence in which $k \in \mathbb{N} \cup \{\omega\}$ steps apply rule $\rho$:

$$t_0^\# \sigma = v_0 \xrightarrow{\mathcal{T}} u_1 \xrightarrow{\rho} v_1 \xrightarrow{\mathcal{T}} u_2 \xrightarrow{\rho} v_2 \xrightarrow{\rho} \ldots \quad (4)$$

The goal is to verify $(t_0^\#)^{\mathcal{M}}(\overline{m}) \geq k$. If $k = 0$ then this is obvious because the domain of $\mathcal{M}$ is $\mathbb{N}$, so suppose $k > 0$. By compatibility, we have $v_0^\mathcal{M} \geq u_1^\mathcal{M}$, $u_i^\mathcal{M} > v_i^\mathcal{M}$, and $v_i^\mathcal{M} > u_{i+1}^\mathcal{M}$, for all $i \geq 1$. Thus $k$ must be finite, and there can be at most $(t_0^\#)^{\mathcal{M}}(\overline{m}) \rho$-steps in $[4]$.

Lemma 2 Suppose $P = (s_0, D, R)$ is a complexity problem such that $D' \subseteq D$ has no initial dependency tuples and $\mathcal{R}$ is weakly, and $D'$ is strictly compatible with a DT-measure interpretation $\mathcal{M}$. Then the following processor is sound:

$$\vdash P : (T, S) \quad \text{TimeBounds}$$
such that \( T'(\rho) = \sum_{(\gamma, \delta) \in \text{pre}(D')} T(\gamma) \cdot \ell_2^{M}(S_\delta) \) for all \( \rho \in D'_\geq \), where \( \delta : \ell_2 \rightarrow r_2 \) \([\varphi_2]\) and \( S_\delta \) the vector \( S(\delta, y_1), \ldots, S(\delta, y_k) \) for \( y_1, \ldots, y_k \) the variables in \( \ell_2 \).

Otherwise, for all \( \rho \in D \setminus D'_\geq \), set \( T'(\rho) = T(\rho) \).

Proof. It suffices to show that

\[
T'(\rho)(m) \geq \sup \{ k \mid t_0^#(\rho) ((\rightarrow_{\mathcal{R}} \cdot \rightarrow_{\mathcal{R}})^* u \text{ and } |\pi| \leq n \text{ and } \sigma \text{ respects } \varphi_0) \}
\]

holds for all \( \rho \in D'_\geq \) and \( m \in \mathbb{N}^n \). Fix some \( \rho \in D'_\geq \) and let \( \rightarrow_{\mathcal{R}} \rightarrow_{\mathcal{R} \cup \text{calc}} \). Consider a substitution \( \sigma \) such that \( |\pi| \leq m \), and a (potentially infinite)

rewrite sequence

\[
t_0^\#(\rho) = v_0 \xrightarrow{u_1} v_1 \xrightarrow{u_2} v_2 \xrightarrow{u_3} \ldots \tag{5}
\]

with \( k \in \mathbb{N} \cup \{ \omega \} \) steps using \( \rho \). It has to be verified that the expression \( B := \sum_{(\gamma, \delta) \in \text{pre}(D')} T(\gamma) \cdot \ell_2^{M}(S_\delta) \) satisfies \( B(m) \geq k \). We can write sequence \ref{eq:5} as

\[
t_0^\#(\rho) = v_0 \xrightarrow{u_1} w_1 \xrightarrow{u_1} v_1 \xrightarrow{u_2} w_2 \xrightarrow{u_2} v_2 \xrightarrow{u_3} \ldots \tag{6}
\]

for \( u = \mathcal{R} \cup \mathcal{D} \cup \text{calc} \setminus D' \). Every step \( w_i \xrightarrow{u} w_i \) before a term \( w_i \) must use a rule \( \rho_i \) such that \( (\rho_i, \delta_i) \in \text{pre}(D') \), for all \( i > 0 \). Fix some \( (\gamma, \delta) \in \text{pre}(D') \), and let \( i \) be such that \( (\rho_i, \delta_i) = (\gamma, \delta) \). Moreover, in the subsequence \( w_i \xrightarrow{u} \ldots \xrightarrow{u} \) the value under the interpretation \( M \) is weakly decreasing, and for every \( \rho \)-step there is a strict decrease as \( \rho \in D'_\geq \). Hence the number of \( D'_\geq \) steps (and in particular \( \rho \)-steps) in \( w_i \xrightarrow{u} \ldots \xrightarrow{u} \) is bounded by \( N := \mathit{lhs}(\delta)^M(S_\delta) \). The number of \( \gamma \) steps followed by \( \delta \) steps in \( \mathbb{R} \) is clearly bounded by \( T(\gamma) \). Summing up the product \( N \cdot T(\gamma) \) over all pairs \((\gamma, \delta) \in \text{pre}(D') \) yields \( B(m) \geq k \).

\[\]

Lemma 5 Let \( P = (s_0, D, \mathcal{R}) \) be a problem with DG \( G \) which admits the splitting \((D_1, D_2)\) with respect to \((G_1, G_2)\). Then the following processor is sound:

\[\]

\( \vdash P: (T, S) \vdash (s_0, D_1, \mathcal{R}) : C_0 \land \sum_{(\delta_i, \rho_i) \in \text{pre}(D_2)} \vdash ((\ell_i, \psi_i), D_2, \mathcal{R}) : C_i \)

\[\]

Split

where \( \text{pre}(D_2) = \{(\delta_1, \rho_1), \ldots, (\delta_m, \rho_m)\} \), \( \ell_i = \mathit{lhs}(\rho_i) \), \( \psi_i = \text{const}(\rho_i) \) and \( S_\rho \),

is \((S(\rho_1, y_{i1}), \ldots, S(\rho_i, y_{ik}))\) such that \( y_{i1}, \ldots, y_{ik} = \text{Var}(\ell_i) \), for \( 1 \leq i \leq m \).

Proof. We can represent an evaluation tree as

\[
\{t_0\} \xrightarrow{D_1 \cup \mathcal{R}} \langle u_1, \ldots, u_k \rangle \xrightarrow{\delta_1, \ldots, \delta_k} \langle v_1, \ldots, v_k \rangle \xrightarrow{D_2 \cup \mathcal{R}} \langle w_1, \ldots, w_l \rangle \tag{7}
\]

since the evaluation must start with \( D_1 \) steps, and there are no edges from \( G_2 \) to \( G_1 \). (Here the notation \( \langle \ldots \rangle \) indicates a flattened list of nested \( \langle \ldots \rangle \) terms.)

For the rule \( \delta_i \) used in \( u_i \xrightarrow{\delta_i} v_i \), \( 1 \leq i \leq k \), there is some \( \rho_i \) such that \( (\delta_i, \rho_i) \in \text{pre}(D_2) \). For \( (\delta, \rho) \in \text{pre}(D_2) \) and some \( i \) such that \( (\delta, \rho) = (\delta_i, \rho_i) \), we can thus overapproximate the number of \( D_2 \)-steps starting from \( v_i \) by \( C(\mathit{lhs}(\rho), D_2, \mathcal{R}) \). This yields an expression in \( \text{Var}(\mathit{lhs}(\rho)) \), and we obtain a respective expression in \( \text{Var}(t_0) \) by applying \( C(\mathit{lhs}(\rho), D_2, \mathcal{R}) \) to \( S_\rho \). In order to account for multiple occurrences of \( \delta \) in \( \delta_1, \ldots, \delta_k \), this expression is multiplied by \( T(\delta) \), and summed up over all pairs \((\delta, \rho)\).
Lemma 6 Let \( P = ((t, \varphi), \mathcal{D}, \mathcal{R}) \) be a cyclic complexity problem of the form \([2]\) with termination condition \( \bar{b} \) and subproblems \((h_i(t_i), \varphi \land \psi), \mathcal{D}, \mathcal{R}\), for \( 1 \leq i \leq p \). Then the following processor is sound:

\[
\vdash \Box((h_i(t_i), \varphi \land \psi), \mathcal{D}, \mathcal{R}) \colon (T_i, S_i) \\
\vdash (s_0, \mathcal{D}, \mathcal{R}) : (\lambda \rho. F(\bar{x}), \lambda \rho. \max_i S_i(\rho))
\]

Recurrence

for every function \( F \) such that \( F(\bar{x}) \geq \hat{f}(\bar{x}) \) for all \( \bar{x} \) and solution \( \hat{f} \) to the recurrence system

\[
f(\bar{x}) = f(\bar{r}_1) + \ldots + f(\bar{r}_m) + H(\bar{x}) \quad \hat{f}(\bar{b}) = 0 \quad (4)
\]

and where \( H(\bar{x}) \geq \sum_{i=1}^p C_i(\bar{x}) \) for all \( \bar{x} \).

Proof. As \( \rho \) has no incoming edges in \( G \) except from itself, \( G \) is of the form depicted at the start of this section. So an overapproximation of a solution to

\[
f(\bar{x}) = f(\bar{r}_1) + \ldots + f(\bar{r}_m) + \sum_{i=1}^p C_i(\bar{x}) \quad \hat{f}(\bar{b}) = 0 \quad (8)
\]

is a complexity approximation for \((t, \mathcal{D}, \mathcal{R})\). It remains to show that the overapproximation of \( C(\bar{x}) := \sum_{i=1}^p C_i(\bar{x}) \) is sound. To that end, let \( F'(\bar{x}) \) be a solution to \([3]\) and \( F(\bar{x}) \) a solution to \([8]\). It is easy to verify that \( F'(\bar{x}) \geq F(\bar{x}) \) holds:

\[
F'(\bar{x}) = F'(\bar{r}_1) + \ldots + F'(\bar{r}_m) + H(\bar{x}) \quad (F' \text{ solves } [3])
\]

\[
\geq F(\bar{r}_1) + \ldots + F(\bar{r}_m) + C(\bar{x}) \quad (\text{IH as } \bar{r}_i \supseteq \bar{x}, \text{ and } H(\bar{x}) \geq C(\bar{x}))
\]

\[
= F(\bar{x}) \quad (F \text{ solves } [8]) \quad \square
\]

B Further Processors

In this section we outline how two standard techniques from termination and complexity analysis fit into our framework, namely chaining and usable rules.

Their adaptation is straightforward, hence we describe these techniques only in the appendix for the sake of completeness.

Chaining. Rule chaining is a commonly used complexity-reflecting transformation \([15,7]\) that can also be used for LCTRSs. Its aim is to combine rules that are applied subsequently, which can simplify further analysis.

Definition 14. Let \( P \) be a problem \((s_0, \mathcal{D}, \mathcal{R})\) with \( \mathcal{D} \) \( G \) such that \( \rho : t \rightarrow u \ [\varphi] \in \mathcal{D} \) has successors \( \delta_i : \ell_i \rightarrow r_i \ [\psi_i] \) in \( G \), for \( 1 \leq i \leq k \). Then \( \rho \) and \( \Delta = \{\delta_1, \ldots, \delta_k\} \) are chainable if there are positions \( p_i \) and substitutions \( \sigma_i \) such that

\[
u | p_i = \ell_i \sigma_i, \text{ and all } \psi_i \sigma_i \text{ are in } T(\mathcal{F}_L, V)\]

Then \( \text{chain}(\rho, \delta_i) = t \rightarrow u[r_i \sigma]_{p_i} \ [\varphi \land \psi_i \sigma_i] \), and \( \text{chain}(\rho, \Delta) = \{\text{chain}(\rho, \delta_i)\}_{1 \leq i \leq k} \).

\footnote{We assume that these are all possibilities to create a redex in \( u \) wrt. rules in \( \Delta \).}
Lemma 6. For a complexity problem \((s_0, D, R)\), the following processor is sound:

\[
\frac{\vdash (s_0, D \cup \text{chain}(\rho, \Delta) \setminus \{\rho\}, R) : (T, S)}{\vdash (s_0, D, R) : (T', S')}
\]

Chaining

where \(\rho\) and \(\Delta\) are chainable, the updated runtime approximation is

\[
T'(\delta_i) = T(\delta_i) + T(\text{chain}(\rho, \delta_i)), \quad T'(\rho) = \sum T(\text{chain}(\rho, \delta_i)) \quad \text{and} \quad T'(\gamma) = T(\gamma)
\]

otherwise; and \(S'(\rho, y) = \max\{S(\text{chain}(\rho, \delta_i))\}_{i}, S'(\delta_i, y) = \max(S(\delta_i, y), \psi)\) if \(\psi \in T(F_L, V)\) and \(\omega\) otherwise, and \(S'(\gamma, y) = S(\gamma, y)\) for all other \(\gamma \in D\).

Example 8. In two chaining steps, the cycles (9) − (3) − (9) and (9) − (7) − (9) in Ex. 1 can be chained to single-step cycles, such that (9) is replaced by

\[
m(x, y, z) \rightarrow \langle m_0(x, u, v), m(u, u, v), m(v, u, v), m_3(x, u, v) \rangle [\psi]
\]

\[
\psi = x \geq 2 \land u \geq 0 \land v \geq 0 \land x + 1 \geq 2u \land 2u \geq x \land x \geq 2v \land 2v + 1 \geq x
\]

Usable Rules

Complexity problems can be simplified by restricting rewrite rules to usable rules, a common concept in rewriting \[13,16\]. We next give a possible definition for the case of LCTRSs. A weaker version thereof was already integrated in the definition of dependency pairs in \[27, Thm. 12\]. A stronger version taking constraints into account is conceivable as well, but not in the focus of this paper.

Definition 15. For an LCTRS \(R\) and a symbol \(f \in F_T\), let \(RLs_R(f)\) consist of all rules \(\ell \rightarrow r \ [\varphi] \in R\) such that \(\text{root}(\ell) = f\). For any term \(t\), \(UR_R(t)\) is the smallest set such that \(UR_R(t) = \emptyset\) if \(t \in T(F_L, V)\), and \(UR_R(t) = RLs_R(f) \cup \bigcup_{\ell \rightarrow r \in RLs_R(f)} UR(r) \cup \bigcup_{i} UR(t_i)\) if \(t = f(t_1, \ldots, t_n)\) such that \(f \in F_T\). For any set of DTs \(D\), let \(UR_R(D) = \bigcup_{\rho \in D} UR_R(\text{rhs}(\rho))\).

Lemma 7. For a complexity problem \((s_0, D, R)\), the following processor is sound:

\[
\frac{\vdash (t_0, D, UR_R(D)) : (T, S)}{\vdash (s_0, D, R) : (T, S)}
\]

Usable Rules
Confluence and Commutation for Nominal Rewriting Systems with Atom-Variables

Kentaro Kikuchi\(^1\) and Takahito Aoto\(^2\)

\(^1\) RIEC, Tohoku University, Sendai, Japan
kentaro.kikuchi@ric.tohoku.ac.jp

\(^2\) Faculty of Engineering, Niigata University, Niigata, Japan
aoto@ie.niigata-u.ac.jp

Abstract. Nominal rewriting was introduced as an extension of first-order term rewriting by a binding mechanism based on the nominal approach. Recently, a new format of nominal rewriting has been introduced where rewrite rules are defined with atom-variables rather than atoms. In this paper, we investigate the difference between the new format and the original nominal rewriting, and prove confluence and commutation for some classes of rewriting systems whose rules have no proper overlaps which are computed using nominal unification with atom-variables. The properties we prove are expected to be used in a form of program transformation that is realized as an equivalence transformation of rewriting systems.

Key words: Variable binding, Alpha-equivalence, Nominal unification, Nominal rewriting, Atom-variable, Confluence, Commutation

1 Introduction

Confluence is a fundamental property of rewriting systems that guarantees uniqueness of results of computation. Commutation is a generalization of confluence to a property of computation by two rewriting systems. These properties are important in applications of rewriting techniques; for instance, they are essential to correctness of a form of program transformation, called equivalence transformation of rewriting systems [3, 9, 24]. For first-order term rewriting systems, confluence and commutation have been well studied, and many criteria to ensure them have been developed (e.g. [7, 8, 11, 16, 18, 22, 23]).

Nominal rewriting [4, 5] was introduced as an extension of first-order term rewriting by a binding mechanism based on the nominal approach [6, 17], where variables that are possibly bound are called atoms. A distinctive feature of nominal rewriting is that α-conversion and capture-avoiding substitution are not relegated to the meta-level—they are explicitly dealt with at the object-level. Some basic confluence criteria such as Rosen’s criterion [18] (orthogonal systems are confluent), Knuth-Bendix’s criterion [11] (terminating systems with joinable critical pairs are confluent) and Huet’s criterion [8] (left-linear systems with parallel closed critical pairs are confluent) have been discussed in the case of nominal rewriting [2, 4, 10, 20, 21].
Recently, Kutz and Schmidt-Schauß [12] have introduced a somewhat different format of nominal rewriting from the original one. In their systems, rewrite rules are written using atom-variables for which atoms are substituted in each rewrite step. The use of atom-variables appears to be problematic, because for keeping binding structures correct, it is necessary to use permutations (or injections) on atoms instead of substitutions. Their systems, however, have a device that makes substitutions for atom-variables injective so that this problem can be avoided.

In the present paper, we treat nominal rewriting with atom-variables in the style of [12], looking into how it differs from nominal rewriting in previous work. We study confluence and commutation for some classes of nominal rewriting systems including those which are difficult to represent by previous approaches.

The main differences between the format of [12] and those in the traditional style are explained as follows. First, as mentioned above, rewrite rules are written with atom-variables rather than atoms, and rewriting is performed through substituting atoms for atom-variables where the substitution is not necessarily injective. This is particularly effective when representing rewrite rules with some atoms that are not bound (after being substituted for atom-variables). Motivations and examples showing its usefulness can be found in Section 1 of [19]. If the substitution should be injective, one can add appropriate freshness constraints on atom-variables in the rewrite rule, which are used in matching process. In this way, the definition of rewrite relation is given without involving equivariance (as in [4]) or parametrised permutations (as in [20]), and so simpler than those in previous work. However, there is a not small price to pay for these advantages. Since terms with atom-variables are not objects for rewriting to be analysed any more, the language has split into one for rewrite rules and one for objects for rewriting. In [12], the latter is the language of ground nominal terms, i.e. nominal terms with neither (term-)variables nor atom-variables, and those variables are used only for representing rewrite rules. Accordingly, the confluence properties treated in [12] are restricted to those on ground nominal terms, and in this sense weaker properties than those discussed in previous work.

Contributions of the paper. The contributions of the present paper are summarised as follows:

– We prove confluence on ground terms for orthogonal nominal rewriting systems with atom-variables. In Theorem 6.7 of [12], only local confluence is stated for orthogonal systems, and no proof of confluence has been given.
– We prove the commutation property of mutually orthogonal nominal rewriting systems with atom-variables. The commutation property has never been studied in previous work on nominal rewriting.

Although the properties we prove are restricted to those on ground terms, they are enough for application to the form of program transformation mentioned at the beginning of this section.

Organisation of the paper. The paper is organised as follows. In Section 2, we explain basic notions of nominal rewriting systems with atom-variables. In Sec-
tion 3, we study confluence and commutation for some classes of nominal rewriting systems with atom-variables. In Section 4, we discuss related work and conclude with suggestions for further work.

2 Nominal rewriting systems with atom-variables

In this section, we introduce basic notions on nominal rewriting systems with atom-variables [12]. Unlike the original nominal rewriting [4], the framework of [12] uses two different languages: one is for objects for rewriting, called ground nominal terms, and the other is for components of rewrite rules.

2.1 Preliminaries

First, we introduce some notations on nominal terms.

A nominal signature $\Sigma$ is a set of function symbols ranged over by $f, g, \ldots$. We fix a countably infinite set $X$ of variables ranged over by $X, Y, \ldots$, a countably infinite set $A$ of atoms ranged over by $a, b, \ldots$, and a countably infinite set $X_A$ of atom-variables ranged over by $A, B, \ldots$. We assume that $\Sigma, X, A$ and $X_A$ are pairwise disjoint. Unless otherwise stated, different meta-variables for objects in $\Sigma, X, A$ or $X_A$ denote different objects.

The domain $\text{dom}(\phi)$ of a mapping $\phi : D \rightarrow E$ is the set $D$ if $D \neq E$ and $\{d \in D \mid \phi(d) \neq d\}$ if $D = E$. A mapping $\phi : D \rightarrow E$ is finite if its domain $\text{dom}(\phi)$ is finite. For finite mappings $\phi$ and $\psi$ with $\text{dom}(\phi) \cap \text{dom}(\psi) = \emptyset$, we define the mapping $\phi \cup \psi$ with $\text{dom}(\phi \cup \psi) = \text{dom}(\phi) \cup \text{dom}(\psi)$ by $(\phi \cup \psi)(d) = \phi(d)$ if $d \in \text{dom}(\phi)$ and $(\phi \cup \psi)(d) = \psi(d)$ if $d \in \text{dom}(\psi)$.

2.2 Ground nominal terms

In this subsection, we introduce the set of ground nominal terms, which we call $NL_a$ following [12, 19].

The set $NL_a$ of ground nominal terms, or simply ground terms, are generated by the following grammar:

$$t, s ::= a \mid [a]t \mid f t \mid \langle t_1, \ldots, t_n \rangle$$

Ground terms of the forms in the right-hand side are called, respectively, atoms, abstractions, function applications and tuples. We assume that function applications bind more strongly than abstractions. We abbreviate $f (\langle \rangle)$ as $f$, referring to it as a constant. An abstraction $[a]t$ is intended to represent $t$ with $a$ bound. The set of free atoms occurring in $t$, denoted by $\text{FA}(t)$, is defined as follows: $\text{FA}(a) = \{a\}$; $\text{FA}([a]t) = \text{FA}(t) \setminus \{a\}$; $\text{FA}(f t) = \text{FA}(t)$; $\text{FA}(\langle t_1, \ldots, t_n \rangle) = \bigcup_i \text{FA}(t_i)$.

Example 1. The nominal signature of the $\lambda$-calculus has two function symbols $\text{lam}$ and $\text{app}$. The ground nominal term $\text{app}(\text{lam}(\langle a \text{lam}(\langle b \text{app}(b, a) \rangle) \rangle), b)$ represents the $\lambda$-term $(\lambda a. \lambda b. b)\ a$ in the usual notation. For this ground term $t$, we have $\text{FA}(t) = \{b\}$. □
A swapping is a pair of atoms, written \((a \ b)\). Permutations \(\pi\) are bijections on \(A\) such that \(\text{dom}(\pi)\) is finite. Permutations are represented by lists of swappings applied in the right-to-left order. For example, \(((b \ c)(a \ b))(a) = c, ((b \ c)(a \ b))(b) = a, ((b \ c)(a \ b))(c) = b\). The permutation action \(\pi \cdot t\), which operates on terms extending a permutation on atoms, is defined as follows: \(\pi \cdot a = \pi(a); \pi \cdot (a[t]) = [\pi \cdot a](\pi \cdot t); \pi \cdot (f \ t) = f \cdot (\pi \cdot t); \pi \cdot ([t_1, \ldots, t_n]) = (\pi \cdot t_1, \ldots, \pi \cdot t_n)\).

Positions are finite sequences of positive integers. The empty sequence is denoted by \(\varepsilon\). The set of positions in a ground term \(t\), denoted by \(\text{Pos}(t)\), is defined as follows: \(\text{Pos}(a) = \{\varepsilon\}; \text{Pos}(\{a \mid t\}) = \{1p \mid p \in \text{Pos}(t)\} \cup \{\varepsilon\}\). \(\text{Pos}(\{t_1, \ldots, t_n\}) = \bigcup\{\{ip \mid p \in \text{Pos}(t_i)\} \cup \{\varepsilon\}\. The subterm of \(t\) at a position \(p \in \text{Pos}(t)\) is written as \(t_p\).

A context is a ground term in which a distinguished constant \(\Box\) occurs. The ground term obtained from a context \(C\) by replacing each \(\Box\) at positions \(p_i\) by ground terms \(t_i\) is written as \(C[t_1, \ldots, t_n]_{p_1, \ldots, p_n}\) or simply \(C[t_1, \ldots, t_n]\).

A pair \(a \# t\) of an atom \(a\) and a ground term \(t\) is called a freshness constraint. The rules in Figure 1 define the validity of freshness constraints. Note that the defined \(\vdash_{NL_a} a \# t\) coincides with \(a \notin FA(t)\).

The rules in Figure 2 define the relation \(\vdash_{NL_a} t \approx_{\alpha} s\). This is a congruence relation \([4]\) and coincides with usual \(\alpha\)-equivalence (i.e. the relation reached by renamings of bound atoms) \([6]\). The definition of \(\vdash_{NL_a} t \approx_{\alpha} s\) will be used in some of the proofs afterwards.


The following properties are shown in \([4, 25]\).

**Proposition 1.** 1. \(\vdash_{NL_a} a \# t\) if and only if \(\vdash_{NL_a} \pi \cdot a \# \pi \cdot t\).
2. \(\vdash_{NL_a} t \approx_{\alpha} s\) if and only if \(\vdash_{NL_a} \pi \cdot t \approx_{\alpha} \pi \cdot s\).
3. If \(\vdash_{NL_a} a \# t\) and \(\vdash_{NL_a} t \approx_{\alpha} s\) then \(\vdash_{NL_a} a \# s\).

### 2.3 Nominal term expressions

Next, we introduce the set of term expressions used in rewrite rules, which we call \(NL_{AS}\) following \([12, 19]\).
The set $NL_A$ of nominal term expressions, or simply term expressions, are generated by the following grammar:

$$
e ::= v | \pi \cdot X | [v] e | f e | \langle e_1, \ldots, e_n \rangle
\pi ::= \emptyset | (v v') \cdot \pi
v ::= \pi \cdot A$$

where $\pi$ and $v$ are non-terminals for permutation expressions and atom expressions, respectively. A term expression of the form $\pi \cdot X$ is called a moderated variable. Also, an expression of the form $\pi \cdot X$ or $\pi \cdot A$ is called a suspension. We abbreviate $\emptyset \cdot X$ and $\emptyset \cdot A$ as $X$ and $A$, respectively, if there is no ambiguity. We write $\text{Var}_X(e)$ for the set of variables occurring in a term expression $e$, and $\text{Var}_A(e)$, $\text{Var}_X(\pi)$ and $\text{Var}_X(v)$ for the sets of atom-variables occurring in expressions $e$, $\pi$ and $v$, respectively. For a term expression $e$, we define $\text{Var}_X,\text{Var}_A(e)$ as $\text{Var}_X(e) \cup \text{Var}_A(e)$. A term expression $e$ is linear if each variable $X \in \text{Var}_X(e)$ occurs only once in $e$.

The set $\text{Pos}(e)$ of positions in a term expression $e$ is defined similarly to that for a ground term (using atom expressions for atoms) with the additional clause that $\text{Pos}(\pi \cdot X) = \{e\}$. The subexpression of $e$ at a position $p \in \text{Pos}(e)$ is written as $e|_p$. If $p \neq e$, then $e|_p$ is called a proper subexpression. A position $p \in \text{Pos}(e)$ is called a variable position if $e|_p$ is a moderated variable, and a non-variable position otherwise.

A ground substitution is a finite mapping that assigns ground terms to variables and atoms to atom-variables. We use $\sigma, \delta$ for ground substitutions. We write $\sigma_X$ and $\sigma_A$ for ground substitutions obtained from $\sigma$ by restricting the domain to $\text{dom}(\sigma) \cap X$ and $\text{dom}(\sigma) \cap A$, respectively. When $\text{Var}_X,\text{Var}_A(e)$, the application of $\sigma$ on $e$ is written as $e\sigma$ and called a ground instance of $e$. (Similarly for expressions $\pi$ and $v$.) The application of $\sigma$ does not simply replace the variables $X$ and atom-variables $A$ occurring in $e$ by $\sigma(X)$ and $\sigma(A)$, but, when replacing $X$ and $A$ of suspensions $\pi \cdot X$ and $\pi \cdot A$, induce permutation actions $\pi\sigma\cdot(\sigma(X))$ and $\pi\sigma\cdot(\sigma(A))$ viewing the list $\pi\sigma$ as a permutation. So we have $e\sigma \in NL_a$ for any ground instance $e\sigma$.

A pair $v\#e$ of an atom expression $v$ and a term expression $e$ is called a freshness constraint expression. A finite set of freshness constraint expressions is called a freshness context. For a freshness context $\nabla$, we define $\text{Var}_{X,\nabla}(\nabla) = \bigcup_{v\#e \in \nabla} (\text{Var}_{X}(v) \cup \text{Var}_{X,\nabla}(e))$ and $\nabla \sigma = \{v\sigma\#e\sigma \mid v\#e \in \nabla\}$.

Now we recall nominal unification problems with atom-variables [19].

**Definition 1.** Let $\Gamma$ be a finite set of equations of the form $e_1 \approx e_2$ where $e_1$ and $e_2$ are term expressions, and let $\nabla$ be a freshness context. Then the pair $(\Gamma, \nabla)$ is called a variable-atom nominal unification problem (VANUP for short).

**Definition 2 (Solution of a VANUP).** A ground substitution $\sigma$ is a solution of a VANUP $(\Gamma, \nabla)$ if $\vdash_{NL_a} e_1 \sigma \approx e_2 \sigma$ for every equation $e_1 \approx e_2 \in \Gamma$ and $\vdash_{NL_a} v\#e\sigma$ for every freshness constraint expression $v\#e \in \nabla$. A VANUP $(\Gamma, \nabla)$ is solvable if there exists a solution of $(\Gamma, \nabla)$. 

5
Example 2. Consider the nominal signature for the $\lambda$-calculus in Example 1, and let $P$ be the VANUP ($\{\text{lam}(\lambda\,A\text{app}(X,A)) \approx \text{lam}(\lambda\,B\,Y)\}, \{A\#X\})$. Then, the ground substitution $[A := a, B := b, X := c, Y := \text{app}(c,b)]$ is a solution of $P$. 

2.4 Nominal rewriting systems with atom-variables

Next we define nominal rewrite rules and nominal rewriting systems with atom-variables.

Definition 3. A nominal rewrite rule with atom-variables, or simply rewrite rule, is a triple of a freshness context $\nabla$ and term expressions $l, r \in NLAS$ such that $\text{Var}_X.X_{A}(\nabla) \cup \text{Var}_X.X_{A}(r) \subseteq \text{Var}_X.X_{A}(l)$ and $l$ is not a moderated variable. We write $\nabla \vdash l \rightarrow r$ for a rewrite rule, and identify rewrite rules modulo renaming of variables and atom-variables. A rewrite rule $\nabla \vdash l \rightarrow r$ is left-linear if $l$ is linear.

Definition 4 (Nominal rewriting system with atom-variables). A nominal rewriting system with atom-variables ($NRS_{AS}$ for short) is a finite set of rewrite rules. An $NRS_{AS}$ is left-linear if so are all its rewrite rules.

The following example of an $NRS_{AS}$ corresponds to Example 8 of [20] written in the style of traditional nominal rewriting. Note that the freshness constraint $A\#B$ is used to mean that distinct atoms should be substituted for the atom-variables $A$ and $B$.

Example 3. We extend the signature in Example 1 by a function symbol $\text{sub}$. By $\text{sub}(\lambda\,[a]t,s)$, we represent an explicit substitution $t(a := s)$. Then, an $NRS_{AS}$ to perform $\beta$-reduction is defined by the rule (Beta):

$$\nabla \vdash \text{app}(\lambda\,[A]X,Y) \rightarrow \text{sub}(\lambda\,[A]X,Y) \quad \text{(Beta)}$$

together with an $NRS_{AS} \mathcal{R}_{sub}$ to execute substitution:

$$\nabla \vdash \text{sub}(\lambda\,[A]\text{app}(X,Y),Z) \rightarrow \text{app}(\text{sub}(\lambda\,[A]X,Z),\text{sub}(\lambda\,[A]Y,Z)) \quad \text{(sub\text{app})}$$

$$\nabla \vdash \text{sub}(\lambda\,[A]A,X) \rightarrow X \quad \text{(sub\text{var})}$$

$$A\#B \vdash \text{sub}(\lambda\,[A]B,X) \rightarrow B \quad \text{(sub\text{var}\ell})$$

$$A\#B, B\#Y \vdash \text{sub}(\lambda\,[A]\text{lam}(\lambda\,[B]X),Y) \rightarrow \text{lam}(\lambda\,[B]\text{sub}(\lambda\,[A]X,Y)) \quad \text{(sub\text{lam})}$$

In a standard notation, the system $\mathcal{R}_{sub}$ is represented as follows:

$$\nabla \vdash (XY)(A := Z) \rightarrow (X(A := Z))(Y(A := Z)) \quad \text{(sub\text{app})}$$

$$\nabla \vdash A(A := X) \rightarrow X \quad \text{(sub\text{var})}$$

$$A\#B \vdash B(A := X) \rightarrow B \quad \text{(sub\text{var}\ell})$$

$$A\#B, B\#Y \vdash (\lambda B.X)(A := Y) \rightarrow \lambda B.(X(A := Y)) \quad \text{(sub\text{lam})}$$

In the sequel, $\vdash_{NL\alpha}$ is extended to mean to hold for all members of the set in the right-hand side.
**Definition 5 (Rewrite relation).** Let $R = \nabla \vdash l \rightarrow r$ be a rewrite rule. For ground terms $s, t \in NL_\alpha$, the **rewrite relation** is defined by

$$s \rightarrow_{(R,p,\sigma)} t \iff s = C[s']_p, \vdash_{NL_\alpha} s' \approx_{\alpha} l\sigma, \ t = C[r\sigma]_p$$

We write $s \xrightarrow{R} t$ if there exists $\sigma$ such that $s \rightarrow_{(R,p,\sigma)} t$. We write $s \xrightarrow{R} t$ if there exist $p$ and $\sigma$ such that $s \rightarrow_{(R,p,\sigma)} t$. For an $NRS_{AS} R$, we write $s \xrightarrow{R} t$ if there exists $R \in \mathcal{R}$ such that $s \rightarrow_{R} t$.

The following is an example of rewriting by the $NRS_{AS}$ in Example 3. It corresponds to Example 10 of [20] using traditional nominal rewriting. We see that a substitution for atom-variables and the additional freshness constraint can provide a mechanism to avoid capture of a free atom (as far as rewriting on ground terms is concerned).

**Example 4.** Using the rule ($\text{Beta}$) in Example 3, we see that the ground term representing $(\lambda a.\lambda b. b)a$ rewrites to $(\lambda b. ba)(a := b)$, that is, we have

$$\operatorname{app}(\lambda a.\lambda b. ba) \rightarrow_{(\text{Beta},\pi,\sigma)} \operatorname{sub}(\lambda a.\lambda b. ba)(a := b),$$

where $\sigma$ is the ground substitution $[A := a, X := \lambda a.\lambda b. ba, Y := b]$. The resulting ground term rewrites further to a normal form $\lambda a.\lambda b. ba$ in four steps with rules of the system $\mathcal{R}_{\text{sub}}$. Here we give a detail of the first step with rule ($\text{sub}_{\text{lam}}$) to see how capture of a free atom is avoided.

Let $s = \operatorname{sub}(\lambda a.\lambda b. ba)(a := b), b).$ Since the rule has a freshness context $\nabla = \{A\#B, B\#Y\}$, to apply (sub$_{\text{lam}}$) to $s$ at the position $p = \varepsilon$, it is necessary to find a ground substitution $\sigma$ such that $\vdash_{NL_\alpha} s \approx_{\alpha} \lambda a.\lambda b. ba$ and $\vdash_{NL_\alpha} s' \approx_{\alpha} \lambda a.\lambda b. ba$. Here one cannot take $\sigma$ with $\sigma(B) = b$, which together with $\sigma(Y) = b$ from the condition for $\approx_{\alpha}$ contradicts $\vdash_{NL_\alpha} s$. So we take, e.g., $\sigma = [A := a, B := c, X := \lambda a.\lambda b. ba, Y := b]$ to satisfy the conditions, and get $\lambda a.\lambda b. ba(\lambda a.\lambda b. ba)(a := b)$ as the result of rewriting. □

As observed in Proposition 4.4 of [12], the following holds.

**Lemma 1.** Let $R = \nabla \vdash l \rightarrow r$ be a rewrite rule, and let $s, t$ be ground terms. If $p \in \text{Pos}(s)$ and $s \xrightarrow{R} t$ then $\pi \cdot s \xrightarrow{R} \pi \cdot t$ for every permutation $\pi$.

**Remark 1.** In previous papers on nominal rewriting except for the authors’, the rewrite relation is often defined so that $\alpha$-equivalent terms are allowed on the result of rewriting, like $l \approx_{\alpha} C[r\sigma]$. However, such a definition makes arguments by induction difficult, since from $s \xrightarrow{R} t$ one can only say $l \approx_{\alpha} C[r\sigma]$ for some $C$ and $\sigma$, where $s$ is of the form $C[s']$ but $t$ is not necessarily of the form $C[t']$.

### 2.5 Overlaps and orthogonality

The notion of overlap is defined using nominal unification with atom-variables.
**Definition 6 (Overlap).** Let \( R_i = \nabla_i \vdash l_i \rightarrow r_i \ (i = 1, 2) \) be rewrite rules. We assume without loss of generality that \( \text{Var}_{X,X}(l_i) \cap \text{Var}_{X,X}(l_2) = \emptyset \). If the variable-atom nominal unification problem \( \{(l_1 \approx l_2)_{\rho}, \nabla_1 \cup \nabla_2\} \) is solvable for some non-variable position \( p \) of \( l_2 \), then we say that \( R_1 \) overlaps on \( R_2 \), and the situation is called an overlap of \( R_1 \) on \( R_2 \). If \( R_1 \) and \( R_2 \) are identical modulo renaming of variables and atom-variables, and \( p = \varepsilon \), then the overlap is said to be self-rooted. An overlap that is not self-rooted is said to be proper.

**Example 5.** Let \( R_1 \) and \( R_2 \) be the rules (\textbf{Eta}) \( A \# X \vdash \text{lam}([A]\text{app}(X,A)) \rightarrow X \) and (\textbf{Beta}) \( \text{app}(\text{lam}([B]Y),Z) \rightarrow \text{sub}([B]Y,Z) \), respectively. Then, \( R_1 \) overlaps on \( R_2 \), since the VANUP \( \{\text{lam}([A]\text{app}(X,A)) \approx \text{app}(\text{lam}([B]Y),Z)\}_{\eta_{(= l\text{am}([B]Y))}}, \{A\# X\} \) is solvable as seen in Example 2. This overlap is proper. \( \Box \)

In first-order term rewriting, self-rooted overlaps do not matter, and only proper overlaps need to be analysed. However, in the case of nominal rewriting, that is not enough as discussed in [20].

Using the above notion of overlap, we define the notions of orthogonality of an \( NRS_{AS} \) and mutual orthogonality of two \( NRS_{AS} \)'s.

**Definition 7 (Orthogonality).** An \( NRS_{AS} \) \( R \) is orthogonal if it is left-linear and for any rewrite rules \( R_1, R_2 \in R \), there exists no proper overlap of \( R_1 \) on \( R_2 \).

**Definition 8 (Mutual orthogonality).** \( NRS_{AS} \)'s \( R_1 \) and \( R_2 \) are mutually orthogonal if they are left-linear and for any rewrite rules \( R_1 \in R_1 \) and \( R_2 \in R_2 \), there exists no overlap of \( R_1 \) on \( R_2 \), and there exists no overlap of \( R_2 \) on \( R_1 \).

Unlike in first-order term rewriting, orthogonality is not enough to guarantee a confluence property of an \( NRS_{AS} \) as seen in the following example.

**Example 7.** Consider the nominal rewriting system \( R_{uc-\eta} \) with the only rewrite rule (\textbf{Uncond-eta}) \( \text{lam}([A]\text{app}(X,A)) \rightarrow X \). This system \( R_{uc-\eta} \) is orthogonal. However, \( \text{lam}([a]\text{app}(a,a)) \rightarrow_{\text{Uncond-eta}} a \) and \( \text{lam}([a]\text{app}(a,a)) \rightarrow_{\text{Uncond-eta}} b \). The latter holds since \( \forall_{\text{Dc}} \text{lam}([a]\text{app}(a,a)) \approx_{\alpha} \text{lam}([b]\text{app}(b,b)) \) (cf. the third condition of rewrite relation in Definition 5). \( \Box \)

## 3 Confluence and commutation for left-linear nominal rewriting systems with atom-variables

In this section, we study confluence and commutation properties of left-linear \( NRS_{AS} \)'s. The properties are defined modulo the equivalence relation \( \approx_{\alpha} \) in terms of abstract reduction systems [14]. To do so, we first introduce some notations.
Let $\succeq$ be a binary relation. We write $\succeq^\circ$ for the reflexive closure and $\succeq^*$ for the reflexive transitive closure. If $\succeq$ is written using $\to$, then the inverse $\succeq^{-1}$ is written using $\leftarrow$. In what follows, we write simply $t \simeq_\alpha s$ for $\vdash_{\text{NL}_\alpha} t \simeq_\alpha s$. We use $\circ$ for the composition of two binary relations.

**Definition 9.** Let $\mathcal{R}$ be an NRS$_{\text{AS}}$.

1. Ground terms $s$ and $t$ are joinable modulo $\simeq_\alpha$, denoted by $s \downarrow_{\simeq_\alpha} t$, if $s (\to^*_\mathcal{R} \circ \simeq_\alpha \circ \leftarrow^*_\mathcal{R}) t$.
2. $\to\mathcal{R}$ is confluent modulo $\simeq_\alpha$ iff for every ground terms $s$ and $t$, if $s (\leftarrow^*_\mathcal{R} \circ \to^*_\mathcal{R}) t$ then $\vdash_{\text{NL}_\alpha} s \downarrow_{\simeq_\alpha} t$.
3. $\to\mathcal{R}$ is Church-Rosser modulo $\simeq_\alpha$ iff for every ground terms $s$ and $t$, if $s (\leftarrow^*_\mathcal{R} \cup \to^*_\mathcal{R} \cup \simeq_\alpha)^* t$ then $s \downarrow_{\simeq_\alpha} t$.
4. $\to\mathcal{R}$ is strongly compatible with $\simeq_\alpha$ iff for every ground terms $s$ and $t$, if $s (\simeq_\alpha \circ \to\mathcal{R}) t$ then $s (\leftarrow^*_\mathcal{R} \circ \simeq_\alpha) t$.
5. Let $\mathcal{R}'$ be another NRS$_{\text{AS}}$. $\to\mathcal{R}$ subcommutes with $\to\mathcal{R}'$ modulo $\simeq_\alpha$ iff for every ground terms $s$ and $t$, if $s (\leftarrow^*_\mathcal{R} \circ \to\mathcal{R}') t$ then $s (\leftarrow^*_\mathcal{R} \circ \simeq_\alpha \circ \leftarrow^*_\mathcal{R}) t$.
6. Let $\mathcal{R}'$ be another NRS$_{\text{AS}}$. $\to\mathcal{R}$ commutes with $\to\mathcal{R}'$ modulo $\simeq_\alpha$ iff for every ground terms $s$ and $t$, if $s (\leftarrow^*_\mathcal{R} \circ \to\mathcal{R}') t$ then $s (\leftarrow^*_\mathcal{R} \circ \simeq_\alpha \circ \leftarrow^*_\mathcal{R}) t$.

It is known that Church-Rosser modulo an equivalence relation $\sim$ is a stronger property than confluence modulo $\sim$ [14]. In the rest of this section, we aim to show Church-Rosser modulo $\simeq_\alpha$ for a class of left-linear NRS$_{\text{AS}}$’s, and commutation modulo $\simeq_\alpha$ of two systems in a class of left-linear NRS$_{\text{AS}}$’s.

### 3.1 Uniformity and $\alpha$-stability

To our aim mentioned above, we restrict NRS$_{\text{AS}}$’s by some conditions. First we consider (an adaptation of) the uniformity condition [4]. Intuitively, uniformity means that if an atom $a$ is not free in $s$ and $s$ rewrites to $t$ then $a$ is not free in $t$.

**Definition 10 (Uniformity).** A rewrite rule $\nabla \vdash l \to r$ is uniform if the following holds: for every atom $a$ and every ground substitution $\sigma$ such that $\text{Var}_{X,\nabla\xi}(l) \subseteq \text{dom}(\sigma)$, if $\vdash_{\text{NL}_\alpha} \nabla\sigma$ and $\vdash_{\text{NL}_\alpha} a \text{#} l \sigma$ then $\vdash_{\text{NL}_\alpha} a \text{#} r \sigma$. A rewriting system is uniform if so are all its rewrite rules.

The following properties of uniform rewrite rules are important and will be used in the sequel.

**Proposition 2.** Suppose $s \to\mathcal{R} t$ for a uniform rewrite rule $\mathcal{R}$. Then, for every atom $a$, if $\vdash_{\text{NL}_\alpha} a \text{#} s$ then $\vdash_{\text{NL}_\alpha} a \text{#} t$.

**Proof:** By induction on the structure of $s$. \qed

**Lemma 2.** Let $\mathcal{R}$ be a uniform rewrite rule. If $s' \simeq_\alpha s \to_{\langle \mathcal{R}, p, \sigma \rangle} t$, then there exist $\sigma'$ and $t'$ such that $s' \to_{\langle \mathcal{R}, p, \sigma' \rangle} t' \simeq_\alpha t$.

**Proof:** By induction on the structure of $s$. \qed
Next we introduce the notion of $\alpha$-stability [20]. This notion can be seen as a complement to orthogonality in proving Church-Rosser modulo $\approx_{\alpha}$.

**Definition 11 ($\alpha$-stability).** A rewrite rule $R = \nabla \vdash l \to r$ is $\alpha$-stable if $\vdash_{NL_a} s \approx_{\alpha} s'$, $s \rightarrow_{(R,x,\sigma)} t$ and $s' \rightarrow_{(R,x,\sigma')} t'$ imply $\vdash_{NL_a} t \approx_\alpha t'$. An NRS $\mathcal{R}$ is $\alpha$-stable if so are all its rewrite rules.

### 3.2 Parallel reduction

A key notion for proving confluence of left-linear rewriting systems is parallel reduction. Here we define it inductively, using a particular kind of contexts.

**Definition 12.** The *grammatical contexts*, ranged over by $G$, are the contexts defined by

$$G ::= a \mid [a] \square \mid f \square \mid \langle \square_1, \ldots, \square_n \rangle$$

Let $\mathcal{R}$ be an NRS$_{AS}$. We define the relation $\rightarrow_{\mathcal{R}}$ inductively by the following rules:

$$s_1 \rightarrow_{\mathcal{R}} t_1 \cdots s_n \rightarrow_{\mathcal{R}} t_n \quad (\text{C})$$

$$s \overset{\gamma}{\rightarrow_{\mathcal{R}}} R t \quad (\text{B})$$

where $n (\geq 0)$ depends on the form of $G$. We write $\sigma \rightarrow_{\mathcal{R}} \delta$ to denote $\forall X \in \text{dom}(\sigma_X). X\sigma \rightarrow_{\mathcal{R}} X\delta$ and $\forall A \in \text{dom}(\sigma_X). A\sigma = A\delta$.

**Lemma 3.**

1. If $s \rightarrow_{\mathcal{R}} s$.
2. If $s \rightarrow_{\mathcal{R}} t$ then $C[s] \rightarrow_{\mathcal{R}} C[t]$.
3. If $s \rightarrow_{(R,p,\sigma)} t$ and $R \in \mathcal{R}$ then $s \rightarrow_{\mathcal{R}} t$.
4. If $s \rightarrow_{\mathcal{R}} t$ then $s \rightarrow_{\mathcal{R}} t$.

**Proof.**

1. By induction on the structure of $s$.
2. By induction on the context $C$.
3. By 2 and the rule (B).
4. By induction on the derivation of $s \rightarrow_{\mathcal{R}} t$. \hfill $\Box$

**Lemma 4.** If $s \rightarrow_{\mathcal{R}} t$ then $\pi \cdot s \rightarrow_{\mathcal{R}} \pi \cdot t$.

**Proof.** By induction on the derivation of $s \rightarrow_{\mathcal{R}} t$. If the last applied rule in the derivation is (B), then we use Lemma 1. \hfill $\Box$

**Lemma 5.** If $\sigma \rightarrow_{\mathcal{R}} \delta$ then $e\sigma \rightarrow_{\mathcal{R}} e\delta$.

**Proof.** By induction on the structure of $e$. If $e = \pi \cdot X$, then we use Lemma 4. \hfill $\Box$

**Lemma 6.** Let $\mathcal{R}$ be a uniform NRS$_{AS}$.

1. If $\vdash_{NL_a} a\#s$ and $s \rightarrow_{\mathcal{R}} t$ then $\vdash_{NL_a} a\#t$.
2. If $\vdash_{NL_a} \nabla \sigma$ and $\sigma \rightarrow_{\mathcal{R}} \delta$ then $\vdash_{NL_a} \nabla \delta$.

**Proof.**

1. By Proposition 2 and Lemma 3(4).
2. By 1 and Lemma 5, if \( \vdash_{NL_\alpha} \nu \sigma \# e \sigma \) then \( \vdash_{NL_\alpha} \nu \sigma \# e \delta \). Hence, from \( \vdash_{NL_\alpha} \nabla \sigma \), we have \( \vdash_{NL_\alpha} \nabla \delta \). \( \square \)

We define the notions in Definition 9 for \( \rightarrow R \) as well. Then strong compatibility of \( \rightarrow R \) with \( \approx_\alpha \) can be extended to strong compatibility of \( \rightarrow \rightarrow R \) with \( \approx_\alpha \).

**Lemma 7 (Strong compatibility with \( \approx_\alpha \)).** Let \( R \) be a uniform NRS\(_{AS} \). If \( s' \approx_\alpha s \rightarrow \rightarrow R t \) then there exists \( t' \) such that \( s' \rightarrow \rightarrow R t' \approx_\alpha t \).

**Proof.** By induction on the derivation of \( s \rightarrow \rightarrow R t \). If the last applied rule in the derivation is \( (B) \), then the claim follows by Lemma 2. Among the other cases, we treat the case where \( G = [a] \square \). Then the last part of the derivation has the form

\[
\frac{s_1 \rightarrow \rightarrow R t_1}{[a]s_1 \rightarrow \rightarrow R [a]t_1} \ (C)
\]

where \( [a]s_1 = s \) and \( [a]t_1 = t \). Now we have two cases.

(a) \( s' = [a]s'_1 \) and \( \vdash_{NL_\alpha} [a]s'_1 \approx_\alpha [a]s_1 \). Then \( \vdash_{NL_\alpha} s'_1 \approx_\alpha s_1 \), and so by the induction hypothesis, there exists \( t'_1 \) such that \( s'_1 \rightarrow \rightarrow R t'_1 \approx_\alpha t_1 \). Hence we have \( [a]s'_1 \rightarrow \rightarrow R [a]t'_1 \approx_\alpha [a]t_1 \).

(b) \( s' = [b]s'_1 \) and \( \vdash_{NL_\alpha} [b]s'_1 \approx_\alpha [a]s_1 \). Then \( \vdash_{NL_\alpha} (b a) \cdot s'_1 \approx_\alpha s_1 \) and \( \vdash_{NL_\alpha} a \cdot s'_1 \). So by the induction hypothesis, there exists \( t'_1 \) such that \( (b a) \cdot s'_1 \rightarrow \rightarrow R t'_1 \approx_\alpha t_1 \). By taking \( \pi = (a b) \) in Lemma 4, we have \( s'_1 \rightarrow \rightarrow R (a b) \cdot t'_1 \), and by Lemma 6, we have \( \vdash_{NL_\alpha} a \cdot (a b) \cdot t'_1 \). Hence, we obtain the following derivations, from which the claim follows.

\[
\frac{s'_1 \rightarrow \rightarrow R (a b) \cdot t'_1}{[b]s'_1 \rightarrow \rightarrow R [b] (a b) \cdot t'_1} \ (C) \quad \text{and} \quad \frac{\vdash_{NL_\alpha} t'_1 \approx_\alpha t_1}{\vdash_{NL_\alpha} [a]t'_1 \approx_\alpha [a]t_1}
\]

The cases where \( G \neq [a] \square \) are simpler. \( \square \)

### 3.3 Proofs of commutation and confluence

A key lemma to our theorems of commutation and confluence is Lemma 9. It says that for two mutually orthogonal NRS\(_{AS} \)'s \( R_1 \) and \( R_2 \), if parallel reduction of \( R_1 \) takes place from a redex of \( R_2 \) then all the reductions are below variable positions of the rule of \( R_2 \). This property is used in the proof of subcommutation lemma (Lemma 10).

First we show an auxiliary lemma to address the separated case of moderated variables.

**Lemma 8.** Let \( R \) be a uniform NRS\(_{AS} \). If \( \vdash_{NL_\alpha} s \approx_\alpha (\pi \cdot X) \sigma \) and \( s \rightarrow \rightarrow R t \) then there exists \( \delta \) such that \( \vdash_{NL_\alpha} t \approx_\alpha (\pi \cdot X) \delta \), \( \sigma \rightarrow \rightarrow R \delta \), and \( \forall \gamma \in \text{dom}(\sigma X) \setminus \{X\} \). \( Y \sigma = Y \delta \).
Proof. From $\vdash_{NL_a} s \approx_\alpha (\pi X)\sigma = \pi \sigma \cdot (X \sigma)$, we have $\vdash_{NL_a} (\pi \sigma)^{-1} \cdot s \approx_\alpha X \sigma$, and from $s \rightarrow_R t$, we have $(\pi \sigma)^{-1} \cdot s \rightarrow_R (\pi \sigma)^{-1} \cdot t$ by Lemma 4. Hence by Lemma 7, there exists $t'$ such that $X \sigma \rightarrow_R t' \approx_\alpha (\pi \sigma)^{-1} \cdot t$. Now we define $\delta$ by $X \delta = t'$ and $Y \delta = Y \sigma$ for $Y \in \text{dom}(\sigma_X) \setminus \{X\}$ (and $A \delta = A \sigma$ for $A \in \text{dom}(\sigma_X)$). Then we have $s \rightarrow_R \delta$, and from $X \delta = t' \approx_\alpha (\pi \sigma)^{-1} \cdot t = (\pi \delta)^{-1} \cdot t$, we obtain $t \approx_\alpha (\pi \delta \cdot (X \delta) = (\pi \cdot X) \delta$.

Now we prove the announced lemma. Note that the linearity condition is supposed for variables, but not for atom-variables. We therefore restrict induction to proper subexpressions of the left-hand side of the rule of $R_2$, rather than all linear expressions, so that the substitution for atom-variables can be fixed. This is a different point from the proof of Lemma 11 of [10] which uses induction on all linear terms.

Note also that there is not an atom but an atom expression (a suspension on an atom-variable) at each place of binder in a rewrite rule. So we have to be careful in the cases of abstractions (cases 2(a) and 2(b)) in the following proof.

Lemma 9. Let $R_1$ and $R_2$ be mutually orthogonal uniform NRS$_{AS}$’s, and let $\nabla \vdash l \rightarrow r \in R_2$. Suppose that $\sigma$ is a ground substitution with $\text{Var}_X, X_\lambda(l) \subseteq \text{dom}(\sigma)$ and $\vdash_{NL_a} \nabla \sigma$. Then, for every proper subexpression $l'$ of $l$, if $\vdash_{NL_a} s \approx_\alpha l' \sigma$ and $s \rightarrow_{R_1} t$ then there exists $\delta$ such that $\vdash_{NL_a} t \approx_\alpha l' \delta$, $\sigma \rightarrow_{R_1} \delta$, and $\forall X \in \text{dom}(\sigma_X) \setminus \text{Var}_X(l')$. $X \sigma = X \delta$.

Proof. By induction on the structure of $l'$. The case where $l'$ is a moderated variable $\pi X$ follows from Lemma 8. For the other cases, we first show that the last rule used in the derivation of $s \rightarrow_{R_1} t$ cannot be (B). Suppose otherwise. Then by the definition of rewrite relation, we have $\vdash_{NL_a} \nabla \delta$ and $\vdash_{NL_a} s \approx_\alpha \hat{l} \sigma$ for some $\nabla \vdash \hat{l} \rightarrow \hat{r} \in R_1$ and $\hat{\sigma}$, where we assume without loss of generality that $\text{dom}(\hat{\sigma}) \cap \text{dom}(\sigma) = \emptyset$. However, then the VANUP ($\{l \approx l'\}, \nabla \cup \nabla$) has a solution $\hat{\sigma} \cup \sigma$, which means that $\nabla \vdash \hat{l} \rightarrow \hat{r}$ overlaps on $\nabla \vdash l \rightarrow r$, contradicting the mutual orthogonality of $R_1$ and $R_2$. Hence, the last rule used in the derivation of $s \rightarrow_{R_2} t$ must be (C). The rest of the proof is by case analysis according to the form of $l'$.

Here we consider the cases where $l' = (l'_1, \ldots, l'_n)$ and $l' = [v][l'_1]$.

1. $l' = (l'_1, \ldots, l'_n)$. Then the last part of the derivation of $s \rightarrow_{R_1} t$ has the form

$$\frac{s_1 \rightarrow_{R_1} t_1 \ldots s_n \rightarrow_{R_1} t_n}{\langle s_1, \ldots, s_n \rangle \rightarrow_{R_1} \langle t_1, \ldots, t_n \rangle} \quad (C)$$

and for each $i \in \{1, \ldots, n\}, \vdash_{NL_a} s_i \approx_\alpha l'_i \sigma$. By the induction hypothesis, there exist $\delta_i$’s such that $\vdash_{NL_a} t_i \approx_\alpha l'_i \delta_i$, $\sigma \rightarrow_{R_1} \delta_i$, and $\forall X \notin \text{Var}_X(l'_i)$. $X \sigma = X \delta_i$. Since $l'$ is linear, we can take $\delta$ such that if $X \in \text{Var}_X(l'_i)$ then $X \delta = X \delta_i$, and if $X \in \text{dom}(\sigma_X) \setminus \text{Var}_X(l')$ then $X \delta = X \sigma$ (and $A \delta = A \sigma$ for $A \in \text{dom}(\sigma_X)$). It is easy to check that this $\delta$ satisfies the required condition.

2. $l' = [v][l'_1]$. Since $\vdash_{NL_a} s \approx_\alpha l' \sigma = ([v][l'_1]) \sigma = [v \sigma][l'_1 \sigma]$, we have two cases. Let $a = v \sigma$. 12
(a) \( s = [a]s' \). Then \( \vdash_{\text{NL}_{\alpha}} s' \approx_{\alpha} l'_1\sigma \), and the last part of the derivation of \( s \rightarrow_{\mathcal{R}_1} t \) has the form

\[
\frac{s' \rightarrow_{\mathcal{R}_1} t'}{[a]s' \rightarrow_{\mathcal{R}_1} [a]t'} \quad (C)
\]

Then by the induction hypothesis, there exists \( \delta \) such that \( \vdash_{\text{NL}_{\alpha}} t' \approx_{\alpha} l'_1\delta \), \( \sigma \rightarrow_{\mathcal{R}_1} \delta \), and \( \forall X \notin \text{Var}(l'_1). X\sigma = X\delta \). From \( \vdash_{\text{NL}_{\alpha}} t' \approx_{\alpha} l'_1\delta \), we have \( \vdash_{\text{NL}_{\alpha}} [a][l'_1\delta] = [v\sigma][l'_1\delta] = [v\delta][l'_1\delta] = ([v]l'_1)\delta \). Hence, the claim follows.

(b) \( s = [b]s' \). Then, \( \vdash_{\text{NL}_{\alpha}} (b\alpha)\cdot s' \approx_{\alpha} l'_1\sigma \), \( \vdash_{\text{NL}_{\alpha}} a\#s' \) and the last part of the derivation of \( s \rightarrow_{\mathcal{R}_1} t \) has the form

\[
\frac{s' \rightarrow_{\mathcal{R}_1} t'}{[b]s' \rightarrow_{\mathcal{R}_1} [b]t'} \quad (C)
\]

From \( s' \rightarrow_{\mathcal{R}_1} t' \), we have \( (b\alpha)\cdot s' \rightarrow_{\mathcal{R}_1} (b\alpha)\cdot t' \) by Lemma 4. Since \( \mathcal{R}_1 \) is uniform, we also have \( \vdash_{\text{NL}_{\alpha}} a\#t' \) by Lemma 6. Now, by the induction hypothesis for \( l'_1 \), there exists \( \delta \) such that \( \vdash_{\text{NL}_{\alpha}} (b\alpha)\cdot t' \approx_{\alpha} l'_1\delta \), \( \sigma \rightarrow_{\mathcal{R}_1} \delta \), and \( \forall X \notin \text{Var}(l'_1). X\sigma = X\delta \). Then from \( \vdash_{\text{NL}_{\alpha}} (b\alpha)\cdot t' \approx_{\alpha} l'_1\delta \) and \( \vdash_{\text{NL}_{\alpha}} a\#t' \), we have \( \vdash_{\text{NL}_{\alpha}} [b][t'] \approx_{\alpha} [a](l'_1\delta) = [v\sigma](l'_1\delta) = [v\delta](l'_1\delta) = ([v]l'_1)\delta \). Hence, the claim follows.

Now we show subcommutation modulo \( \approx_{\alpha} \) of mutually orthogonal uniform NRS_{AS}'s.

**Lemma 10 (Subcommutation modulo \( \approx_{\alpha} \)).** Let \( \mathcal{R} \) and \( \mathcal{R}' \) be mutually orthogonal uniform NRS_{AS}'s. If \( s \rightarrow_{\mathcal{R}} t \) and \( s \rightarrow_{\mathcal{R}'} t' \) then there exist \( u \) and \( u' \) such that \( t \rightarrow_{\mathcal{R}'} u \), \( t' \rightarrow_{\mathcal{R}} u' \) and \( \vdash_{\text{NL}_{\alpha}} u \approx_{\alpha} u' \).

**Proof.** By induction on the structure of \( s \). We distinguish cases according to the last rules used in the derivations of \( s \rightarrow_{\mathcal{R}} t \) and \( s \rightarrow_{\mathcal{R}} t' \).

1. Both rules are (B). This case contradicts the mutual orthogonality of \( \mathcal{R} \) and \( \mathcal{R}' \).
2. Both rules are (C). The claim follows from the induction hypothesis.
3. One is (C) and the other is (B). Suppose that \( s \rightarrow_{\mathcal{R}} t \) is derived by (C) and that \( s \rightarrow_{\mathcal{R}'} t' \) is derived by (B). Then there exist \( \mathcal{R}' = \nabla \vdash l \rightarrow r \in \mathcal{R}' \) and \( \sigma \) such that \( s \rightarrow_{(\mathcal{R},\mathcal{R}',\sigma)} t' \). By the definition of rewrite relation, we have \( \vdash_{\text{NL}_{\alpha}} \nabla\sigma, \vdash_{\text{NL}_{\alpha}} s \approx_{\alpha} l\sigma \) and \( t' = r\sigma \).

Here we only consider the case where \( s = f \cdot s_1 \). Then the last part of the derivation of \( s \rightarrow_{\mathcal{R}} t \) has the form

\[
\frac{s_1 \rightarrow_{\mathcal{R}} l_1}{f \cdot s_1 \rightarrow_{\mathcal{R}} f \cdot l_1} \quad (C)
\]

Since \( \vdash_{\text{NL}_{\alpha}} s \approx_{\alpha} l\sigma \), we have \( \vdash_{\text{NL}_{\alpha}} f \cdot s_1 \approx_{\alpha} l\sigma \). So \( l \) is of the form \( f \cdot l_1 \) and \( \vdash_{\text{NL}_{\alpha}} s_1 \approx_{\alpha} l_1\sigma \). Hence by Lemma 9 with \( l_1 \) as \( l \) and \( t_1 \) as \( t \), there
exists $\delta$ such that $\vdash_{NLa} t_1 \approx_\alpha l_1 \delta$ and $\sigma \rightarrow_R \delta$. From the former, we have $\vdash_{NLa} f t_1 \approx_\alpha f l_1 \delta = l \delta$. From the latter and $\vdash_{NLa} \nabla \sigma$, we have $\vdash_{NLa} \nabla \delta$ by Lemma 6(2). Thus $t = f t_1 \rightarrow_{(R, s, \delta)} r \delta$, and so $t \rightarrow_R r \delta$ by the rule (B). On the other hand, by Lemma 5, we have $t' = r \sigma \rightarrow_R r \delta$. Hence, we can take $u = u' = r \delta$.

Actually, the above lemma also holds with the stronger condition $u = u'$. It does not necessarily hold in the case of an orthogonal uniform NRS $R = R'$ as we see in Lemma 11 below.

Since $\rightarrow_R \subseteq \rightarrow_{R'} \subseteq \rightarrow_{R'}^{*}$ by Lemma 3, we have the following theorem.

**Theorem 1 (Commutation modulo $\approx_\alpha$).** Let $R$ and $R'$ be mutually orthogonal uniform NRS $\alpha$’s. If $s \rightarrow_R t$ and $s \rightarrow_{R'} t'$ then there exist $u$ and $u'$ such that $t \rightarrow_{R'} u$, $t' \rightarrow_{R'} u'$ and $\vdash_{NLa} u \approx_\alpha u'$.

Next we show Church-Rosser modulo $\approx_\alpha$ for orthogonal NRS $\alpha$’s. Observing the proof of Lemma 9, we see that the claim also holds for any orthogonal NRS $\alpha$’s $R_1 = R_2$ since $l'$ is a proper subexpression of $l$ and there exists no proper overlap in an orthogonal NRS $\alpha$.

**Lemma 11.** Let $R$ be an orthogonal uniform NRS $\alpha$ that is $\alpha$-stable. Then, if $s \rightarrow_{R'} t$ and $s \rightarrow_{R'} t'$ then there exist $u$ and $u'$ such that $t \rightarrow_{R'} u$, $t' \rightarrow_{R'} u'$ and $\vdash_{NLa} u \approx_\alpha u'$.

**Proof.** We proceed in a similar way to the proof of Lemma 10 except that in the case where both rules are (B), they may be by the same rewrite rule. In that case, we use the $\alpha$-stability of $R$. □

**Theorem 2 (Church-Rosser modulo $\approx_\alpha$).** Let $R$ be an orthogonal uniform NRS $\alpha$ that is $\alpha$-stable. Then, $\rightarrow_R$ is Church-Rosser modulo $\approx_\alpha$.

**Proof.** By Lemma 7, $\rightarrow_R$ is strongly compatible with $\approx_\alpha$, and by Lemma 11, $\rightarrow_R$ subcommutes with $\rightarrow_R$ modulo $\approx_\alpha$. Hence by the results in [14] (see also Corollary 2.6.5 of [15]), $\rightarrow_R$ is Church-Rosser modulo $\approx_\alpha$. Since $\rightarrow_R \subseteq \rightarrow_{R'}$ by Lemma 3, it follows that $\rightarrow_R$ is Church-Rosser modulo $\approx_\alpha$. □
Example 8. The \( NRS_{AS} R_{\text{sub}} \) in Example 3 is orthogonal, uniform and \( \alpha \)-stable. Hence by Theorem 2, we see that \( \rightarrow_{R_{\text{sub}}} \) is Church-Rosser modulo \( \approx_{\alpha} \).

Example 9. Consider the \( NRS_{AS} R_{\text{pnfcom}} \) with the following rewrite rules:

\[
\begin{align*}
A \# Y \vdash A[\forall X,Y] & \rightarrow A[\forall X,Y] \quad (\forall_1) \\
A \# X \vdash X, A[\forall X,Y] & \rightarrow A[\forall X,Y] \quad (\forall_2) \\
A \# B \vdash B[\forall Y] & \rightarrow B[\forall Y] \quad (\text{COM}_\forall)
\end{align*}
\]

Then \( R_{\text{pnfcom}} \) has proper overlaps, and hence it is not orthogonal. So we cannot apply Theorem 2. However, it can be shown that \( \rightarrow_{R_{\text{pnfcom}}} \) is Church-Rosser modulo \( \approx_{\alpha} \) (cf. Example 34 of [21]).

4 Conclusion and related work

We have presented proofs of commutation modulo \( \approx_{\alpha} \) of mutually orthogonal uniform \( NRS_{AS} \)'s and Church-Rosser modulo \( \approx_{\alpha} \) for orthogonal uniform \( NRS_{AS} \)'s that are \( \alpha \)-stable. These are about the rewrite relations on ground terms, but they are enough for application to correctness of a form of program transformation. (Commutation of mutually orthogonal systems is used, e.g. in the proof of Lemma 4.3(3) of [9].) In the presence of atom-variables, which are not necessarily linear in the left-hand side of a rewrite rule, and atom expressions (suspensions on atom-variables), the proofs include delicate arguments.

Currently, we are working on implementation of a confluence checking tool that verifies sufficient conditions for confluence and commutation in this paper. To verify non-existence of overlaps in \( NRS_{AS} \)'s, it should use some procedure for deciding variable-atom nominal unification problems. Since our previous tool [1] implements an equivariant unification procedure which uses atom-variables and permutation-variables, it is expected to help us to implement such a procedure.

The difference between confluence on ground nominal terms like in this paper and confluence in previous work on traditional nominal rewriting corresponds to the difference between confluence of some concrete calculus and meta-confluence of that calculus with meta-variables (see, e.g. [13]). The traditional nominal rewriting system corresponding to \( R_{\text{pnfcom}} \) in Example 9 above is an example of a system for which confluence on ground terms holds but confluence on general terms does not (cf. Examples 12 and 34 of [21]).

As a rewriting mechanism with matching and overlaps without involving permutations, closed rewriting has been considered [2, 4]. However, closed rewriting is incompatible with rewrite rules with free atoms like the rule (\( \text{sub}_{\text{var}} \)) in Example 3. In fact, according to Lemma 5.3 of [2], no rewrite step is induced by such rewrite rules.

Acknowledgements. We are grateful to the anonymous referees for valuable comments. The first author thanks Makoto Hamana for useful discussions. This work was partly supported by JSPS KAKENHI Grant Numbers JP17K00005, JP18K1158, JP19K11891 and JP20H04164.
References


Pattern eliminating transformations

Horatiu Cirstea\textsuperscript{1,2}, Pierre Lermusiaux\textsuperscript{1,2}, and Pierre-Etienne Moreau\textsuperscript{1,2}

\textsuperscript{1} Université de Lorraine
\textsuperscript{2} LORIA \{name\}.\{surname\}@loria.fr

Abstract. Program transformation is a common practice in computer science, and its many applications can have a range of different objectives. For example, a program written in an original high level language could be either translated into machine code for execution purposes, or towards a language suitable for formal verification. Such compilations are split into several so-called passes which generally aim at eliminating certain constructions of the original language to get a program in some intermediate languages and finally generate the target code. Rewriting is a widely established formalism to describe the mechanism and the logic behind such transformations. In a typed context, the underlying type system can be used to give syntactic guarantees on the shape of the results obtained after each pass, but this approach could lead to an accumulation of auxiliary types that should be considered. We propose in this paper a less intrusive approach based on simply annotating the function symbols with the (anti-)patterns the corresponding transformations are supposed to eliminate. We show how this approach allows one to statically check that the rewrite system implementing the transformation is consistent with the annotations and thus, that it eliminates the respective patterns.

Keywords: Rewriting, Pattern-matching, Pattern semantics, Compilation

1 Introduction

Rewriting is a well established formalism widely used in both computer science and mathematics. It has been used, for example, in semantics in order to describe the meaning of programming languages \cite{29}, but also in automated reasoning when describing, by inference rules, a logic, a theorem prover \cite{20}, or a constraint solver \cite{19}. Rewriting has turned out to be particularly well adapted to describe program semantics \cite{29} and program transformations \cite{25,27}. There are several languages and tools implementing the notions of pattern matching and rewriting rules ranging from functional languages, featuring relatively simple patterns and fixed rewriting strategies, to rule based languages like Maude \cite{10}, Stratego \cite{32}, or Tom \cite{5}, providing equational matching and flexible strategies; they have been all used as underlying languages for more or less sophisticated compilers.

In the context of compilation, the complete transformation is usually performed in multiple phases, also called passes, in order to eventually obtain a
program in a different target language. Most of these passes concern transforma-
tions between some intermediate languages and often aim at eliminating certain
constructions of the original language. These transformations could eliminate
just some symbols, like in desugaring passes for example, or more elaborate
constructions, like in code optimization passes.

To guarantee the correctness of the transformations we could of course use
runtime assertions but static guarantees are certainly preferable. When using
typed languages, the types can be used to guarantee the correctness of some
of the constraints on the target language. In this case, the type of the function
implicitly expresses the expected result of the transformation. The differences
between the source and the target language concern generally only a small per-
centage of the symbols, and the definition of the target language is often tedious
and contains a lot of the symbols from the source type. For example, for a pass
performing desugaring we would have to define a target language using the same
symbols as the source one but the syntactic sugar symbols.

Formalisms such as the one proposed for NanoPass [21] have proposed a
method to eliminate a lot of the overhead induced by the definition of the in-
termediate languages by specifying only the symbols eliminated from the source
language and generating automatically the corresponding intermediate language.

For instance, let us consider expressions which are build out of (wrapped)
integers, (wrapped) strings and lists:

\[
Expr = \text{int(Int)} \\
| \text{str(String)} \\
| \text{lst(List)} \\
\]

\[
List = \text{nil()} \\
| \text{cons(Expr,List)}
\]

If, for some reason, we want to define a transformation encoding integers by
strings then, the target language in NanoPass would be \(Expr^-\text{int}\), i.e. expres-
sions build out of strings and lists. Note that in this case the tool (automatically)
removes the symbol \text{int} from \(Expr\) and replaces accordingly \(Expr\) with the new
type in the type of \text{cons}.

This kind of approaches reach their limitations when the transformation of
the source language goes beyond the removal of some symbols. For example, if we
want to define a transformation which flattens the list expressions and ensures
thus that there is no nested list, the following target type should be considered:

\[
Expr = \text{lit(Literal)} \\
| \text{lst(List)} \\
\]

\[
Literal = \text{int(Int)} \\
| \text{str(String)} \\
| \text{cons(Literal,List)}
\]

Functional approaches to transformation [28] relying on the use of fine grained
type systems which combine overloading, subtyping and polymorphism through
the use of variants [13] can be used to define the transformation and perform
(implicitly) such verifications. While effective, this method requires to design
such adjusted types in a case by case basis.

We propose in this paper a formalism where function symbols are simply
annotated with the patterns that should be eliminated by the corresponding
transformation and a mechanism to statically verify that the rewriting system
implementing the function eliminates indeed these patterns. The method is min-
imally intrusive: for the above example, we should just annotate the flattening
function symbol with the (anti-)pattern \( \text{cons}(\text{lst}(l_1), l_2) \) and the checker (implemented in Haskell) verifies that the underlying rewriting system is consistent with the annotation, or exhibits the problematic rule(s) and issue(s) if it is not. The method applies to constructor based term rewriting systems which correspond to functional programs where functions are defined by pattern matching, programs which are very common and often used when defining transformations.

First, in the next section, we introduce the basic notions and notations used in the article. We introduce then, in Section 3, the notion of pattern-free terms together with their ground semantics and we state the pattern-free properties a rewriting system should satisfy to be consistent with the pattern annotations. Section 4 describes a method for automatically checking pattern-free properties relying on the deep semantics, an extension of the ground semantics, and shows how this method can be used to verify that a rewriting system is consistent with the pattern annotations and thus, that specific patterns are absent from the result of the corresponding transformation. We finally present some related work and conclude. All proofs are available in the appendix.

2 Preliminary notions

We define in this section the basic notions and notations used in this paper; more details can be found in [31].

A many-sorted signature \( \Sigma = (\mathcal{S}, \mathcal{F}) \), consists of a set of sorts \( \mathcal{S} \) and a set of symbols \( \mathcal{F} \). The set of symbols is partitioned into two disjoint sets \( \mathcal{F} = \mathcal{D} \cup \mathcal{C} \); \( \mathcal{D} \) is the set of defined symbols and \( \mathcal{C} \) the set of constructors. A symbol \( f \) with domain \( \text{Dom}(f) = s_1 \times \cdots \times s_n \in \mathcal{S}^* \) and co-domain \( \text{CoDom}(f) = s \in \mathcal{S} \) is written \( f : s_1 \times \cdots \times s_n \rightarrow s \); we may write \( f_s \) to indicate explicitly the co-domain. We denote by \( \mathcal{C}_s \), resp. \( \mathcal{D}_s \), the set of constructors, resp. defined symbols, with co-domain \( s \). Variables are also sorted and we write \( x : s \) or \( x_s \) to indicate that variable \( x \) has sort \( s \). The set \( \mathcal{X}_s \) denotes a set of variables of sort \( s \) and \( \mathcal{X} = \bigcup_{s \in \mathcal{S}} \mathcal{X}_s \) is the set of sorted variables.

The set of terms of sort \( s \in \mathcal{S} \), denoted \( \mathcal{T}_s(\mathcal{F}, \mathcal{X}) \) is the smallest set containing \( \mathcal{X}_s \) and such that \( f(t_1, \ldots, t_n) \) is in \( \mathcal{T}_s(\mathcal{F}, \mathcal{X}) \) whenever \( f : s_1 \times \cdots \times s_n \rightarrow s \) and \( t_i \in \mathcal{T}_{s_i}(\mathcal{F}, \mathcal{X}) \), \( i \in [1, n] \). We write \( t : s \) to indicate that the term \( t \) is of sort \( s \), i.e. when \( t \in \mathcal{T}_s(\mathcal{F}, \mathcal{X}) \). The set of sorted terms is defined as \( \mathcal{T}(\mathcal{F}, \mathcal{X}) = \bigcup_{s \in \mathcal{S}} \mathcal{T}_s(\mathcal{F}, \mathcal{X}) \). The set of variables occurring in \( t \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \) is denoted by \( \text{Var}(t) \). If \( \text{Var}(t) \) is empty, \( t \) is called a ground term. \( \mathcal{T}(\mathcal{F}) \) denotes the set of all ground terms of sort \( s \) and \( \mathcal{T}(\mathcal{F}) \) denotes the set of all ground terms. Terms in \( \mathcal{T}(\mathcal{C}) \) are called values. A linear term is a term where every variable occurs at most once. The linear terms in \( \mathcal{T}(\mathcal{C}, \mathcal{X}) \) are called constructor patterns or simply patterns.

A position of a term \( t \) is a sequence of positive integers describing the path from the root of \( t \) to the root of the subterm at that position. The empty sequence representing the root position is denoted by \( \varepsilon \). \( t|_\omega \), resp. \( t(\omega) \), denotes the subterm of \( t \), resp. the symbol of \( t \), at position \( \omega \). \( t|_\omega [s]_\omega \) denotes the term \( t \) with the subterm at position \( \omega \) replaced by \( s \). \( \text{Pos}(t) \) denotes the set of positions of \( t \).
We consider that every defined symbol \( f \in \mathcal{D} \) is now annotated with a pattern \( p \in \mathcal{T}_\mathcal{D}(\mathcal{C}, \mathcal{X}) \) and we use this notation to define pattern-free terms.

3.1 Pattern-free terms

We want to ensure that the normal form of a term, if it exists, does not contain a specific constructor and more generally that no subterm of this normal form matches a given pattern. The sort of the term provides some information on the shape of the normal forms since the precise language of the values of a given sort is implicitly given by the signature. Sometimes the normal forms satisfy constraints stronger than those induced from the sorts but these constraints cannot always be determined statically only from the sorts but also depend on the underlying CBTRS.

To guarantee these constraints we annotate all defined symbols with the patterns that are supposed to be absent when reducing a term headed by the respective symbol and we check that the CBTRS defining the corresponding functions are consistent with these annotations.

We focus first on the notion of pattern-free term and on the corresponding ground semantics, and explain in the next sections how one can check pattern-freeness and verify the consistency of the symbol annotations with a CBTRS.
We can compute \( f^{-p}(t_1, \ldots , t_n) \) contains no subterms matched by \( p \); in particular, if the term is eventually reduced to a value then this value contains no subterms matched by \( p \). Given the example from the introduction, we can consider two function symbols, \( flattenE^{-p} : Expr \mapsto Expr \) and \( flattenL^{-p} : List \mapsto List \), with \( p = \text{cons}(\text{lst}(11), l2) \), to indicate that the normal forms of any term headed by one of these symbols contain no nested lists. The annotation of the function symbol for the concatenation, \( \text{concat}^{-1} : List \times List \mapsto List \), indicates that no particular shape is expected for the reducts of the corresponding terms.

**Definition 3.1 (Pattern-free terms).** Given \( p \), a constructor pattern or \( \bot \),

- a value \( v \in \mathcal{T}(\mathcal{C}) \) is \( p \)-free iff \( \forall \omega \in \mathcal{Pos}(v) \), \( p \not\in \omega \);
- a pattern \( t \in \mathcal{T}(\mathcal{C}, \mathcal{X}) \) is \( p \)-free iff \( \forall \sigma \) such that \( \sigma(t) \in \mathcal{T}(\mathcal{C}) \), \( \sigma(t) \) is \( p \)-free;
- a term \( t \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \) is \( p \)-free iff, \( \forall \omega \in \mathcal{Pos}(t) \) such that \( t(\omega) = f_{\omega}^{-q'} \in \mathcal{F} \), \( t[\nu]_\omega \) is \( p \)-free for all \( q \)-free value \( v \in \mathcal{T}(\mathcal{C}) \).

A value is \( p \)-free if and only if \( p \) matches no subterm of the value. For constructor patterns, verifying a pattern-free property comes to verifying the property for all the ground instances of the term. Finally, a general term is \( p \)-free if and only if replacing (all) the subterms headed by a defined symbol \( f_{\omega}^{-q'} \) by any \( q \)-free value of the same sort \( s \) results in a \( p \)-free term. Intuitively, this corresponds to considering an over-approximation of the set of normal forms of an annotated term. While pattern-free properties can be checked for any value by exploring all its subterms, this is not possible for a general term since the property has to be verified by a potentially infinite number of values. We present in Section 4 an approach for solving this problem.

### 3.2 Generalized ground semantics

The notion of ground semantics presented in Section 2 and, in particular, the approach proposed in 9 to compute differences (and thus intersections) of such semantics, can be used to compare the shape of two constructor patterns \( p, q \) (at the root position). More precisely, when \( \llbracket p \rrbracket \cap \llbracket q \rrbracket = \emptyset \) we have that \( \forall \sigma, \sigma(q) \not\in \llbracket p \rrbracket \) and therefore, we can establish that \( \forall \sigma, p \not\in \sigma(q) \). We can thus compare the semantics of a given pattern \( p \) with the semantics of each of the subterms of a constructor pattern \( t \) in order to check that \( t \) is \( p \)-free.

**Example 3.1.** Consider the signature \( \Sigma \) with \( S = \{s_1, s_2, s_3\} \) and \( F = \mathcal{C} = \{c_1 : s_2 \times s_1 \mapsto s_1, c_2 : s_3 \mapsto s_1, c_3 : s_1 \mapsto s_2, c_4 : s_3 \mapsto s_2, c_5 : s_3 \mapsto s_3, c_6 : s_3 \mapsto s_3\} \).

We can compute \( \llbracket c_1(c_4(c_6), y_{s_3}) \rrbracket \cap \llbracket c_1(x_{s_2}, c_2(c_6)) \rrbracket = \llbracket c_1(c_4(c_6), c_2(c_6)) \rrbracket \) and thus neither \( c_1(c_4(c_6), y_{s_3}) \) is \( c_1(x, c_2(c_6)) \)-free nor \( c_1(x_{s_2}, c_2(c_6)) \) is \( c_1(c_4(c_6), y) \)-free.

Similarly, we can check that \( \llbracket c_3(c_2(z_{s_3}) \rrbracket \cap \llbracket c_4(z_{s_3}) \rrbracket = \emptyset \) and that \( \llbracket c_2(z_{s_3}) \rrbracket \cap \llbracket c_4(z_{s_3}) \rrbracket = \emptyset \) and, as a term of sort \( s_3 \) can only contain constructors \( c_5 \) and \( c_6 \), we can deduce that \( c_3(c_2(z_{s_3})) \) is \( c_3(z) \)-free.

We want to establish a general method to verify pattern-free properties for any term and we propose an approach which relies on the notion of ground semantics extended in order to take into account all terms in \( \mathcal{T}(\mathcal{F}, \mathcal{X}) \):
Definition 3.2 (Generalized ground semantics). Given a sort $s \in S$, a pattern $p \in T_\perp(C, X)$, a symbol $f_s^{-p}: s_1 \times \cdots \times s_n \mapsto s \in D_s$, a term $u \in T(C, X)$, and a term $t \in T_s(F, X)$, with $t_i \in T_{s_i}(F, X)$, $i \in [1, n]$, we can use a recursive definition for the non-variable patterns:

- $[u] = \{ \sigma(u) \mid \sigma(u) \in T(C) \}$
- $\left[t \left[f_s^{-p}(t_1, \ldots, t_n)\right]_u\right] = \{ u \in [t[v]]_u \mid \forall v \in T_s(C) \text{ p-free} \}$

Note that the ground semantics of a variable $x_s$ is the set of all possible ground patterns of the corresponding sort: $[x_s] = T_s(C)$, and for patterns, since they are linear, we can use a recursive definition for the non-variable patterns: $[c(p_1, \ldots, p_n)] = \{ c(v_1, \ldots, v_n) \mid (v_1, \ldots, v_n) \in [p_1] \times \cdots \times [p_n] \}$ for all $c \in C$.

Moreover, by definition we have $[[f_s^{-p}(t_1, \ldots, t_n)]] = \{ v \in T_s(C) \mid v \text{ p-free} \}$.

The generalized ground semantics of a term rooted by a defined symbol represents an over-approximation of all the possible values obtained by reducing the term with respect to a CBTRS preserving the pattern-free properties.

Pattern-freeness can be checked by exploring the semantics of the term:

Proposition 3.1. Let $t \in T(F, X), p \in T_\perp(C, X)$, $t$ is p-free iff $\forall v \in [t], v$ is p-free.

For convenience, we consider also annotated variables whose semantics is that of any term headed by a defined symbol with the same co-domain as the sort of the variable:

$[x_s^{-p}] = \{ v \in T_s(C) \mid v \text{ p-free} \}$

Thus, $[[f_s^{-p}(t_1, \ldots, t_n)]] = [x_s^{-p}]$ for all $f_s^{-p} \in D_s$. Note that $x_s^{-p}$ has the same semantics as $x_s$. We denote by $X^a$ the set of annotated variables.

Given a linear term $t \in T(F, X)$, we can systematically construct its symbolic equivalent $\tilde{t} \in T(C, X^a)$ by replacing all the subterms of $t$ headed by a defined symbol $f_s^{-p}$ by a fresh variable $x_s^{-p}$ of the corresponding sort and annotated by the same pattern:

Proposition 3.2. $\forall \tilde{t} \in T(F, X), [\tilde{t}] = [[\tilde{t}]]$

Example 3.2. We consider the signature from Example 3.1 enriched with the defined symbols $D = \{ f_1^{-p1} : s_1 \mapsto s_1, g^{-p2} : s_2 \mapsto s_2 \}$ with $p_1 = c_1(x, y)$ and $p_2 = c_4(z)$. If we consider the term $r_1 = c_1(g^{-p2}(x), f_1^{-p1}(y))$, to construct its symbolic equivalent, we replace $f_1^{-p1}(y)$ and $g^{-p2}(x)$ by $y_{s_1}^{-p1}$ and $x_{s_2}^{-p2}$, respectively. Thus we have $r_1 = c_1(x_{s_2}^{-p2}, y_{s_1}^{-p1})$.

We can thus restrict in what follows to patterns using annotated variables and we consider extended patterns built out of this kind of patterns:

$p, q := x \mid (c(q_1, \ldots, q_n) \mid p_1 + p_2 \mid p_1 \setminus p_2 \mid p_1 \times p_2 \mid \perp)

$with $x \in X^a, p, p_1, p_2 : s$ for some $s \in S, c : s_1 \times \cdots \times s_n \mapsto s \in C$ and $\forall i \in [1, n], q_i : s_i$.

The pattern matching relation can be extended to take into account disjunctions, conjunctions and complements of patterns:

$p_1 + p_2 \not\iff v \iff p_1 \not\iff v \lor p_2 \not\iff v \quad p_1 \times p_2 \not\iff v \iff p_1 \not\iff v \land p_2 \not\iff v

p_1 \setminus p_2 \not\iff v \iff p_1 \not\iff v \land p_2 \not\iff v \quad \perp \not\iff v$
Intuitively, a pattern $p_1 + p_2$ matches any term matched by one of its components while a pattern $p_1 \times p_2$ matches any term matched by both its components. The relative complement of $p_2$ w.r.t. $p_1$, $p_1 \setminus p_2$, matches all terms matched by $p_1$ but those matched by $p_2$. $\bot$ matches no term. $\times$ has a higher priority than $\setminus$ which has a higher priority than $+$.

The notion of ground semantics can be also adapted to handle such patterns:

$$
J[p_1 + p_2] = J[p_1] \cup J[p_2] \\
J[p_1 \times p_2] = J[p_1] \cap J[p_2] \\
J[\bot] = \emptyset
$$

Extended patterns $p_1 + p_2, p_1 \setminus p_2, p_1 \times p_2$ are linear if each of $p_1$ and $p_2$ is linear; this corresponds to the fact that $p_1$ and $p_2$ represent independent alternatives w.r.t. matching and semantics, and thus, that their variables are unrelated. For example, the patterns $c_3(c_2(x)) + c_4(x)$ and $c_3(c_2(x)) + c_4(y)$ both represent all values rooted by $c_3$ followed by $c_2$, or by $c_3$. We can remark that $p_1$ and $p_2$ in Example 3.2 are regular patterns, that $x\setminus p_1 s_2 \setminus p_2$ is a quasi-additive pattern, and that $\tilde{r}_1$ is a symbolic pattern (indeed, the symbolic equivalent of any term is a symbolic pattern).

### 3.3 Semantics preserving CBTRS

Generalized ground semantics rely on the symbol annotations and assume thus a specific shape for the normal forms of reducible terms. This assumption should be checked by verifying that the CBTRSs defining the annotated symbols are consistent with these annotations, i.e. check that the semantics is preserved by reduction.

**Definition 3.3 (Semantics preservation).** A rewrite rule $l \rightarrow r$ is semantics preserving iff $[r] \subseteq [l]$. A CBTRS is semantics preserving iff all its rewrite rules are.

Semantics preservation carries over to the induced rewriting relation:

**Proposition 3.3.** Given a semantics preserving CBTRS $\mathcal{R}$ we have

$$
\forall t, v \in T(\mathcal{F}), \text{ if } t \rightarrow_{\mathcal{R}} v, \text{ then } [v] \subseteq [t].
$$

As an immediate consequence we obtain the pattern-free preservation:

**Corollary 3.1.** Given a semantics preserving CBTRS $\mathcal{R}$ we have

$$
\forall t, v \in T(\mathcal{F}), p \in T(\mathcal{C}, \mathcal{X}), \text{ if } t \text{ is } p\text{-free and } t \rightarrow_{\mathcal{R}} v, \text{ then } v \text{ is } p\text{-free.}
$$

Note that the rules of a CBTRS are of the form $f^{-p}(l_1, \ldots, l_n) \rightarrow r$ and thus, as an immediate consequence of Definition 3.2 the semantics of the left-hand side of the rewrite rule is the set of all $p$-free values. Therefore, according to
Proposition 3.1, such a rule is semantics preserving if and only if its right-hand side \( r \) is \( p \)-free. We will see in the next section how pattern-freeness and thus, semantics preservation, can be statically checked.

Example 3.3. We consider the signature from Example 3.2 and the CBTRS:
\[
\begin{align*}
    f^{-p_1}(c_1(x,y)) & \rightarrow c_1(g^{-p_2}(x),f^{-p_1}(y)) & g^{-p_2}(c_4(z)) & \rightarrow c_3(c_2(z)) \\
    f^{-p_1}(c_2(z)) & \rightarrow c_2(z) & g^{-p_2}(c_3(y)) & \rightarrow c_3(f^{-p_1}(y))
\end{align*}
\]
We have seen in Example 3.1 that \( c_3(c_2(x)) \) is \( p_2 \)-free and we can thus conclude that the rule \( g(c_4(z)) \rightarrow c_3(c_2(z)) \) is semantics preserving. In order to verify in a systematic way the corresponding pattern-free properties of all right-hand sides and conclude that the CBTRS is semantics preserving, we introduce in the next section a method to statically check pattern-freeness.

4 Deep semantics for pattern-free properties

The ground semantics was used in [9] as a means to represent a potentially infinite number of instances of a term in a finite manner and can be employed to check that a pattern matches (or not) a term by computing the intersection between their semantics. For pattern-freeness, we should check not only that the term is not matched by the pattern but also that none of its subterms is matched by this pattern. We would thus need a notion of ground semantics closed by the subterm relation.

We introduce next an extended notion of ground semantics satisfying the above requirements, show how it can be expressed in terms of ground semantics, and provide a method for checking the emptiness of the intersection of such semantics and thus, assert pattern-free properties.

4.1 Deep semantics

The notion of deep semantics is introduced to provide more comprehensive information on the shape of the (sub)terms compared to the ground semantics which describes essentially the shape of the term at the root position.

Definition 4.1 (Deep semantics). Let \( t \) be an extended pattern, its deep semantics \( \{t\} \) is defined as follows:
\[
\{t\} = \{u_\omega \mid u \in \{t\}, \omega \in Pos(u)\}
\]

Note first that, similarly to the case of generalized ground semantics, it is obvious that we can always exhibit a symbolic pattern equivalent in terms of deep semantics to a given term, i.e. \( \forall t \in \mathcal{T}(\mathcal{F}, \mathcal{X}), \{t\} = \{t\} \); consequently, we can focus on the computation of the deep semantics of extended patterns. Following this observation and as an immediate consequence of the definition we have a necessary and sufficient condition with regards to pattern-free properties:

Proposition 4.1 (Pattern-free vs Deep Semantics). Let \( p \in \mathcal{T}(\mathcal{C}, \mathcal{X}), t \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \), \( t \) is \( p \)-free iff \( \{t\} \cap \{p\} = \emptyset \).
To check the emptiness of the above intersection we express the deep semantics of a term as a union of ground semantics and then check for each of them that the intersection with the semantics of the considered pattern is empty.

First, since the deep semantics is based on the generalized ground semantics, we can easily establish a similar recursive definition for constructor patterns:

**Proposition 4.2.** For any constructor symbol $c \in \mathcal{C}$ and extended patterns $t_1, \ldots, t_n$, such that $\text{Dom}(c) = s_1 \times \cdots \times s_n$ and $t_1 : s_1, \ldots, t_n : s_n$, we have:

- If $\forall i \in [1, n], [t_i] \neq \emptyset$, then $\{c(t_1, \ldots, t_n)\} = \{c(t_1)\} \cup \left( \bigcup_{i=1}^{n} [t_i] \right)$;
- If $\exists i \in [1, n], [t_i] = \emptyset$, then $\{c(t_1, \ldots, t_n)\} = \emptyset$.

If we apply the above equation for the non-empty case recursively we eventually have to compute the deep semantics of annotated variables. Given an annotated variable $x_s^{-p}$, we can employ the algorithm introduced in Figure 1 to compute $\text{getReachable}(s, p, \emptyset, \perp)$ and use the result to express the deep semantics of $\{x_s^{-p}\}$ as a union of ground semantics (see Proposition 13).

Intuitively, the algorithm uses the definition of the deep semantics of a variable $\{x_s^{-p}\} = \{u_\omega \mid u \in [x_s^{-p}], \omega \in Pos(u)\}$ and the observation that the ground semantics of an annotated variable can be also defined as:

$$[x_s^{-p}] = \bigcup_{c \in C_s} [c(x_s^{-1}, \ldots, x_s^{-n}) \setminus p]$$  \hspace{1cm} (1)

By distributing the complement pattern $p$ on the subterms, the algorithm builds a set $Q_c(p)$ of tuples $q = (q_1, \ldots, q_n)$ of patterns, with each $q_i$ being either $\perp$ or a subterm of $p$, such that

$$[c(x_s^{-1}, \ldots, x_s^{-n}) \setminus p] = \bigcup_{q \in Q_c(p)} [c(x_s^{-1} \setminus q_1, \ldots, x_s^{-n} \setminus q_n)]$$  \hspace{1cm} (2)

We have thus:

$$[x_s^{-p}] = \{u_\omega \mid u \in [x_s^{-p}], \omega \in Pos(u)\} = \bigcup_{c \in C_s, q \in Q_c(p)} \{u_\omega \mid u \in [c(x_s^{-1} \setminus q_1, \ldots, x_s^{-n} \setminus q_n)], \omega \in Pos(u)\}$$

$$= \bigcup_{c \in C_s, q \in Q_c(p)} \bigcup_{u_\omega \mid \omega \in Pos(u)} [c(x_s^{-1} \setminus q_1, \ldots, x_s^{-n} \setminus q_n)] \text{ (def. of deep semantics)}$$

$$= \bigcup_{c \in C_s, q \in Q'_c(p)} [c(x_s^{-1} \setminus q_1, \ldots, x_s^{-n} \setminus q_n)] \cup \bigcup_{c \in C_s, q \in Q'_c(p)} \bigcup_{i=1}^{n} [x_s^{-1} \setminus q_i]$$

with $Q'_c(p) \subseteq Q_c(p)$ s.t. $\forall q = (q_1, \ldots, q_n) \in Q'_c(p)$, $[x_s^{-1} \setminus q_i] \neq \emptyset, i \in [1, n]$.

Note that $x_s^{-p}$ is the same as $x_s^{-p} \setminus \perp$ and thus, in order to express the deep semantics of annotated variables as a union of ground semantics the algorithm computes a fixpoint for the equation

$$[x_s^{-p} \setminus r] = [x_s^{-p} \setminus r] \cup \bigcup_{c \in C_s, q \in Q'_c(p)} \bigcup_{i=1}^{n} [x_s^{-1} \setminus q_i]$$
Function `getReachable(s, p, S, r)`

- **s**: current sort,
- **p**: pattern annotation,
- **S**: set of couples `(s', p')` reached,
- **r**: induced pattern (sum of constructor patterns)

**Data:**
- `s`: current sort,
- `p`: pattern annotation,
- `S`: set of couples `(s', p')` reached,
- `r`: induced pattern (sum of constructor patterns)

**Result:** set of couples `(s', p')` reachable from `s \ p s \ r`

- If `p : s` then `r ← r + p`
- If `|s \ r| = ∅` then return ∅
- If `∃(s, r') ∈ S, [r'] = [r]` then return `S`
- `R ← S ∪ {(s, r)}`
- `reachable ← False`

for `c ∈ C_s`

- `Q_c ← {(⊥, ..., ⊥, ⊥)}` with `m = arity(c)`
- for `i = 1` to `n` with `r = ∑_i r_i`

  if `r_i(c) = c` then
    `tQ_c ← ∅`
    for `(q_1, ..., q_m) ∈ Q_c, k ∈ [1, m]`
    `tQ_c ← tQ_c ∪ {(q_1, ..., q_k + r_i k, ..., q_m)}`
  `Q_c ← tQ_c`

for `(q_1, ..., q_m) ∈ Q_c`

- `subRs ← []`
- for `i = 1` to `m`

  - `subR ← getReachable(Dom(c)[i], p, q_i)`
    if `subR ≠ ∅` then `subRs ← subR : subRs`
  `Q_c ← tQ_c`

if `reachable` then
  return `R`
else
  return ∅

Fig. 1. Computes deep semantics of quasi-additive terms as a union of ground semantics. The boolean `reachable` indicates if we can exhibit at least one `p`-free value headed by one of the constructors of `s`. The set `Q_c` corresponds to `Q_c(r)` in Equation 2 and is built by accumulation of the pattern complements from `r` for the arguments of `c`. Given a tuple `q ∈ Q_c, subRs` is a list (built with :) which stores the recursive results of `getReachable` over each element of `q`.

**Proposition 4.3 (Correctness).** Given `s ∈ S, p ∈ T_L(C, X)` and `r : s` a sum of constructor patterns, `getReachable(s, p, [r])` terminates and if we have `R = getReachable(s, p, [r])`, then

\[
[x_s^{-P} \setminus r] = \bigcup_{(s', p') ∈ R} [x_{s'}^{-P} \setminus p']
\]

Moreover, we have `|[x_s^{-P} \setminus r]| = ∅` iff `R = ∅`. 
Example 4.1. We consider the symbolic patterns from Example 3.2 and express their deep semantics as explained above. According to Proposition 4.2, we have \( \{ r_1 \} = \{ c_1(x_{s_2}^{-}, y_{s_1}^{-}) \} = \{ c_1(x_{s_2}^{-}, y_{s_1}^{-}) \} \cup \{ x_{s_2}^{-} \} \cup \{ y_{s_1}^{-} \} \) and we should expand \( \{ x_{s_2}^{-} \} \) and \( \{ y_{s_1}^{-} \} \).

To expand \( \{ y_{s_1}^{-} \} \) the sets \( Q_c(p_1) \) are computed for each \( c \in C_{s_1} = \{ c_1, c_2 \} \). First, following equation (1), \( \{ y_{s_1}^{-} \} = \{ c_1(x_{s_2}^{-}, y_{s_1}^{-}) \} \cup \{ c_2(z_{s_3}^{-}, y_{s_1}^{-}) \} \) and we can easily see that the complement relation in terms of ground semantics corresponds to set differences of cartesian products: \( \{ c_1(x_{s_2}^{-}, y_{s_1}^{-}) \} \setminus \{ c_2(z_{s_3}^{-}, y_{s_1}^{-}) \} = \{ c_1(x_{s_2}^{-}, y_{s_1}^{-}) \} \cup \{ c_2(z_{s_3}^{-}, y_{s_1}^{-}) \} \). We get thus, \( \{ y_{s_1}^{-} \} = \{ c_1(x_{s_2}^{-}, y_{s_1}^{-}) \} \cup \{ c_2(z_{s_3}^{-}, y_{s_1}^{-}) \} \). Hence, following equation (2), \( Q_c(p_1) = \{ c_1(z_{s_3}^{-}, \perp), (\perp, y_{s_1}^{-}) \} = \{ (p_2, \perp), (\perp, y_{s_1}^{-}) \} \) and \( Q_c(p_1) = \{ (\perp) \} \). Moreover, \( \{ c_1(x_{s_2}^{-}, y_{s_1}^{-}) \} \) and \( \{ c_2(z_{s_3}^{-}, y_{s_1}^{-}) \} \) are not empty (since \( c_1(c_2(c_2(c_6)), c_2(c_6)) \) and \( c_2(c_6) \) belong respectively to each of them) while \( \{ c_1(x_{s_2}^{-}, y_{s_1}^{-}) \} \) is clearly empty. Thus, \( \{ y_{s_1}^{-} \} = \{ y_{s_1}^{-} \} \cup \{ x_{s_2}^{-} \} \cup \{ y_{s_1}^{-} \} \cup \{ z_{s_3}^{-} \} \).

The getReachable algorithm continues the expansions until a fixpoint is reached. More precisely, we get \( \{ y_{s_1}^{-} \} = \{ y_{s_1}^{-} \} \cup \{ y_{s_1}^{-} \} \cup \{ x_{s_2}^{-} \} \cup \{ y_{s_1}^{-} \} \cup \{ z_{s_3}^{-} \} \), and therefore, the deep semantics of \( r_1 = c_1(x_{s_2}^{-}, y_{s_1}^{-}) \) is the union of \( \{ c_1(x_{s_2}^{-}, y_{s_1}^{-}) \}, \{ y_{s_1}^{-} \}, \{ z_{s_3}^{-} \}, \{ x_{s_2}^{-} \}, \{ y_{s_1}^{-} \} \) and \( \{ z_{s_3}^{-} \} \).

Propositions 4.2 and 4.3 guarantee that the deep semantics of any symbolic pattern and thus, of any term, can actually be expressed as the union of ground semantics of quasi-additive patterns. We introduce in the next section a method to automatically verify that the corresponding intersections with the semantics of a given pattern \( p \) are empty and check thus that a term is \( p \)-free.

4.2 Establishing pattern-free properties

Compared to the approach proposed in [3], we have to provide a method that also takes into account the specific behaviour of annotated variables. On the other hand, in order to establish pattern-free properties, we only need to check that the intersection of the semantics of a symbolic pattern \( t \) with the semantics of the given constructor pattern \( p \) is empty: thus, we want a TRS that reduces a pattern of the form \( t \times p \) to \( \perp \) if and only if its ground semantics is empty.

To this end, we introduce the TRS \( R_p \) presented in Figure 2. The rules generally correspond to their counterparts from set theory where constructor patterns correspond to cartesian products and the other extended patterns to the obvious corresponding set operations.

The rules A1, A2, resp. E2, E3, describe the behaviour of the conjunction, resp. the disjunction, \( w.r.t. \perp \). Rule E1 indicates that the semantics of a pattern containing a subterm with an empty ground semantics is itself empty, while rule S1 corresponds to the distributivity of conjunction over cartesian products. Similarly, rules S2 and S3 express the distributivity of conjunction over disjunction.

The semantics of a variable of a given sort is the set of all ground constructor patterns of the respective sort. Thus, the difference between the ground semantics
Remove empty sets:

(A1) \( \bot + v \Rightarrow v \)
(A2) \( v + \bot \Rightarrow v \)

Distribute sets:

(E1) \( \delta(\bar{w}_1, \ldots, \bot, \ldots, \bar{w}_n) \Rightarrow \bot \)
(E2) \( \bot \times v \Rightarrow \bot \)
(E3) \( v \times \bot \Rightarrow \bot \)
(S1) \( \delta(\bar{w}_1, \ldots, \bar{w} + \bar{u}, \ldots, \bar{w}_n) \Rightarrow \delta(\bar{w}_1, \ldots, \bar{w}, \ldots, \bar{w}_n) + \delta(\bar{w}_1, \ldots, \bar{w}_n) \)
(S2) \( (\bar{w}_1 + \bar{w}_2) \times v \Rightarrow (\bar{w}_1 \times v) + (\bar{w}_2 \times v) \)
(S3) \( \bar{w} \times (\bar{w}_1 + \bar{w}_2) \Rightarrow (\bar{w} \times \bar{w}_1) + (\bar{w} \times \bar{w}_2) \)

Simplify complements:

(M1) \( \bar{v} \setminus \bar{z} \Rightarrow \bot \)
(M2) \( \bar{v} \setminus \bot \Rightarrow \bar{v} \)
(M3) \( (\bar{v} + \bar{w}) \setminus \bar{v} \Rightarrow (\bar{v} \setminus \bar{w}) + (\bar{v} \setminus \bar{w}) \)
(M5) \( \bot \setminus v \Rightarrow \bot \)
(M6) \( \alpha(\bar{v}_1, \ldots, \bar{w}_n) \setminus (\bar{v} + \bar{w}) \Rightarrow (\alpha(\bar{v}_1, \ldots, \bar{w}_n) \setminus \bar{v}) \)
(M7) \( \alpha(\bar{v}_1, \ldots, \bar{w}_n) \setminus \alpha(\bar{t}_1, \ldots, \bar{t}_n) \Rightarrow \alpha(\bar{v}_1 \setminus \bar{t}_1, \ldots, \bar{w}_n \setminus \bar{t}_n) \)
(M8) \( \alpha(\bar{v}_1, \ldots, \bar{w}_n) \setminus \beta(\bar{v}_1, \ldots, \bar{w}_n) \Rightarrow \alpha(\bar{v}_1, \ldots, \bar{w}_n) \quad \text{with } \alpha \neq \beta \)

Simplify conjunctions:

(T1) \( \bar{v} \times \bar{z} \Rightarrow \bar{v} \)
(T2) \( \bar{v} \setminus \bar{z} \Rightarrow \bar{v} \)
(T3) \( \alpha(\bar{v}_1, \ldots, \bar{w}_n) \times \alpha(\bar{t}_1, \ldots, \bar{w}_n) \Rightarrow \alpha(\bar{v}_1 \times \bar{t}_1, \ldots, \bar{w}_n \times \bar{w}_n) \)
(T4) \( \alpha(\bar{v}_1, \ldots, \bar{w}_n) \times \beta(\bar{v}_1, \ldots, \bar{w}_n) \Rightarrow \bot \quad \text{with } \alpha \neq \beta \)

Simplify p-free:

(P1) \( \bar{v}^P \times \alpha(\bar{v}_1, \ldots, \bar{w}_n) \Rightarrow \sum_{c \in C} c(\bar{z}_1 \bar{z}_2 \ldots, \bar{z}_m \bar{z}_m^P) \times (\alpha(\bar{v}_1, \ldots, \bar{w}_n) \setminus \bar{v}) \quad \text{with } m = \text{arity}(c) \)
(P2) \( \alpha(\bar{v}_1, \ldots, \bar{w}_n) \times (\bar{v}^P \setminus \bar{t}) \Rightarrow (\alpha(\bar{v}_1, \ldots, \bar{w}_n) \times \bar{v}^P) \setminus \bar{t} \quad \text{if } [\bar{v}^P \setminus \bar{t}] \neq \emptyset \)
(P3) \( \bar{v}^P \times (\bar{v}^P \setminus \bar{t}) \Rightarrow (\bar{v}^P \times \bar{v}^P) \setminus \bar{t} \quad \text{if } [\bar{v}^P \setminus \bar{t}] \neq \emptyset \)
(P4) \( (\bar{v}^P \setminus \bar{t}) \times v \Rightarrow (\bar{v}^P \times v) \setminus \bar{t} \quad \text{if } [\bar{v}^P \setminus \bar{t}] \neq \emptyset \)
(P5) \( (\bar{v}^P \setminus \bar{t}) \setminus \bar{t} \Rightarrow (\bar{v}^P \setminus \bar{t}) \quad \text{if } [\bar{v}^P \setminus \bar{t}] \neq \emptyset \)
(P6) \( \bar{v}^P \setminus \bar{t} \Rightarrow \bot \quad \text{if } [\bar{v}^P \setminus \bar{t}] = \emptyset \)

Fig. 2. \( R_p \): reduce pattern of the form \( t \times p; \bar{v}, \bar{v}_1, \ldots, \bar{v}_n, \bar{w}, \bar{w}_1, \ldots, \bar{w}_n \) range over quasi-additive patterns, \( \bar{v}, \bar{t} \) range over pure regular additive patterns, \( \bar{t}_1, \ldots, \bar{t}_n \) range over pure symbolic patterns, \( \bar{t}, \bar{p} \) range over constructor patterns, \( \bar{v} \) ranges over pattern variables. \( \alpha, \beta \) expand to all the symbols in \( C \), \( \delta \) expands to all symbols in \( C^n > 0 \).

of any pattern and the ground semantics of a variable of the same sort is the empty set (rule M1). The rules M2-M6 correspond to set operation laws for complements. Rule M7 corresponds to the set difference of cartesian products; the case when the head symbol is a constant \( c \) corresponds to the rule \( c \setminus c \Rightarrow \bot \). Rule M8 corresponds to the special case where complemented sets are disjoint.

The rules T1 and T2 indicate that the intersection with the set of all terms has no effect, rule T3 corresponds to distribution laws for the joint intersection, while T4 corresponds to the disjointed case.

We have seen that the ground semantics of an annotated variable is obtained by considering, for each constructor of the appropriate sort, the set of all terms
having this symbol at the root position complemented by the pattern in the
annotation and taking the union of all these sets, \( R_p \) uses this property in the
rule P1 to expand annotated variables allowing thus for the triggering of the other
rules for conjunction. Note that \( z_i \) are fresh variables generated automatically.
The rules P2, P3 and P4 express the respective behaviour of conjunction over
complements \((A \cap (B \setminus C)) = (A \setminus C) \cap B = (A \cap B) \setminus C)\).

Finally, we can observe that, thanks to the algorithm introduced in Figure 1, we can
determine if \( \{ \tau_s^p \setminus \tau \} = \emptyset \). Moreover, by definition, \( [t] = \emptyset \) if and only if
\( [t] = \emptyset \). Therefore, the TRS is finalized by the rule P6 which eliminates (when possible) annotated variables. In order to apply P6 exhaustively, \( R_p \) also needs
a rule to perform some \( \setminus \)-factorization around variables, resulting in the rule P5.

**Proposition 4.4 (Semantics preservation).** For any extended patterns \( p, q \),
if \( p \rightarrow R_p q \) then \([p] = [q]\).

We have seen that the algorithm in Figure 1 always terminates and that it can
be used to decide the conditions in the TRS \( R_p \) (Proposition 4.3). Based on this,
we can prove the convergence of the TRS \( R_p \). While we cannot provide a simple
description of the normal forms obtained by reduction of a general extended patterns,
\( R_p \) can be used to establish the emptiness of a given intersection:

**Proposition 4.5.** The rewriting system \( R_p \) is confluent and terminating. Given
a quasi-additive pattern \( t \) and a constructor pattern \( p \), we have \( t \times p \rightarrow R_p \bot \) if
and only if \([t \times p] = \emptyset\).

### 4.3 Establishing semantics preserving properties

The approach proposed in the previous section allows the systematic verification
of pattern-free properties for any term in \( T(\mathcal{F}, \mathcal{X}) \) such that \( \tilde{t} \) is linear. It is
easy to see if we denote by \( L(t) \) the term obtain by replacing all the variables in
the term \( t \) by fresh ones then \([t] \subseteq [L(t)]\). We can thus linearize, if necessary,
the right-hand sides of the rules of \( \tilde{t} \) a CBTRS and subsequently check that it is
semantics preserving.

**Example 4.2.** We apply the approach to check that the CBTRS in Example 4.3
is semantics preserving. For this we need to prove that \( c_1(g^{-p_2}(x_{s_2}), f^{-p_1}(y_{s_1})) \)
and \( c_2(z_{s_2}) \) are \( p_1 \)-free, and that \( c_3(c_2(z_{s_2})) \) and \( c_3(f^{-p_1}(y_{s_1})) \) are \( p_2 \)-free.

In order to prove that \( r_1 = c_1(g^{-p_2}(x_{s_2}), f^{-p_1}(y_{s_1})) \) is \( p_1 \)-free, we should
first compute the deep semantics of \( \tilde{r}_1 = c_1(x_{s_2}^{-p_2}, y_{s_1}^{-p_1}) \) and we have seen in
Example 4.1 how \( \text{getReachable} \) is used to compute this deep semantics as the
union of \([c_1(x_{s_2}^{-p_2}, y_{s_1}^{-p_1})], [y_{s_1}^{-p_1}], [z_{s_2}^{-p_1}], [x_{s_2}^{-p_2} \setminus p_2], [x_{s_2}^{-p_2}], [y_{s_1}^{-p_2}], [z_{s_2}^{-p_2}]\)
For all the terms in the union we compute their conjunction with \( p_1 \) using \( R_p \)
which reduces them all to \( \bot \). Hence, by Proposition 4.1, \( r_1 \) is \( p_1 \)-free.

Similarly, we can check that \( c_2(z_{s_2}) \) is \( p_1 \)-free, and \( c_3(c_2(z_{s_2})) \) and \( c_3(f(y_{s_1})) \)
are \( p_2 \)-free. Thus, the CBTRS is semantics preserving. It is easy to check that it is
also terminating and consequently, the normal form of any term \( f(t), t \in T_{s_1}(\mathcal{F}) \),
is \( p_1 \)-free and the normal form of any term \( g(u), u \in T_{s_2}(\mathcal{F}) \), is \( p_2 \)-free.
We can now come back to the initial flattening example presented in the introduction. We consider a signature consisting of the sorts and constructors already presented in the introduction to which we add the defined symbols $D = \{\text{flatten}E^{-p} : \text{Expr} \rightarrow \text{Expr}, \text{flatten}L^{-p} : \text{List} \rightarrow \text{List}, \text{concat}^{-\perp} : \text{List} \times \text{List} \rightarrow \text{List}\}$. with $p = \text{conc}(\text{lst}(l_1), l_2)$, to indicate that the corresponding functions defined by the following CBTRS aim at eliminating this pattern:

$$
\begin{align*}
\text{flatten}E^{-p}(\text{str}(s)) & \rightarrow \text{str}(s) \\
\text{flatten}E^{-p}(\text{lst}(l)) & \rightarrow \text{lst}(\text{flatten}L^{-p}(l)) \\
\text{flatten}L^{-p}(\text{nil}()) & \rightarrow \text{nil}() \\
\text{flatten}L^{-p}(\text{cons}(\text{str}(s), l)) & \rightarrow \text{cons}(\text{str}(s), \text{flatten}L^{-p}(l)) \\
\text{flatten}L^{-p}(\text{cons}(\text{lst}(l_1), l_2)) & \rightarrow \text{flatten}L^{-p}(\text{concat}^{-\perp}(l_1, l_2)) \\
\text{concat}^{-\perp}(\text{cons}(e, l_1), l_2) & \rightarrow \text{cons}(e, \text{concat}^{-\perp}(l_1, l_2)) \\
\text{concat}^{-\perp}(\text{nil}(), l) & \rightarrow l
\end{align*}
$$

Thanks to the method introduced in the previous section we can check that the right-hand sides of the first 5 rules are $p$-free and hence, as explained in Section 3.3, that the CBTRS is semantics preserving. This CBTRS is clearly terminating and complete and thus, we can guarantee that the normal forms of such terms are $p$-free values.

The method has been implemented in Haskell and the source code together with several examples (including the ones presented here) can be downloaded from [https://github.com/plermusiaux/pfree_check](https://github.com/plermusiaux/pfree_check); an online version is available at [http://htmlpreview.github.io/?https://github.com/plermusiaux/pfree_check/blob/webnix/out/index.html](http://htmlpreview.github.io/?https://github.com/plermusiaux/pfree_check/blob/webnix/out/index.html). The implementation takes as input a file defining the signature and the CBTRS to be checked and returns the (potentially empty) set of non pattern-free preserving rules (i.e. rules that do not satisfy the pattern-free requirements implied by the signature). For each such rule we provide a set of terms whose ground semantics is included in the deep semantics of the right-hand side of the rule and that do not satisfy the pattern-free property required by the left-hand side.

The complexity of the method for checking the pattern-freeness w.r.t. to a given pattern $p$ is exponential on the depth of $p$ with a growth rate proportional to the (maximum) arity of the symbols present in $p$. Benchmarks performed on the implementation optimized to minimize repetitive computations showed that, when considering terms and patterns of depth 5 with symbols of arity 6, checking the pattern-freeness of a single term takes $\sim 200\text{ms}$, and checking the semantics preservation of a CBTRS of 25 rules takes $\sim 3\text{s}$ (on an Intel Core i5-8250U). In practice, the size of the pattern annotations is generally lower that the ones we experimented with and we consider that despite the exponential complexity the concrete performances are reasonable for a static analysis technique.

### 5 Related work

While the work presented in this paper introduces an original approach to express and ensure a particular category of syntactical guarantees associated to program
transformation, a number of different approaches presenting methods to obtain some guarantees for similar classes of functions exist in the literature.

**Tree automata completion**  Tree automata completion consists in techniques used to compute an approximation of the set of terms reachable by a rewriting relation [14]. Such techniques could, therefore, be applied to solve similar problems to the one presented in this paper. The application of this approach is nevertheless usually conditioned by the termination of both the TRS and the set of equational approximations used [30][15]. Thus, while providing a more precise characterization of the approximations of the normal forms, these techniques are constrained by these termination conditions. Therefore we believe the formalism presented in this paper provides a viable and original alternative to such techniques, particularly in the context of verification of pass transformations [21].

**Recursion schemes**  Some formalisms propose to deal with higher order functions through the use of higher order recursion schemes, a form of higher order grammars that are used as generators of (possibly infinite) trees [22]. In such approaches, the verification problems are solved by model checking the recursion schemes generated from the given functional program. Higher order recursion schemes have also been extended to include pattern matching [27] and provide the basis for automatic abstraction refinement. These techniques address in a clever way the control-flow analysis of functional programs while the formalism proposed in our work is more focused on providing syntactic guarantees on the shape of the tree obtained through a pass-like transformation. The use of the annotation system also contributes to a more precise way to express and control the considered over-approximation.

**Tree transducers**  Besides term rewriting systems, another popular approach for specifying transformations consists in the use of tree transducers [23]. Transducers have indeed been shown to have a number of appealing properties when applied for strings, even infinite [2], and most notably can provide an interesting approach for model checking certain classes of programs thanks to the decidability of general verification problems [11]. Though the verification problems we tackle here are significantly more strenuous for tree transducers, Kobayashi et al. introduced in [23] a class of higher order tree transducers which can be modeled by recursion schemes and thus, provided a sound and complete algorithm to solve verification problems over that class. We claim that annotated CBTRSs are easier to grasp when specifying pass-like transformations and are less intrusive for expressing the pattern-free properties.

**Refinement types**  Formalisms such as refinement types [11] can be seen as an alternative approach for verifying the absence, or presence, of specific patterns. In particular, notions such as constructor subtypes [6] could be used to construct complex type systems whose type checking would provide guarantees similar to the ones provided by our formalism. This would however result in the construction of multiple type systems in order to type check each transformation as was the case in the original inspiration of our work [21].
6 Conclusion and perspectives

We have proposed a method to statically analyse constructor term rewrite systems and verify the absence of patterns from the corresponding normal forms. We can thus guarantee not only that some constructors are not present in the normal forms but we can also be more specific and verify that more complex constructs cannot be retrieved in the result of the reduction. Such an approach avoids the burden of specifying a specific language to characterize the result of each (intermediate) transformation, as the user is simply requested to indicate the patterns that should be eliminated by the respective transformation.

Different termination analysis techniques [3,18,17] and corresponding tools like AProVE [16,12] and TTT2 [24] can be used for checking the termination of the rewriting systems before applying our method for checking pattern-free properties. On the other hand, the approach applies also for CBTRS which are not complete or not strongly normalising and still guarantees that all the intermediate terms in the reduction are pattern-free; in particular, if the CBTRS is weakly normalising the existing normal forms are pattern-free. It is worth mentioning that the approach extends straightforwardly to sums of constructor patterns of the form $p = p_1 + \cdots + p_n$ in the annotations to indicate simultaneously $p_i$-freeness w.r.t. all the patterns in the sum.

We believe this formalism opens a lot of opportunities for further developments. In the current version, the verification relies on an over-approximation of the set of reducts and thus, can lead to false negatives. For example, an alternative rule $\text{flattenL}(\text{cons}(\text{lst}(l_1), l_2)) \rightarrow \text{concat} (\text{flattenL}(l_1), \text{flattenL}(l_2))$ in our flattening CBTRS would be reported as non pattern-preserving. In our experience, such false negatives arise when the annotations for some symbols are not precise enough in specifying the expected behaviour and, although we conjecture this might indicate some issues in the design of the CBTRS, we work on an alternative approach allowing for a finer-grain analysis. While false negatives could also arise when the right-hand side of a rule has to be linearized, the current implementation already uses an aliasing technique to handle such cases; the technical details have been omitted in the paper due to the space restrictions.

We also intend to extend and use the approach in the context of automatic rewrite rule generation techniques, such as the one introduced in [8], in order to automatize the generation of boilerplate code as in [21].

References


Nominal Unification with Letrec and Environment-Variables *

Manfred Schmidt-Schauß and Yunus Kutz

Goethe-University Frankfurt, Germany \{schauss,kutz\}@ki.cs.uni-frankfurt.de

Abstract. Unification algorithms of nominal expressions with letrec and atom- and expression-variables are already described in the literature. However, only explicit environments could be treated in nominal unification and the use of abstract environments was restricted to nominal matching. This severely restricts the use of algorithms in applications. The following two restrictions permit a step forward and strongly improve the coverage of the application cases: expression- and environment-variables are restricted to occur at most once in the input equations. A terminating and complete nominal unification algorithm is described that computes complete sets of constrained unifiers. Since the set of ground instances of a complete set may be empty due to constraints, we also provide a decision algorithm and show that nominal unifiability under these restrictions is NP-complete. For input without an occurrence-restriction for expression-variables and w.r.t. garbage-free ground expressions, we sketch an adapted unification algorithm that produces a complete set of unifiers in NP time. For the decision problem we conjecture that it is harder in this case. We believe that lifting the linearity-restrictions for environment-variables leads to a prohibitively high computational complexity.

Keywords: Nominal unification · Letrec-expressions · Abstract environments · Program transformations · Automated deduction

1 Introduction

The goal of this paper is to extend the expressive power of nominal unification to allow automated reasoning in calculi with let-environments with multiple, and commutative bindings, in particular recursive bindings, but also non-recursive ones and $\nu$-restrictions. Recursive bindings appear for example in functional programming languages such as Haskell [11, 6], F# [5] and OCaml [14], and $\nu$-restrictions for example in the pi-calculus [13, 17].

Reasoning on program transformations, their correctness and their influence on resource consumption can often be supported by considering overlaps of

* The authors are supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SCHM 986/11-1.
transformation rules with rules of the operational semantics. In programming languages with binders, first order unification is not powerful enough for this task, and higher-order unification is undecidable or too complex. Nominal unification provides a very good balance between expressiveness and computational properties. Nominal techniques \cite{16, 15} support machine-oriented reasoning on the syntactic level for higher-order languages and support alpha-equivalence. An algorithm for (plain) nominal unification was first described in \cite{27}, which outputs unique most general unifiers (with constraints). The essential extension to the expressiveness of first order terms are syntactic permutations of atoms (i.e. variable names) and freshness constraints \( a \# e \) that can restrict free occurrences of names (\( a \# e \) means that \( a \) does not occur free in \( e \)). Efficient algorithms for nominal unification are given in \cite{1, 10}, exhibiting a quadratic algorithm. Nominal unification is also used in higher-order logic programming \cite{2} and nominal techniques in automated theorem provers like Nominal Isabelle \cite{25, 26}. \( \alpha \)-Check \cite{3} was developed for property testing of systems which are specified using nominal logic such as \( \alpha \)-Prolog and maybe in a future version of Nominal Isabelle.

The extension of nominal unification to atom-variables was tackled in \cite{2} where an algorithm to produce a specific unifier was provided. This is improved by developing an algorithm to compute unique most general unifiers and showing that the decision problem is NP-complete \cite{23}. There are also recent applications using atom-variables for confluence checking in nominal rewriting \cite{7}. An extension of nominal unification to languages with a recursive let without atom-variables was worked out in \cite{18}, where it was shown that the nominal unification and matching problems are NP-complete. The nominal unification algorithm for letrec was extended to atom-variables in \cite{19, 20}. Also, a nominal matching algorithm for letrec with environment-variables, but without atom-variables is proposed in \cite{19, 20}. However, adapting the matching algorithm with environment-variables to a unification algorithm was left open as it appeared to require stronger methods.

A motivating example for this paper is the rule (let) which is used in the operational semantics of the calculus LR \cite{24}. It reads:

\[(\text{let}): (\text{let } \text{env in } (\text{let } \text{env'} in r)) \rightarrow (\text{let } \text{env, env'} in r)\]

It has the restriction that in the right hand side the environment \( \text{env}' \) must not capture free variables in \( \text{env} \), and that the binding variables in \( \text{env}, \text{env}' \) must be distinct. This can be encoded with the extended freshness constraints \( LV(\text{env}')\#LV(\text{env}) \), and \( LV(\text{env'})\#LV(\text{env}) \), where \( LV(.) \) means the set of binding names in the top-level of a let-environment, and \( M_1\#M_2 \) abbreviates \( \{e\#M_2 \mid e \in M_1\} \).

The overlap of the subexpression \( (\text{let } \text{env'} \text{ in } r) \) with the full left hand side of the same rule is a step in a confluence check of a subset of the rules in LR (see also Example \ref{3.5}). It is computed by applying a nominal unification algorithm, i.e. solving the equation \( (\text{let } E \text{ in } S) \equiv (\text{let } E_1 \text{ in } (\text{let } E_2 \text{ in } S')) \) with environment-variables \( E, E_1, E_2 \), and expression-variables \( S, S' \) together with the constraints \( LV(E_2)\#LV(E_1) \), \( LV(E_2)\#LV(E_1) \). This illustrates that environment-variables and extended freshness constraints are required. Note that
environment-variables occur only once in the equation. Naively solving this equation would lead to 
\[ E \rightarrow E_1, S \rightarrow (\text{let} \ E_2 \ \text{in} \ S') \]. Unfortunately this is not the most general unifier, since instances of the two expressions with conflicting names are not covered. A general solution requires permutation variables \( P \), such that the solution includes for example the substitution \( E \rightarrow P \cdot E_1 \). We will show in this paper, how permutation variables that are introduced by the unification algorithm can be tamed by specifying their abstract mapping behavior.

The occurrences of expression- and environment-variables in our examples are linear, which fits the input restrictions of our unification-algorithm. It can also deal with non-linear occurrences of atom-variables as for example in an expression \( \text{let} \ x = t, \text{env} \ \text{in} \ldots x \ldots \). We will sketch a more complex algorithm that also can deal with non-linear occurrences of expression-variables, but non-linear occurrences of environment-variables cannot be handled.

The achievements in this paper are the formulation of a nominal unification and of a decision algorithm for equations and constraints in a higher-order calculus with letrec and atom-, expression- and environment-variables where expression- and environment-variables occur linearly in the set of equations (Theorem 3.7 and 4.10). This is a step forward in generalizing nominal unification algorithms to declarative functional programming languages like Haskell. The results are also applicable to (non-recursive) let with multiple commuting bindings and to \( \nu \)-bindings. The complexity of the decision problem is shown to be NP-complete (see Theorem 4.10, Theorem 5.3 and Corollary 4.11).

We also investigated the extension where expression-variables may occur unrestricted, i.e. multiple times. This enforces to restructure the data structure to so-called multi-equations in the unification algorithm, and to adapt in particular the decomposition-rules to the data structure. Also, we change the semantics to the language of garbage-free ground expressions. A full description of this algorithm would exceed the available space, but we provide a sketch. Fortunately, the complexity of the unification algorithm as a device for producing solutions is nondeterministic polynomial time. However, the decision algorithm could not be adapted in a straightforward way. We are still working on it and conjecture that the complexity of the decision problem for non-linear occurrences of expression-variables and linear occurrences of environment-variables is in NEXPTIME.

The structure of the paper is to first introduce the problem and language (Sect. 2), Sect. 3 contains a nominal unification algorithm for the extension and Sect. 4 describes the decision algorithm for equations. Section 5 gives a sketchy overview of the extension of the unification algorithm to nonlinear occurrences of expression-variables. Sect. 6 concludes.

## 2 Nominal Expressions

We first introduce some notation. Let \( F \) be a set of function symbols \( f \in F \), s.t. each \( f \) has a fixed arity \( ar(f) \geq 0 \). Let \( At \) be the set of atoms ranged over by \( a, b, c \). The ground language \( NL_{\text{let}}^{\text{letr}} \) is defined by the grammar:
\[ e ::= a \mid (f \, e_1 \ldots e_{\text{ar}(f)}) \mid \lambda a.\, e \mid \textbf{letr} \, a_1.\, e_1, \ldots, a_n.\, e_n \ \text{in} \ e \]

where \( \lambda \) is a binder for atoms, and \textbf{letr} \ldots in \ldots is the recursive let, where \( a_i \)
are the (binding) atoms in the letrec-environment and where \( a_i.\, e_i \) is called a binding. While we are only interested in expressions with environments where the binding atoms are mutually distinct, we also allow other environments, e.g. \textbf{letr} \ a.1.\ a.1 \ \text{in} \ 1 \ . \) However, we will use the constraint system to mark them as invalid. For example, in \( t = \textbf{letr} \ a.(g \ b \ a).b.(g \ a \ b) \ \text{in} \ (g \ a \ b) \) all occurrences of \( a, b \) are bound. Also, the order of bindings in a \textbf{letr}-environment is irrelevant, such that \( \textbf{letr} \ b.(g \ a \ b).a.(g \ b \ a) \ \text{in} \ (g \ a \ b) \) is the same expression as \( t \).

The basic freshness constraint \( a\#e \) is valid if \( a \) is not free in \( e \). We write \( M\#e \) as an abbreviation of \( \{a\#e \mid a \in M\} \). A set of (basic) constraints \( \mathcal{V} \) is valid if all constraints in \( \mathcal{V} \) are valid. \( \mathcal{L}(\mathcal{E}) \) is the multiset of top let-binders in a letrec-environment \( \mathcal{E} \). The reason for \( \mathcal{L}(\mathcal{E}) \) being a multiset, rather than a set, is that we need to consider invalid environments and constraints, e.g. \( a\#a \) and \( \mathcal{L}(\mathcal{E}) = \{a, a\} \), which may arise during unification and produce a failure.

As a reminder, the \( \alpha \)-equivalence relation \( \sim \) on \( \mathcal{N}_{\text{letr}}^{\alpha} \) is defined as the equivalence closure of renamings of bound atoms. For a better algorithmic treatment of \( \alpha \)-equivalence, we will use a decomposition principle for letrec-expressions modulo \( \alpha \), which is improved compared to the method used in [18], since it supports a systematic way of describing the mapping behavior between the bindings with the help of a permutation-variable in the unification algorithm:

**Lemma 2.1.** Let \( e_1 = (\textbf{letr} \ a_1.s_1, \ldots, a_n.s_n \ \text{in} \ r) \) and \( e_2 = (\textbf{letr} \ b_1.t_1, \ldots, b_n.t_n \ \text{in} \ r') \) be \( \mathcal{N}_{\text{letr}}^{\alpha} \)-expressions, where \( a_i \) are pairwise distinct, \( b_i \) are pairwise distinct, but \( \{a_i \mid 1 \leq i \leq n\} \cap \{b_i \mid 1 \leq i \leq n\} \) may be non-empty. Then \( e_1 \sim e_2 \) is equivalent to the following conditions:

1. There is a permutation \( \varphi \) on atoms, such that \( \text{dom}(\varphi) \subseteq \{a_1, \ldots, a_n\} \cup \{b_1, \ldots, b_n\} \), and it extends the mapping \( \{b_i \mapsto a_{\varphi(i)} \mid i = 1, \ldots, n\} \), where \( \varphi \) is a permutation on the set \( \{1, \ldots, n\} \).
2. For \( M := \{a_1, \ldots, a_n\}\{b_1, \ldots, b_n\} \), we have \( \varphi(M) = \{b_1, \ldots, b_n\}\{a_1, \ldots, a_n\} \).
3. \( M\#(\textbf{letr} \ b_1.t_1, \ldots, b_n.t_n \ \text{in} \ r') \).
4. \( r \sim \varphi(r') \) and \( s_{\varphi(i)} \sim \varphi(t_i) \) for \( i = 1, \ldots, m \) hold.

Note that the permutation \( \varphi \) is not necessarily unique, since for \( (\textbf{letr} \ a.s_1; b.s_2 \ \text{in} \ s_3) \sim (\textbf{letr} \ c.s_1; d.s_2 \ \text{in} \ s_3) \) the permutation \( \varphi \) must map \( \{c \mapsto a, d \mapsto b\} \), but it may in addition map either \( \{a \mapsto c, b \mapsto d\} \) or \( \{a \mapsto d, b \mapsto c\} \). Lemma 2.1 leads to the following decomposition principle for all constructs w.r.t. \( \sim \), which is equivalent to the definition of \( \alpha \)-equivalence on \( \mathcal{N}_{\text{letr}}^{\alpha} \):

**Lemma 2.2.** \( \alpha \)-equivalence \( \sim \) in \( \mathcal{N}_{\text{letr}}^{\alpha} \) is characterized by the following rules:

\[
\begin{align*}
\text{a } & \sim \text{a} \quad (f \ e_1 \ldots e_{\text{ar}(f)}) \sim (f \ e'_1 \ldots e'_{\text{ar}(f)})
\end{align*}
\]

\[
\begin{align*}
\varphi i : e_i & \sim e'_i \quad e \sim e' \quad a\#e' \wedge e \sim (a \ b).e'
\end{align*}
\]

The four conditions of Lemma 2.1 hold.

\( \text{letr} \ a_1.s_1, \ldots, a_n.s_n \ \text{in} \ r \sim \text{letr} \ b_1.t_1, \ldots, b_n.t_n \ \text{in} \ r' \)
Definition 2.3 (Expression languages). Let \( S \) be a set of expression-variables ranged over by \( S, T \); let \( A \) be the set of atom-variables ranged over by \( A, B \); let \( E \) be a set of variables standing for letrec-environments ranged over by \( E \); and let \( P \) be a set of permutation-variables ranged over by \( P \). The grammar of the nominal language \( \text{N}_{\text{ASPE}}^{\text{letr}} \) with atoms, atom-variables, expression-variables, permutation-variables and environment-variables is:

\[
\begin{align*}
e & ::= W \mid \pi \cdot S \mid (f_{e_1} \ldots e_{\text{ar}(f)}) \mid \lambda W.e \mid \text{letr} \; \text{env} \; \text{in} \; e \\
\pi & ::= \emptyset \mid (W \cdot W') \cdot \pi \mid P \cdot \pi \mid P^{-1} \cdot \pi \\
\text{env} & ::= \emptyset \mid \pi \cdot \text{env} \mid (\pi \cdot a.e; \text{env}) \mid (\pi \cdot A.e; \text{env}) \mid (\pi \cdot E; \text{env}) \\
W & ::= \pi \cdot a \mid \pi \cdot A
\end{align*}
\]

where \( \pi \) is a permutation and \( \emptyset \) denotes the identity.

One sublanguage is the ground language without variables \( \text{N}_{\text{AS}}^{\text{letr}} \). For the algorithms we will use sublanguages \( \text{N}_{\text{AS}}^{\text{letr}}, \text{N}_{\text{ASPE}}^{\text{letr}}, \text{N}_{\text{ASPE}}^{\text{letr}} \), where only the variable sorts mentioned in the index are used in the grammar, and where atoms are not permitted.

Note that this definition permits nested permutation expressions. The expression \((\pi \cdot A) (\pi' \cdot A')\) is a swapping that illustrates the nesting. An expression of the form \( \pi \cdot X \) where \( X \) is some variable \( A, S, E \) is called suspension. The inverse \( \pi^{-1} \) of a permutation \( \pi = w_1 \ldots w_n \) is the expression \( w_n^{-1} \ldots w_1^{-1} \) where \((W_1 W_2)^{-1} = (W_2 W_1)\), and \((\pi^{-1})^{-1} = \pi\). \( \text{AtVar}(O) \) are the atom-variables contained in \( O \), \( \text{ExVar}(O) \) the expression-variables contained in \( O \) and \( \text{Var}(O) = \text{AtVar}(O) \cup \text{ExVar}(O) \).

The ground language of \( \text{N}_{\text{ASPE}}^{\text{letr}} \subset \text{N}_{\text{ASPE}}^{\text{letr}} \) is \( \text{N}_{\text{ASPE}}^{\text{letr}} \). Note that this is a slight abuse of the notion of a ground language. A ground substitution \( \rho \) replaces atom-variables with atoms, expression-variables with ground expressions, permutation-variables with ground permutations, and environment-variables with ground environments. After applying a ground substitution \( \rho \), the permutations are applied, such that the result is in \( \text{N}_{\text{ASPE}}^{\text{letr}} \). In fact every ground substitution \( \rho \) is an expression-structure homomorphism from \( \text{N}_{\text{ASPE}}^{\text{letr}} \) into \( \text{N}_{\text{ASPE}}^{\text{letr}} \), and from \( \text{N}_{\text{ASPE}}^{\text{letr}} \) into \( \text{N}_{\text{ASPE}}^{\text{letr}} \). The language \( \text{N}_{\text{ASPE}}^{\text{letr}} \) serves as an intermediate language during the interpretation of \( \text{N}_{\text{ASPE}}^{\text{letr}} \) expressions in proofs.

Constraints are abstract conditions, formulated for \( \text{N}_{\text{ASPE}}^{\text{letr}} \)-expressions and with semantics in \( \text{N}_{\text{ASPE}}^{\text{letr}} \). We will overload the notation, such that it can be used in both languages and in the algorithms.

Definition 2.4 (Constraints in \( \text{N}_{\text{ASPE}}^{\text{letr}} \)).

A freshness constraint has the form \( A \# e \), where \( e \) is an \( \text{N}_{\text{ASPE}}^{\text{letr}} \)-expression. General constraints extend this by:

\[
\begin{align*}
&\# \{W_1, \ldots, W_n\} \text{ and } \# LV(\text{env}), \text{ where } \{W_1, \ldots, W_n\} \text{ is a multiset.} \\
&LV(\text{env}) \# e, \text{ and } (LV(\text{env}_1) \setminus LV(\text{env}_2)) \# e, \text{ and } LV(\text{env}) \# E. \\
&\text{dom}(P) \subseteq LV(\text{env}_1) \cup LV(\text{env}_2) \\
&P \cdot LV(\text{env}_2) = LV(\text{env}_1) \\
&P \cdot (LV(\text{env}_1) \setminus LV(\text{env}_2)) = (LV(\text{env}_2) \setminus LV(\text{env}_1))
\end{align*}
\]
– $A = \pi \cdot B$, which is an abbreviation of $A\#\lambda \pi \cdot B.A$.

Let $\gamma$ be a ground substitution. A constraint $A\#e$ is satisfied by $\gamma$ if $\gamma(A)$ does not occur free in $\gamma e$. The constraint $\#\{W_1, \ldots, W_n\}$ (for the multiset) is satisfied by $\gamma$, if $i \neq j$ implies $W_i \gamma \neq W_j \gamma$. The constraint $LV(env)\#e$ is satisfied by $\gamma$ if for all $a \in LV(env) \gamma$, $a\#e \gamma$ holds. The constraint $LV(env)\#E$ is satisfied by $\gamma$ if for all $a \in LV(env) \gamma$, $a\#(let E in \lambda A.A) \gamma$ holds. Satisfiability of the other constraints is clear from these explanations.

A solution of a set $\nabla$ of (general) constraints is a ground substitution $\gamma$, s.t. $\nabla \gamma$ is ground, and all constraints in $\nabla \gamma$ hold in $NL_{letr}$.

Example 2.5. As an example of the power of the language and of the constraint system we discuss several reduction and transformation rules of the intended applications. In particular we show which constraints are necessary to make the rules correct for every application.

- $let E_1 in letr E_2 in S \rightarrow letr E_1, E_2 in S$ (rearranging letrec-environments). To avoid variable capture in the resulting expression, the constraint $LV(E_2)\#E_1$ is sufficient. Syntactic correct instances require also $LV(E_1, E_2)$.
- $let A.(\lambda B.S_1), E_1 in A \rightarrow letr A.(\lambda B.S_1), E_1 in (\lambda B.S_1)$ (a copy-rule for abstractions, also called dereferencing). This requires $LV(A.(\lambda B.S_1), E_1)$.
- $let A.(let E_2 in S_1), E_1 in S_2 \rightarrow letr A.S_1, E_2, E_1 in S_2$ (rearranging letrec-environments). The required constraints are $LV(A.(let E_2 in S_1), E_1)$, $LV(E_2)$, $LV(A.S_1, E_2, E_1)$, $LV(E_2)\#(let E_1 in S_1)$.
- $(let E_1, A.S, E_2 in S) \rightarrow (let E_1 in S)$ (a garbage collection rule), where the constraints $LV(E_1, A.S, E_2)$ must hold for syntactic correctness, and $LV(A.S, E_2)\#(let E_1 in S)$ for semantic correctness.

Note that the extension of the algorithm in Section 5 assumes a ground language of garbage collected expressions.

3 Nominal Unification with Environments

In this section we construct a unification algorithm for equations and constraints over $NL_{letr}^{ASPE}$ where the intended input is a set of $NL_{letr}^{ASPE}$-equations with expression-variables and environment-variables only occurring linearly.

As data structure we use a set $\Gamma$ of (symmetric) equations between expressions, freshness constraints $\nabla$, and a substitution $\theta$.

Definition 3.1. A set of equations $\Gamma$ over $NL_{letr}^{ASPE}$ is admissible, if every environment-variable and expression-variable occurs at most once in $\Gamma$.

Definition 3.2. Let $Q = (\Gamma, \nabla)$ be a unification problem consisting of an admissible set $\Gamma$ of equations and freshness constraint $\nabla$. 

6
A ground substitution \( \rho \) is a solution, if \( Q \rho \) is ground, \( \nabla \rho \) holds and for all equations \( s = t \) in \( \Gamma \), the relation \( sp \sim t \rho \) holds.

The pair \((\Delta, \sigma)\) is a nominal unifier if for all ground substitutions \( \rho \) s.t. \( \Delta \rho \) holds, then \( \sigma \circ \rho \) is a solution of \( Q \).

A ground substitution \( \rho \) is an instance of the unifier \((\Delta, \sigma)\) of \( Q \), if \( \Delta \rho \) is valid and there is some ground substitution \( \gamma \) such that for all \( x \in \text{Var}(Q) \) of type \( A.S.P \): \((x)\sigma \circ \gamma \sim xp\), and for all \( E \in \text{Var}(Q) \): \((E)\sigma \circ \gamma \) is a permutation of \( E \rho \) modulo \( \sim \).

A set \( U \) of unifiers of \( Q \) is complete, if every solution of \( Q \) is an instance of some unifier in \( U \).

The idea of the following decomposition of letrec-expressions is to relate the syntactic components of the two environments by first guessing and then decomposing without losing solutions. For example, \( \text{letrec} \ E_1, X_1.S_1 \ in \ S_2 \equiv \text{letrec} \ Y_1,S'_1,Y_2.S'_2 \ in \ S'_1 \) has several possibilities for potential solutions: one example is that the instance of \( E_1 \) has bindings in common with \( E_2 \), and may contain instances of \( Y_1,S'_1,Y_2,S'_2 \). In a non-deterministic guessing, it is appropriate to first guess, which binding components or environment-variable suspensions of the left- and right-hand side have something in common. For example, \( E_1 R E_2, E_1 R Y_1,S'_1,X_1.S_1 R Y_2,S'_2 \) is a valid guess. The second part is to apply Lemma 2.1 and also to specify the introduced permutation-variable.

**Definition 3.3.** (Decomposing letrec.) Let \((\text{letrec} \ env_1 \ in \ e_1) \equiv (\text{letrec} \ env_2 \ in \ e_2)\) be the equation to be decomposed, where \( env_j \) for \( j = 1,2 \) consists of a list of bindings \( b_{j,i} \) and environment-variables \( E_{j,i} \). The decomposition is non-deterministic and proceeds as follows:

First there is a single guess of a relation \( R \) consisting of a set of pairs \((k_1, k_2)\) where \( k_j \) is a component of \( env_j \) for \( j = 1,2 \), i.e. a binding or an environment-variable suspension, such that

1. Every binding \( b_{1,i} \) is related to exactly one component of the right hand side.
2. Every binding \( b_{2,i} \) is related to exactly one component of the left hand side.
3. Every suspension \( \pi \cdot E \) is related to at least one component in the other environment.

Let \( P \) be a fresh permutation-variable. The resulting equations \( \Gamma_{\text{res}} \) are: Equations \( K \equiv P \cdot K' \) for binding components \( K, K' \) if these are related by \( R \) and \( e_1 \equiv P \cdot e_2 \).

The resulting substitution components \( \theta_{\text{res}} \) are as follows:

Create fresh environment-variables \( E_{i,j,k,h} \) that represent the intersection of \( E_{i,j} \) and \( E_{k,h} \). There are substitution components for every environment-variable suspension \( \pi_{1,j} \cdot E_{1,j} \) on the left hand side:

\[
E_{1,j} \rightarrow \pi_{1,j}^{-1}.P(E_{1,j,1}, \ldots, E_{1,j,2,m}, B_{2,j})
\]

for an appropriate \( m \) and where \( B_{2,j} \) are the single bindings on the right hand side that are related by \( R \) to \( \pi_{1,j} \cdot E_{1,j} \).

There are substitution components for every environment-variable on the right hand side:

\[
E_{2,j} \rightarrow \pi_{2,j}^{-1}.P^{-1}(E_{1,1,2,j}, \ldots, E_{1,m',2,j}, B_{1,j})
\]

for an appropriate \( m' \) and where \( B_{1,j} \) are bindings on the left hand side that are related by \( R \) to \( \pi_{2,j} \cdot E_{2,j} \).
(E1) \((\Gamma \cup \{e \leftarrow e\}, \nabla, \theta)\) \quad (E2) \((\Gamma \cup \{S \leftarrow e\}, \nabla, \theta)\) 
\((\Gamma \cup \{\pi \leftarrow S\}, \nabla, \theta)\) \quad (E3) \((\Gamma \cup \{A \leftarrow \pi_1 B\}, \nabla, \theta)\) 
\((\Gamma \cup \{\pi_2 A \leftarrow B\}, \nabla, \theta)\) 
(E4) \((\Gamma \cup \{f e_1 \ldots e_{\mathsf{ar}(f)} \leftarrow f' e_1' \ldots e_{\mathsf{ar}(f)}'\}, \nabla, \theta)\) 
(E5) \((\Gamma \cup \{W_1 e_1 \leftarrow W_2 e_2\}, \nabla, \theta)\) 
(E6) \((\Gamma \cup \{A_1, e_1 \leftarrow \lambda \pi_2 A_2 e_2\}, \nabla, \theta)\) 
(E7) \((\Gamma \cup \{\mathsf{letr} \ e_1 \mathsf{in} \ e_2 \leftarrow \mathsf{letr} \ e_2 \mathsf{in} \ e_2\}, \nabla, \theta)\) 

\(\text{guess according to Def. 3.3}\)

Fig. 1. Rules of the unification algorithm NomEnv1

The resulting constraints \(\nabla_{\mathsf{res}}\) are:

1. \(\mathsf{dom}(P) \subseteq \mathsf{LV}(\mathsf{env}_1) \cup \mathsf{LV}(\mathsf{env}_2)\).
2. \(P.(\mathsf{LV}(\mathsf{env}_2)) = \mathsf{LV}(\mathsf{env}_1)\)
3. \(\mathsf{LV}(\mathsf{env}_1) \# \mathsf{letr} \ \mathsf{env}_2 \ \mathsf{in} \ e_2\)

The effects of applying the rule is to remove the equation, and to add \(\Gamma_{\mathsf{res}}\):
\(\nabla_{\mathsf{res}}\) and \(\theta_{\mathsf{res}}\).

Note that the following properties follow from the constraints:

1. \(\mathsf{LV}(\mathsf{env}_2) \# \mathsf{letr} \ \mathsf{env}_2 \ \mathsf{in} \ e_2\)
2. \(P.(\mathsf{LV}(\mathsf{env}_1) \setminus \mathsf{LV}(\mathsf{env}_2)) = (\mathsf{LV}(\mathsf{env}_2) \setminus \mathsf{LV}(\mathsf{env}_1))\).

**Definition 3.4.** The algorithm NomEnv1 is defined by the rules in Fig. 1 on \(\Gamma, \nabla\), where the input \(\Gamma_0, \nabla_0\), does not contain permutation-variables, and the occurrences of environment-variables and of expression-variables in \(\Gamma\) are linear.

The set \(\nabla\) may contain constraints. Since binders in a letrec must be different, \(\nabla\) must also contain constraints which ensure for every environment \(\mathsf{env}\) that only valid instances are covered, i.e. \#\(\mathsf{LV}(\mathsf{env})\). It is assumed that the abstract representation of permutations uses a sharing structure for components (see e.g. [21]).

Note that intermediate \(\Gamma\)-s may contain (generated) permutation-variables.

**Example 3.5.** We illustrate the execution of the algorithm NomEnv1 on an example equation that occurs in unification problems related with a correctness proof of transformations in an extended lambda calculus with letrec, i.e. in the Haskell core-calculus LR [24].

The equation is \(\mathsf{letr} \ E_1 \ \mathsf{in} \ \mathsf{letr} \ E_2 \ \mathsf{in} \ S_1 \leftarrow \mathsf{letr} \ A.S_2; E_3 \ \mathsf{in} \ S_3\). A naive non-general solution would be: \(E_1 \leftarrow A.S_2; E_3, S_3 \leftarrow \mathsf{letr} \ E_2 \ \mathsf{in} \ S_1\) plus
constraints for syntactic validity of the represented instances, and for avoiding capture of free variables.

Application of the algorithm NomEnv1D yields:

1. The substitution \( \theta = \{ E_1 \mapsto P \cdot (A.S_2, E_{13}), E_3 \mapsto P^{-1} \cdot E_{13}, S_3 \mapsto \text{let} \ P \cdot E_2 \mathrm{in} \ P \cdot S_1 \} \)

2. Additional constraints \( \nabla \) restricting the permutation \( P \), i.e.

\[
\nabla = \begin{cases}
\text{dom}(P) \subseteq LV(E_1, E_3, A), \\
P \cdot LV(A, E_3) = LV(E_1), \\
LV(E_1)\#(\text{let} \ A.S_2; E_3 \mathrm{in} \ S_3)
\end{cases}
\]

Based on this most-general unifier we can see what was missing from the naive result above. It did not take possible renamings of binding variables into account and therefore did not cover all possible solutions modulo \( \alpha \)-equivalence of the unification problem. The possible renamings are covered in the general solution by the permutation-variable \( P \).

**Example 3.6.** We give a further example of the use of nominal unification. We overlap the left hand side, i.e., \( \text{let} \ E_1 \mathrm{in} \text{let} \ E_2 \mathrm{in} \ S \), of the first transformation in Example 2.5 with a (renamed) subexpression of itself, \( \text{let} \ E_2 \mathrm{in} \ S \Leftrightarrow \text{let} \ E_1' \mathrm{in} \text{let} \ E_2' \mathrm{in} \ S' \), which results in a solution: \( \{ E_2 \mapsto P \cdot E_1'; S \mapsto P \cdot \text{let} \ E_2' \mathrm{in} \ S' \} \). The common instance is \( \text{let} \ E_1 \mathrm{in} \text{let} \ P \cdot E_1' \mathrm{in} \text{let} \ P \cdot E_2' \mathrm{in} \ P \cdot S' \). This reduces in two ways, using the rule as reduction and transformation rule, to \( \text{let} \ E_1, P \cdot E_1', P \cdot E_2' \mathrm{in} \ P \cdot S' \). However, we also have to add the constraint sets for the two sequences that are mentioned Example 2.5. These are at least: \( LV(E_2)\#(\text{let} \ E_1 \mathrm{in} \ S) \), and \( LV(E_2')\#(\text{let} \ E_1' \mathrm{in} \ S') \). The arguments for showing confluence in the style of Knuth Bendix require a deeper analysis and are left for future work.

**Theorem 3.7.** If \( \Gamma \) is admissible and does not contain permutation-variables then the algorithm NomEnv1 is terminating, sound and complete; i.e. for every solution the algorithm computes a unifier consisting only of a substitution \( \theta \) and a constraint \( \nabla \). A single run takes polynomial time (provided the implementation uses sharing). The collection version of the algorithm will generate at most exponentially many unifiers.

However, the algorithm NomEnv1 does not automatically decide solvability, since it is possible that no constraint set can be satisfied and hence all computed unifiers have an empty set of instances. We also refrain from providing an algorithm, which decides solvability of the constraint system itself, because the many variable kinds (\( A,S,E \) and \( P \)-variables) make reasoning on it very difficult. This is (partially) remedied in the next section, where we construct a special (incomplete) variant NomEnv1D for nominal unification, which decides unifiability. We will show that the algorithm NomEnv1D will find a (small) solution if there is any solution, where in addition the constraints do not contain \( E \)-variables, and instead of \( P \)-variables we use the \( \xi \)-construct (see below in Def. 4.2) that replaces permutation-variables and keeps more information on the mapping behavior of the permutations.
(RemoveEl)
\[(\Gamma \cup \{ \text{let } W_1.s_1; \ldots; W_k.s_k; E_1 \in s \}) \setminus \{ \text{let } W'_1.s'_1; \ldots; W'_{k'}{.s'}_{k'} \in s' \}\), \nabla, \theta)\]

(\Gamma \cup \{ \text{let } W_1.s_1; \ldots; W_k.s_k, A_1.S_1; \ldots; A_{k'}.S_{k'} \in s \}
= \text{let } W'_1.s'_1; \ldots; W'_{k'}.S'_{k'} \in s' \), \nabla \theta_E, \theta \circ \theta_E)\]

where \( k' \geq k \). Let \( \theta_E = \{ E_1 \mapsto A_1.S_1; \ldots; A_{k'}.S_{k'} \} \). \( A_1, S_1 \) are fresh.

(A symmetric variant of RemoveEl is omitted since \( = \) is symmetric.)

(RemoveElr)
\[(\Gamma \cup \{ \text{let } W_1.s_1; \ldots; W_k.s_k; E_1 \in s \}
= \text{let } W'_1.s'_1; \ldots; W'_{k'}.S'_{k'} \in s' \), \nabla, \theta)\]

\[(\Gamma \cup \{ \text{let } W_1.s_1; \ldots; W_k.s_k, A_1.S_1; \ldots; A_{k'}.S_{k'} \in s \}
= \text{let } W'_1.s'_1; \ldots; W'_{k'}.S'_{k'} \in s' \), \nabla \theta_E, \theta \circ \theta_E)\]

where \( k' \geq k \). Let \( \theta_E = \{ E_1 \mapsto A_1.s_1; \ldots; A_{k'}.s_{k'} \} \) and \( A_1, S_1 \) are fresh,
and \( k' + h' = k + h = k + k' + N_2; \) where \( N_2 = |AtPos(s_1, \ldots, s_{k'}, s'_1, \ldots, s'_{k'})| \)

(RemoveE) is defined as the union of RemoveEl and RemoveElr.

4 A Decision Algorithm

In this section we define a decision algorithm for admissible problems, complementing the algorithm NOMENV1. We want to keep the description simple and also as close as possible to potential applications. Thus we describe the decision algorithm for the simpler case that in letrec environments at most one environment-variable occurs. The advantage is that this is the variant which is required in most applications, and in addition the rule RemoveE (see Fig. 2) is deterministic. The decidability result also holds in the general case where several environment-variables occur in a single letrec-environment. (See Appendix for hints on a generalization.)

Definition 4.1. Let \( \Gamma \) be an admissible set of equations. We say \( \Gamma \) is a 1E-problem, if in every letrec-environment in \( \Gamma \), there is at most one environment-variable.

A special kind of permutation-variable is introduced by the rule Decompllet. These variables are so strongly restricted by constraints, that one can always provide an explicit list of swappings instead.
Hence we introduce an extra notation to avoid explicit permutation-variables. The benefit of this variant is that no new constraint concepts are required and thus it is compatible with [3].

**Definition 4.2.** Let $W_1, W_2, \ldots, W_k, W'_1, W'_2, \ldots, W'_k$ be $W$-expressions, where only instantiations are considered that satisfy the constraints $\#\{W_1, W_2, \ldots, W_k\}$, and $\#\{W'_1, W'_2, \ldots, W'_k\}$. Then we denote with $\xi((W_1, W_2, \ldots, W_k), (W'_1, W'_2, \ldots, W'_k))$ the permutation instances that obey the following: $W_1 \mapsto W'_1, \ldots, W_k \mapsto W'_k$ and the domain of $\xi$ is contained in $\{W_1, \ldots, W_k, W'_1, \ldots, W'_k\}$. This does not completely define the permutation, but the omitted parts will have no effect when used in our algorithms, when it is applied, due to further freshness constraints. The operations with $\xi$ are the same as for permutations, when applied on the expression level.

Note that there may be more than one permutation that could be $\xi((W_1, W_2, \ldots, W_k), (W'_1, W'_2, \ldots, W'_k))$ (see the remark after Lemma 2.1).

**Lemma 4.3.** A representation for one permutation that is $\xi((W_1, W_2, \ldots, W_k), (W'_1, W'_2, \ldots, W'_k))$ as a list of swappings is given by the following recursion scheme: $\pi_0 = \emptyset$; $\pi_i = ((\pi_{i-1} \circ W_i) \circ W'_i) \circ \pi_{i-1}$.

More precisely, given $\nabla \supseteq \{\#\{W_1, \ldots, W_n\}, \#\{W'_1, \ldots, W'_n\}\}$ – which has to hold due to Definition 4.2 – every solution of $\nabla$ requires $\pi_i$ mapping $W_j \mapsto W'_j$ for all $j \leq i \leq k$.

Note that the plain size of this representation is exponential due to iterated doubling of $\pi_{i-1}$. If we use sharing for $\pi_{i-1}$, then the representation is of polynomial size.

**Definition 4.4.** The function $\text{AtPos}(e)$ is the set of all atom-variable suspensions ($W$-variables) in $e$.

**Definition 4.5.** The (non-deterministic) algorithm $\text{NomEnv1D}$ is defined for $\text{1E}$-problems and uses the rules in Fig. 1 on the input $\Gamma_0, \nabla_0$, with the exception of $E7$, and the rules $\text{RemoveE}$ and $\text{DecompLet}$ in Fig. 3, resulting in $\Gamma', \nabla', \theta'$. The final test is whether $\Gamma' = \emptyset$ and whether $\nabla'$ under $\theta'$ is satisfiable using the constraint-test $\text{NomEnv1DCon}$ below.

The answer is “yes”, if one run of the algorithm answers “yes, solvable”.

Note that the rules can be applied in any order. All rules are deterministic with the exception of $\text{DecompLet}$, which requires a guess on the permutation of the let-variables.

Note also that the substitution $\theta$ is intended to be an instantiation of the input problem. The necessary instantiations of the current $\Gamma, \nabla$ are done by the rules. However, it is necessary to assume a directed graph implementation of expressions in order to exploit sharing, in particular in the representation of permutations.

**Definition 4.6.** The final constraint-test $\text{NomEnv1DCon}$ i.e., whether the final constraint $\nabla', \theta'$ is satisfiable is as follows:
1. Let $A_0$ be a set of atoms of cardinality $|\Gamma_0|$, where $\Gamma_0$ is the original input to the algorithm.
2. Guess for all atom-variables their mapping to atoms in $A_0$.
3. Guess for all expression-variables their set of free atoms in $A_0$.
4. Check the freshness constraints: This can be done in polynomial time. The algorithm has to respect the sharing by directed graph implementation.

Note that NOMEnv1D is not complete w.r.t. solutions, since by intention, its rules do not cover all solutions. However, it is decision-complete, i.e. sufficient for a decision algorithm, since the algorithm will find a (small) solution if there is one at all.

Example 4.7. It is helpful to look at examples for the actions of NOMEnv1D. Let the equation be \texttt{let} $E$ in $A$ = \texttt{let} $E'$ in $B$ and $\nabla = \{ A \# B \}$. The rule \texttt{RemoveE} is not permitted to instantiate $E, E'$ with the empty environment, since there are 2 atom-position which must possibly be bound. The algorithm then instantiates both environment variables with 2 bindings, i.e. $\theta = \{ E \mapsto \{ A_1, S_1, A_2, S_2 \}, E' \mapsto \{ B_1, T_1, B_2, T_2 \} \}$ with appropriate constraints. Using $\rho = id$ we get $A \equiv \xi \cdot B$ which we can move into the constraints as $A = \# \xi \cdot B$ and $S_i = \xi \cdot T_i$. The final satisfiability check yields true.

Proposition 4.8. The rule \texttt{RemoveE} is correct/decision-complete. I.e. if the algorithm NOMEnv1D is in the state $(\Gamma, \nabla)$ and the output of \texttt{RemoveE} is $(\Gamma', \nabla')$, then $\Gamma', \nabla'$ is solvable if and only if the input $\Gamma, \nabla$ is solvable.

Lemma 4.9. All rules of NOMEnv1D are correct and decision-complete, where only \texttt{DecomplLet} is non-deterministic.

Theorem 4.10. Given an admissible nominal unification problem $\Gamma, \nabla$ without permutation-variables such that $\Gamma$ is $1E$. Then the Algorithm NOMEnv1D is a decision algorithm, which runs in NP time, under the assumption that sharing is used in the representation of expressions and permutations.

Proof. Proposition 4.8 and Lemma 4.9 show that the rules are correct and decision-complete. If $\Gamma$ is non-empty, then there is a rule that is applicable. Every application of a rule makes $\Gamma$ smaller, which can be seen by using the following measure: (i) The number of symbols \texttt{let}, $\lambda$, $E$-variables, binding-dot; (ii) the number of equations in $\Gamma$. This holds, since every rule strictly reduces this measure, and since the occurrences of environment- and expression-variables is linear.

The algorithm can be performed in nondeterministic polynomial time, due to this measure, and since sharing ensures polynomial size, and since the evaluation of constraints can be done in polynomial time also for the sharing structure.

From the previous theorems and since a subproblem is already NP-hard [18]:

Corollary 4.11. Solvability of admissible nominal unification problem $\Gamma, \nabla$ without permutation-variables and such that $\Gamma$ is $1E$ is NP-complete.
5 Nonlinear Occurrences of Expression-Variables

In this section we sketch the generalization of our method to input problems where a single expression-variable may occur more than once (i.e., non-linear), and where letrec-environments contain at most one environment-variable. An example is the right hand side of the second rule in Example 2.5. In addition, we use as semantics only ground expressions that are garbage-free.

With this restriction, our algorithmic ideas can be adjusted to non-linear occurrences of expression-variables and the restriction itself is minor with respect to functional programs, since in the application domain garbage does not contribute to the proper actions of programs. However, the algorithm NomEn-vNS1E und the data structures have a more complex description.

We start by providing a definition of garbage-free expressions. A ground expression \( e \) is garbage-free if in every subexpression \( \lettr{env}{in}{s} \) and for every proper subenvironment \( env' \) of \( FV(\lettr{env'}{in}{s}) \supset FV(\lettr{env}{in}{s}) \) holds. Note that \( \supset \) refers to a proper superset.

A nice property of expressions \( e \) in \( NL_{\text{ASPE}} \) with garbage free ground language is that fixpoint equations of the form \( \pi \cdot e = e \) can be expressed as a set of freshness constraints: \( \{ A \# \lambda \pi.\pi \cdot e \mid A \text{ occurs in } \pi \} \) as in [19, 23]. Information and results on extended alpha-equivalence and garbage-free expressions is in [21], which shows that the graph-structure is kept by alpha-equivalence. This implies the claimed representation.

The data structure for the nominal unification algorithm in this case are multi-equations instead of equations [12]. We write a multi-equation that equates the expressions \( e_1, \ldots, e_n \) as set \( \{ e_1, \ldots, e_n \} \). Of course, a set of equations is a special set of multi-equations.

**Standard rules** for handling multi-equations are:

1. As a standard prerequisite, all expressions must be in a so-called flat form: deep subexpressions \( C[e] \ldots \) are represented as \( \ldots C[S] \ldots \) and an equation \( S = e \) for a fresh expression-variable \( S \).

The permitted elements of the multi-equations after the exhaustive flattening operation are of the form \( \pi \cdot S \mid \lambda W.\pi \cdot S \mid (f \pi_1 \cdot S_1 \ldots \pi_n \cdot S_n) \mid (\lettr{W_1}{\pi}{S_1}, \ldots, W_n.\pi \cdot S_n, \pi_{n+1} \cdot E_1, \ldots, \pi_{n+k} \cdot E_k \text{ in } \pi \cdot S) \).

2. Treat multi-equations as sets, i.e. remove duplicates.

3. Merge: if two multi-equations have a common expression (a common variable), then replace them by their union.

4. Single: if a multi-equation consists of a single expression, then remove it.

**Further standard rules** for handling multi-equations are:

1. A multi-equation that only contains suspensions of atom-variables can be moved to the constraints, since atom-variable equations can be represented as constraints.

2. A multi-equation that is of the form \( \{ \pi \cdot S, e_1, \ldots, e_n \} \) can be changed into \( \{ S, \pi^{-1} \cdot e_1, \pi^{-1} \cdot e_2, \ldots, \pi^{-1} \cdot e_n \} \) by applying the permutation \( \pi^{-1} \).
3. A multi-equation that is of the form \{S, e\}, where \( S \) is not contained in \( e \) or in other expressions in \( \Gamma \) and where \( e \) is not a suspension of \( S \), can be removed, and \( S \mapsto e \) moved to the substitution.

**Fixpoint Rules** for handling multi-equations are:

**Fixpoints** A multi-equation \( M \) that contains two expressions \( S \) and \( \pi \cdot S \), where \( \pi \) is nontrivial is modified to \( M \setminus \{\pi \cdot S\} \), and the constraints \( \{\lambda \pi \cdot A. S \mid A \text{ occurs in } \pi\} \) are added.

**Fixpoint-Chains** If there is a chain \( S_1 \mapsto \pi_2 \cdot S_2 \mapsto M_1, S_{n-1} \mapsto \pi_n \cdot S_n \mapsto M_{n-1}, \ldots, S_n \mapsto \pi_1 \cdot S_1 \mapsto M_n \), then using the derivable equation \( S_1 \mapsto (\pi_2 \cdot \ldots \cdot \pi_n \cdot \pi_1) \cdot S_1 \), let \( \pi = (\pi_2 \cdot \ldots \cdot \pi_n \cdot \pi_1) \); then the constraints \( \{\lambda \pi \cdot A. S \mid A \text{ occurs in } \pi\} \) are added. Also, \( S_1 \) is removed from the first multi-equation.

### 5.1 Decomposition Rules

**Non-unifiability.** For a better understanding of the rules, it is clear that the following equations lead to an empty set of solutions, and hence do not need extra rules: The following can pairwise not be equated: atom suspension, application-expression, lambda-expression, letrec-expression. Hence, we can restrict attention to a situation where multisets are only of the following five forms, where every multiset can contain additional \( \pi \cdot S \)-expressions: (i) empty, (ii) atom suspension, (iii) application-expression, (iv) lambda-expression, and (v) letrec-expression.

We need only decomposition rules for these types of multi-equations. There is also a (standard) cycle check as generalized occurs-check.

**Decomposition Rules for Compound Expressions** The decomposition rules are analogous to the rules for the equation-based algorithm.

The general principle to adapt the rules of the algorithm \texttt{NomEnv1E} to \texttt{NomEnvNS1E}, i.e., to multiset is as follows:

If \( e_1 \mapsto e_2 \) is decomposed with a result, then the corresponding rule \( R \) for multisets is:

\[ \{e_1, e_2\} \cup M \text{ is replaced by } \{e_2\} \cup M \text{ and the result of } R \text{ is added to the corresponding component of the equations or the constraints and solution.} \]

The final result of the algorithm is a set of constraint.

We observe that the multiset-algorithm does never duplicate or generate expressions in the equation set of the three forms: application-expression, lambda-expression, and letrec-expression. Hence the number of occurrences of environment-variables remains linear during the run of the algorithm. Soundness and completeness are mainly derived from the arguments for the \texttt{NomEnv1E} algorithm.

Complexity estimation of the modified algorithm starting with the a flattened input:
1. The number of letrec-s, lambda-expressions and application-expressions does not increase. These numbers are strictly decreased by decomposition rules.
2. Multi-equations of type empty can be removed with a constant increase: at most the maximal arity of applications and letrec-components in the equations.
3. Multi-equations of type lambda, application and letrec can be decomposed with a constant increase per decomposition application.
4. Permutation components can grow exponentially large due to iterated doubling. However, a sharing data structure and clever maintenance leads to a polynomial size increase.

This sums up to a polynomial complexity for a single run, and to a non-deterministic polynomial algorithm for computing unifiers.

**Claim:** The algorithm \textsc{NomEnvNS1E} is sound and complete and runs in \textsc{NP} time.

However, the adaptations of the decision algorithm in this case are not obvious: The reason is that the same \(S\)-variable may occur multiple times and hence there may be interferences.

We conjecture that there is an exponential upper bound on the maximal number of bindings that have to be generated in the rule adapted from \textsc{RemoveE}.

We also conjecture that the computational complexity of the decision problem without restrictions on the number of occurrences of \(E\)-variables is strictly higher than for the case with linear occurrences of \(E\)-variables.

## 6 Conclusion and Future Work

Nominal unification of letrec-expressions is extended to also allow abstract environments in letrec-environments, which are encoded as environment-variables. An algorithm for computing a finite set of unifiers as well as a decision algorithm are described and proved correct, under linearity constraints for expression- and environment-variables. Nonlinear occurrences of expression-variables are also sketched where the semantics are garbage-free ground expressions. The approach and the algorithms have a high potential of improving automated tools for reasoning about program transformations in higher languages with recursive let.

For the generalization to nonlinear occurrences of environment-variables, we did not find a terminating nominal unification algorithm. We conjecture the decision problem to be strictly harder in this case.

Our work provides also the base for an extension of the rewriting mechanisms and confluence checks in [9, 8] to a setting, where letrec-expressions, environment-variables and expression-variables are allowed, and where also context-variables as in [22] are permitted.

This would enable the study of the rules of functional (in particular call-by-need) calculi and transformations, by checking the overlaps between them. In fact, several of these rules and transformation have abstract variables for parts of the environment, where usually the occurrences of the environment-variables are linear, yielding a natural field of applications for the work of this paper.
Bibliography


A Proof of Lemma 2.1

Proof. Assume $e_1 \sim e_2$. Since the notation is symmetric, we can assume that the permutation $\rho$ on $\{1, \ldots, n\}$ is the identity, hence that the alpha-equivalence relates $a_i.s_i$ with $b_i.t_i$ for all $i$ and and $r$ with $r'$ via a renaming sequence for $e_2$. Since binding names in an environment are all different, let $\varphi_0$ be the function $\{t_1 \mapsto a_1, \ldots, t_n \mapsto a_n\}$. We can make this a permutation by extending it by an arbitrary bijection $\{a_1, \ldots, a_n\} \setminus \{t_1, \ldots, t_n\} \mapsto \{t_1, \ldots, t_n\} \setminus \{a_1, \ldots, a_n\}$, resulting in a permutation $\varphi$ with domain $\{a_1, \ldots, a_n\} \cup \{t_1, \ldots, t_n\}$. Since a name-permutation does not change the $\sim$-equivalence, we obtain $s_i \sim t_i \varphi$, and $r \sim r' \varphi$. We have to argue that for $M := \{a_1, \ldots, a_n\} \setminus \{t_1, \ldots, t_n\}$, we have $M \#(\texttt{let } b_1.t_1, \ldots, b_n.t_n \texttt{ in } r')$. Assume that some atom $a \in M$ occurs free in $(\texttt{let } b_1.t_1, \ldots, b_n.t_n \texttt{ in } r')$. Then $\varphi(a) \in \{b_1, \ldots, b_n\}$, which implies that $\varphi(a) \notin \{b_1, \ldots, b_n\}$ is free in $\texttt{let } a_1.s_1, \ldots, a_n.s_n \texttt{ in } r$, which, however, contradicts the relation $\sim$, since $\varphi(a)$ is not free in $e_1$.

The other direction is to show that the four conditions imply $e_1 \sim e_2$. These conditions imply that also $\texttt{let } E_1; E_2 \texttt{ in } 0 \sim \texttt{let } a.E_3 \texttt{ in } 0$ where $a \neq b$, and let $\nabla = \{a \neq E_1\}$. There is an instantiation $\sigma_1$ with $E_1 \mapsto b.a, E_2 \mapsto 0; E_3 \mapsto 0$, which is not a solution to the constraint, and another one, $\sigma_2$, with $E_2 \mapsto b.a, E_1 \mapsto 0; E_3 \mapsto 0$ that is a solution, since the constraint is satisfied.

The latter solution cannot be covered by a general solution that instantiates $E_2$ by the empty environment. The conclusion is that all possibilities for the concrete bindings for $E_1, E_2$, like $a.b$, have to be tried, which makes the general rule nondeterministic.

Remark B.2 (Decision Algorithm for the General Case). We concentrate on the case that multiple environment-variables $E_1, \ldots, E_n$ are in a letrec environment. The generalization is that the rule(s) RemoveE has to try all possibilities of distributing the single bindings into the environment-variables. This makes the
generalized rule(s) RemoveE non-deterministic. It is obvious how to transfer the correctness proof to this case.

This nondeterminism could be omitted if the environment-variables are not restricted by freshness constraints, or if the freshness constraints contain them such that the environment-variables cannot be distinguished. This determinism can be achieved by selecting a single environment-variable per environment, say $E_1$, and put all instantiation into $E_1$, end instantiate other environment-variables by $\emptyset$.

**Theorem B.3.** Given an admissible nominal unification problem $\Gamma$ without permutation-variables, then the Algorithm NomEnv1D is a decision algorithm, which runs in NP time.

**Proof.** Lemma 9 shows that if there is a solution of the input, then the Algorithm NomEnv1D can reach a final state with empty $\Gamma_{fin}$ and $\nabla_{fin}$, such that there is a solution of the final state.

We estimate the size of the final state: The rules strictly decrease $\Gamma$, where we first ignore the size of permutations. Even $E2$ strictly decreases it, since there are at most 2 occurrences of $S$, and the increase by instantiation is strictly smaller than the decrease by removing the equation. Rule $E7$ also decreases the size, since at least the letrec construct is removed.

Finally, the size of the permutations is also polynomial, since the size increase is either constant or there is only a shift, but no duplication. Furthermore the conditions imply that fixpoint-equations for expression-variables do not occur during the algorithm.

Since the number of steps is polynomial, and the size increase is also polynomial, the (non-deterministic) rule applications run in NP time.

It remains to argue that the constraints can be solved in NP time: this holds, since we can guess the sets of active atoms by first guessing equality / disequality of atom-variables; and then the mapping behaviour of permutation-variables. Finally, we can evaluate the constraints in polynomial time.

**C Proof of Proposition 4.8**

**Proof.** (of Proposition 4.8)

It is easy to see that the rule is correct and deterministic. We will show that it is decision-complete:

Let $\sigma$ be a solution of $(\Gamma, \nabla)$ before application of the rule RemoveE. Let us assume that the equation between the two letrec expressions contains each an environment-variable, such that RemoveElr will be applied. The case of RemoveEl is similar and thus omitted. We show in several steps that there is also a solution $\sigma'$ that instantiates the environment-variables in a restricted way. Let the equation $eq_1$ be $\text{letr } E_1; env_{1,B} \text{ in } r_1 \equiv \text{letr } E'_1; env'_{1,B} \text{ in } r'_1$ where $env_{1,B}, env'_{1,B}$ consist only of bindings.
We show that we first can partly apply instantiations to the equation, such that there is also a solution $\sigma_1$, and the alpha-equivalence only relates bindings with bindings: The instantiation is such that $E_1$ is instantiated by an environment expression of the form $A_1.S_1; \ldots ; A_k.S_k; E_2$, and $E_1'$ by an environment expression of the form $A_1'.S_1'; \ldots ; A_k'.S_k'; E_2'$, where all atom-variables $A_j, A_j'$ and expression-variables $S_j, S_j'$ are fresh. The result is the equation $eq_2$.

The argument is that the solution $\sigma$ can be adapted to the resulting equation $eq_2$, giving $\sigma'$ as follows. Since $eq_1 \sigma$ is a valid $\alpha$-equivalence, there is a (bijective) relation $R$ between the two environments of the letrec-components. This relation also holds after the instantiation of $eq_2$ by $\sigma'$, where the substitution $\sigma'$ has to instantiate $A_i, S_i$, and $A_i'.S_i'$ accordingly. If there are more bindings in the instance, then these remain in the instance of $E_2, E_2'$. If there are less bindings in the instance, then we can use dummy instantiations for $A_j, S_j, A_j'.S_j'$, but such that the $\alpha$-equivalence holds for the instance.

Note that the constraints are not changed by this substitution. Now we can assume that $\sigma'$ is a solution of $eq_2$ such that there is a bijective relation $R_2$ on the bindings and bindings in $eq_2$ on the left hand side are only related to bindings on the right hand side and vice versa. In particular, the number of bindings is the same for the left and right hand side.

The next step is to prepare a cut of parts of $\sigma'$ that only permits a bounded number of single bindings as instantiations for the environment-variables.

Let $M_{\text{left}}$ be the set of atoms in the left hand side of $eq_2 \sigma'$ that occur in the instance of bindings or in the in-expression, and that in addition are results of $W\sigma'$ for a $W$-expression in the left hand side (at any position) of $eq_2$, and let $M_{\text{right}}$ be the same for the right-hand side.

Now we can define restriction $\sigma''$ of $\sigma'$: The following instantiations into the $E$-variables of the top environments are kept: On the left hand side, the bindings that bind an atom in $W_{\text{left}}$, and in the right hand side the bindings that bind an atom in $M_{\text{right}}$. Due to the linearity of expression-variables, they cannot introduce new $W$-expressions, which need to be bound. We also do not add further environment-variables.

Furthermore, also the bindings are kept that are related by $R$ to the kept bindings. All other bindings are omitted. Let this substitution be $\sigma''$. Unfortunately, this is not yet a solution: Due to the latter omission, there may be positions $Q$ of atoms, that were bound before the omission but unbound after the omission. All these atoms and their positions are introduced by $\sigma''$. The last modification is to select a single fresh atom $c$, which replaces all the atoms of the unbound positions in $Q$ by $c$. This leads to the substitution $\sigma'''$.

We also have to construct on the abstract level another equation $eq_3$ by instantiating the environment-variables exactly with environments of the form $A_1.S_1; \ldots ; A_h.S_h$, and $A_1'.S_1'; \ldots ; A_h'.S_h'$, (with fresh names, and the same $h$, such that no environment-variables remain, and such that there is an instantiation $\sigma''$ of $eq_3$ that is the same as $eq_2 \sigma'''$.)

20
Note that the two modifications of $eq$ are covered by the rule $\text{RemoveElr}$ by guessing sufficiently many abstract bindings for instantiation.

(III) We claim that $\sigma''$ is a solution of $eq_2$. We also claim that $\sigma^{(4)}$ is a solution of $eq_3$. Both claims follow from the construction.

(IV) We have to estimate the number of binding components that maximally have to be guessed. We had to add at most $k + k'$ single bindings in both top environments, plus the number $N_1$ of positions of $W$-expressions in $eq$, i.e. $\text{AtPos}(eq)$. The reason is that only atom-variables which were not already present in the unification problem could be omitted from the domain of $\sigma$.

(V) We need to argue that the freshness constraints still hold. This is the case, since “visible” atoms are treated in the same way as for the solution $\sigma$. Furthermore there are only freshness constraints, and we only remove unbound atoms and do not add unbound atoms in the expressions in $\nabla$. Hence all freshness constraints still hold.

D Proof of Theorem 3.7

Proof. This proof depends on Lemmas 2.2 and 2.1 that tell us that the general algorithm will follow all possibilities of solutions, and hence is complete. Termination holds, since every step makes $\Gamma$ strictly smaller, if we only count the term symbols, and ignore the permutations. This measure also implies the polynomial runtime of a single run, where we have to take into account the sharing of the data structures.

E Proof of Lemma 4.3

Proof. We use induction to show that $\pi_i$ satisfies the requirements. The base case $i = 0$ is clear.

Now, consider $\xi((W_1, W_2, \ldots, W_{i+1}), (W'_1, W'_2, \ldots, W'_{i+1}))$. By the induction hypothesis $\pi_k$ maps $W_i \mapsto W'_i$ for $i \leq k$ and $\pi_{i+1} = ((\pi_i \circ W_{i+1}) W'_{i+1}) \circ \pi_i$.

We need to show that 1) $\pi_{i+1}$ maps $W_{i+1} \mapsto W'_{i+1}$ and 2) for $j \leq i$: the permutation $\pi_{i+1}$ maps $W_j \mapsto W'_j$.

The first item follows from construction.

\[
\begin{align*}
\pi_{i+1} \circ W_{i+1} \\
= ((\pi_i \circ W_{i+1}) W'_{i+1}) \circ \pi_i \circ W_{i+1} \\
= ((\pi_i \circ W_{i+1}) W'_{i+1}) \circ (\pi_i \circ W_{i+1}) \\
= (W'_{i+1} W'_{i+1}) \circ W_{i+1} \\
= W_{i+1}
\end{align*}
\]

(since $\pi_i$ does not change $W_{i+1}$)

For the second point, note that from the induction hypothesis for $j \leq i$ it follows:

\[
\begin{align*}
\pi_{i+1} \circ W_j = ((\pi_i \circ W_{i+1}) W'_{i+1}) \circ \pi_i \circ W_j
\end{align*}
\]

\[
= ((\pi_i \circ W_{i+1}) W'_{i+1}) \circ W'_j
\]
From $\nabla$ it follows $W_j' \neq W_{i+1}'$. Suppose $W_j' = \pi_i \circ W_{i+1}$. Then $\pi_i^{-1} \circ W_j' = W_{i+1}$ and by induction hypothesis $\pi_i^{-1} \circ W_j' = W_j = W_{i+1}$. This is ruled out by $\nabla$. Therefore $W_j' \neq W_{i+1}'$ and $W_j' \neq \pi_i \circ W_i+1$ follow from $\nabla$ and the hypothesis. Hence we obtain $\pi_{i+1} \circ W_j = ((\pi_i \circ W_i+1) \ W_i+1) \circ W_j' = W_j'$. 
Terminating Non-Disjoint Combined Unification

Serdar Erbatur\textsuperscript{1}, Andrew M. Marshall\textsuperscript{2}, and Christophe Ringeissen\textsuperscript{3}

\textsuperscript{1} University of Texas at Dallas, USA
\textsuperscript{2} University of Mary Washington, USA
\textsuperscript{3} Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

\textbf{Abstract.} The equational unification problem, where the underlying equational theory may be given as the union of component equational theories, appears often in practice in many fields such as automated reasoning, logic programming, declarative programming, and the formal analysis of security protocols. In this paper, we investigate the unification problem in the non-disjoint union of equational theories via the combination of hierarchical unification procedures. In this context, a unification algorithm known for a base theory is extended with some additional inference rules to take into account the rest of the theory. We show theories for which a hierarchical approach is applicable and present a simple form of hierarchical unification procedure. The approach is particularly well-suited for any theory where a unification procedure can be obtained in a syntactic way using transformation rules to process the axioms of the theory. Hierarchical unification procedures are exemplified with various theories used in protocol analysis. Next, we look at modularity methods for combining theories already using a hierarchical approach. In addition, we consider a new complexity measure that allows us to obtain terminating (combined) hierarchical unification procedures.

\section{Introduction}

Unification is a critical tool in many fields such as automated reasoning, logic programming, declarative programming, and the formal analysis of security protocols. For many of these applications we want to consider equational unification, where the problem is defined modulo an equational theory $E$, such as Associativity-Commutativity. For example, one approach to the analysis of security protocols is based on deductive reasoning, as is done in the following tools \cite{6,18,25}. In this approach protocols are usually represented by clauses in first-order logic with equality and equational theories are used to specify the capabilities of an intruder \cite{1}. To support this reasoning approach we need to use $E$-unification procedures. Since equational unification is undecidable in general, specialized techniques have been developed to solve the problem for particular classes of equational theories, many of high practical interest. For instance, when the equational theory $E$ has the Finite Variant Property (FVP) \cite{11,19}, there exists a reduction from $E$-unification to syntactic unification via the computation of finitely many variants of the unification problem. The class of equational
theories with the FVP has attracted a considerable interest since it contains theories that are crucial in protocol analysis [19,37,12,20].

Another ubiquitous scenario is given by an equational theory $E$ involved in a union of theories $F \cup E$. To solve this case, it is quite natural to proceed in a modular way by reusing the unification algorithms available for $F$ and for $E$. There are terminating and complete combination procedures for signature-disjoint unions of theories [29,3]. However, the non-disjoint case remains a challenging problem. One approach to the non-disjoint combination problem that has been successful in some cases is the hierarchical approach [14]. In this approach, $F \cup E$-unification can be considered as a conservative extension of $E$-unification. Then, a new inference system related to $F$, say $U_F$, can be combined with an $E$-unification algorithm to obtain an $F \cup E$ unification algorithm. While this hierarchical approach won’t work for every $F \cup E$ it can be a very useful tool when applicable. However, up to now it could be complex to know if a combination $F \cup E$ could be solved via the hierarchical approach. For example, there is no general method for obtaining the inference system $U_F$, and the resulting hierarchical unification procedure may not terminate.

In this paper, we consider “syntactic” theories $F \cup E$ where $U_F$ can be defined as a system of mutation rules, and we present new terminating instances of the hierarchical unification procedure. When an equational theory fulfills the syntacticness property [22,28], there exists a rule-based unification procedure in the same vein as the one known for syntactic unification, which is called a mutation-based unification procedure. Unfortunately, being syntactic is not a sufficient condition to ensure the termination of this mutation-based unification procedure. However, terminating mutation-based unification procedures are known for some particular theories such as one one-side distributivity [30,24], distributive exponentiation theories [15], shallow theories [10] and theories closed by paramodulation [23]. All the theories investigated here using the hierarchical approach are both syntactic and finitary: each of them is actually a syntactic theory for which a (finitary) unification algorithm is shown. On the one hand, we study theories which are both collapse-free and finitary, that is, finitary theories defined by axioms between non-variable terms. These theories are known to be syntactic [22]. On the other hand, we also examine forward-closed theories that are known to be both syntactic and finitary, just like theories closed by paramodulation [23]. The forward-closed theories we are interested in are actually examples of theories having the Finite Variant Property.

The contributions of the paper consist of several improvements to the hierarchical combination method [14,13] including: simplifying the method, clarifying the theories for which the approach is applicable, and reducing some of the restrictions. Furthermore, we develop several new results including general reduction procedures for certain types of theories, and modular termination results. More specifically:

– We better define theories for which a hierarchical approach is applicable, constructor-based theories, and simplify the hierarchical unification proce-
dure denoted here by $H_E(U_F)$, where $U_F$ is an additional rule-based procedure to be combined with an $E$-unification algorithm (Section 3).

– We define the requirements for the $U_F$ rule-based procedure, and develop new general rule-based procedures for subterm collapse-free and forward-closed theories (Section 3).

– Using the hierarchical approach, we develop new modularity results for the unification problem in unions of constructor-sharing theories. We define a new complexity measure to show terminating combinations of hierarchical unification algorithms. This allows us to obtain new (combined) unification algorithms for a wider variety of theories (Section 4).

– We show how the combination of hierarchical unification algorithms can be applied to unions of constructor-sharing forward-closed theories (Section 4).

The rest of the paper is organized as follows. Section 2 provides the background material. Section 2.3 contains an introduction to forward-closed theories. Section 3 introduces the notion of hierarchical unification and presents examples of theories admitting a hierarchical unification algorithm. Section 4 focuses on the combination of hierarchical unification algorithms. Finally, Section 5 contains the conclusions and future work. Omitted proofs can be found in the appendix.

2 Preliminaries

We use the standard notation of equational unification [4] and term rewriting systems [2]. Given a first-order signature $\Sigma$ and a (countable) set of variables $V$, the set of $\Sigma$-terms over variables $V$ is denoted by $T(\Sigma, V)$. The set of variables in a term $t$ is denoted by $\text{Var}(t)$. A term $t$ is ground if $\text{Var}(t) = \emptyset$. A term is linear if all its variables occur only once. For any position $p$ in a term $t$ (including the root position $\epsilon$), $t(p)$ is the symbol at position $p$, $t|_p$ is the subterm of $t$ at position $p$, and $t[u|_p]$ is the term $t$ in which $t|_p$ is replaced by $u$. A substitution is an endomorphism of $T(\Sigma, V)$ with only finitely many variables not mapped to themselves. A substitution is denoted by $\sigma = \{ x_1 \mapsto t_1, \ldots, x_m \mapsto t_m \}$, where the domain of $\sigma$ is $\text{Dom}(\sigma) = \{ x_1, \ldots, x_m \}$. Application of a substitution $\sigma$ to $t$ is written $t\sigma$.

2.1 Equational Theories

Given a set $E$ of $\Sigma$-axioms (i.e., pairs of $\Sigma$-terms, denoted by $l = r$), the equational theory $=_E$ is the congruence closure of $E$ under the law of substitutivity (by a slight abuse of terminology, $E$ is often called an equational theory). Equivalently, $=_E$ can be defined as the reflexive transitive closure $\Rightarrow_E$ of an equational step $\leftrightarrow_E$ defined as follows: $s \leftrightarrow_E t$ if there exist a position $p$ of $s$, $l = r$ (or $r = l$) in $E$, and substitution $\sigma$ such that $s|_p = l\sigma$ and $t = s[r\sigma]|_p$. An axiom $l = r$ is regular if $\text{Var}(l) = \text{Var}(r)$. An axiom $l = r$ is linear (resp., collapse-free) if $l$ and $r$ are linear (resp. non-variable terms). An equational theory is regular (resp., linear/collapse-free) if all its axioms are regular (resp., linear/collapse-free). A
theory \( E \) is subterm collapse-free if and only if for all terms \( t \) it is not the case that \( t =_E u \) where \( u \) is a strict subterm of \( t \). A theory \( E \) is syntactic if it has finite resolvent presentation \( S \), defined as a finite set of axioms \( S \) such that each equality \( t =_E u \) has an equational proof \( t \leftrightarrow_S u \) with at most one equational step \( \leftrightarrow_S \) applied at the root position. One can easily check that \( C = \{ x \ast y = y \ast x \} \) (Commutativity) and \( AC = \{ x \ast (y \ast z) = (x \ast y) \ast z, \ x \ast y = y \ast x \} \) (Associativity-Commutativity) are regular, collapse-free, and linear. Moreover, \( C \) and \( AC \) are syntactic [22]. A \( \Sigma \)-equation is a pair of \( \Sigma \)-terms denoted by \( s =^*_E t \) or simply \( s = t \) when it is clear from the context that we do not refer to an axiom. A flat \( \Sigma \)-equation is either an equation between variables or a non-variable flat \( \Sigma \)-equation of the form \( x_0 = f(x_1, \ldots, x_n) \) where \( x_0, x_1, \ldots, x_n \) are variables and \( f \) is a function symbol in \( \Sigma \). An \( E \)-unification problem is a set of \( \Sigma \)-equations, \( \Sigma = \{ s_1 =^* t_1, \ldots, s_n =^* t_n \} \), or equivalently a conjunction of \( \Sigma \)-equations. The set of variables in \( G \) is denoted by \( \text{Var}(G) \). A solution to \( G \), called an \( E \)-unifier, is a substitution \( \sigma \) such that \( s_i \sigma =_E t_i \sigma \) for all \( 1 \leq i \leq n \), written \( E \models \sigma \). A substitution \( \sigma \) is more general modulo \( E \) than \( \theta \) on a set of variables \( V \), denoted as \( \sigma \leq^V_E \theta \), if there is a substitution \( \tau \) such that \( x \tau =_E x \theta \) for all \( x \in V \). A Complete Set of \( E \)-Unifiers of \( G \), denoted by \( CSU_E(G) \), is a set of substitutions such that each \( \sigma \in CSU_E(G) \) is an \( E \)-unifier of \( G \), and for each \( E \)-unifier \( \theta \) of \( G \), there exists \( \sigma \in CSU_E(G) \) such that \( \sigma \leq^V_{\text{Var}(G)} \theta \). An \( E \)-unification algorithm is an algorithm that computes a finite \( CSU_E(G) \) for all \( E \)-unification problems \( G \). An inference rule \( G \vdash G' \) for \( E \)-unification is sound if each \( E \)-unifier \( G' \) is an \( E \)-unifier of \( G \); and complete if for each \( E \)-unifier \( \sigma \) of \( G \), there exists an \( E \)-unifier \( \sigma' \) of \( G' \) such that \( \sigma' \leq^V_{\text{Var}(G)} \sigma \). An inference system for \( E \)-unification is sound if all its inference rules are sound; and complete if for each \( E \)-unification problem \( G \) on which an inference applies and each \( E \)-unifier \( \sigma \) of \( G \), there exist an \( E \)-unification problem \( G' \) inferred from \( G \) and an \( E \)-unifier \( \sigma' \) of \( G' \) such that \( \sigma' \leq^V_{\text{Var}(G)} \sigma \). A set of equations \( G = \{ x_0 =^\ast t_0, \ldots, x_n =^\ast t_n \} \) is said to be in tree solved form if each \( x_i \) is a variable occurring once in \( G \). Given an idempotent substitution \( \sigma = \{ x_1 \mapsto t_1, \ldots, x_n \mapsto t_n \} \) (such that \( \sigma \sigma = \sigma \), \( \hat{\sigma} \) denotes the corresponding tree solved form. A set of equations is said to be in dag solved form if they can be arranged as a list \( x_1 =^\ast t_1, \ldots, x_n =^\ast t_n \) where (a) each left-hand side \( x_i \) is a distinct variable, and (b) \( \forall 1 \leq i < j \leq n: x_i \) does not occur in \( t_j \). A set of equations \( \{ x_1 =^\ast t_1, \ldots, x_n =^\ast t_n \} \) is a cycle if for any \( i \in [1, n - 1], x_i+1 \in \text{Var}(t_i), x_1 \in \text{Var}(t_n) \), and there exists \( j \in [1, n] \) such that \( t_j \) is not a variable. Given two disjoint signatures \( \Sigma_0 \) and \( \Sigma_1 \) and any \( i = 1, 0, \Sigma_i \)-terms (including the variables) and \( \Sigma_i \)-equations (including the equations between variables) are called \( \Sigma_i \)-pure. A term \( t \) is called a \( \Sigma_i \)-rooted term if its root symbol is in \( \Sigma_i \). An alien subterm of a \( \Sigma_i \)-rooted term \( t \) is a \( \Sigma_j \)-rooted subterm \( s (i \neq j) \) such that all superterms of \( s \) are \( \Sigma_j \)-rooted. We define \( E \)-unification as the unification problem in the equational theory obtained by extending \( E \) with arbitrary free function symbols.

Given a \( \Sigma_0 \)-theory \( E \), a theory \( F \cup E \) is a conservative extension of \( E \) if \( =_{F \cup E} \) and \( =_E \) coincide on \( \Sigma_0 \)-terms. When \( F \cup E \) is a conservative extension of \( E \), \( E \)-unification is said to be complete for solving the \( \Sigma_0 \)-fragment of \( F \cup E \).
unification if for any \( \Sigma_0 \)-pure \( F \cup E \)-unification problem \( G \), any \( CSU_E(G) \) is a \( CSU_{F∪E}(G) \). If \( F \) and \( E \) have disjoint signatures, \( E \)-unification is known to be complete for solving the \( \Sigma_0 \)-fragment of \( F \cup E \)-unification.

2.2 Equational Term Rewrite Systems

Given a signature \( \Sigma \), an equational term rewrite system (TRS) \( (R, E) \) is defined by a \( \Sigma \)-theory \( E \) and a finite set \( R \) of oriented \( \Sigma \)-axioms called rewrite rules and of the form \( l \rightarrow r \) such that \( l, r \) are \( \Sigma \)-terms, \( l \) is not a variable and \( \text{Var}(r) \subseteq \text{Var}(l) \). A term \( s \) rewrites to a term \( t \) w.r.t \( (R, E) \), denoted by \( s \rightarrow_{R,E} t \), if there exist a position \( p \) of \( s \), \( l \rightarrow r \in R \), and substitution \( \sigma \) such that \( s_p =_E l\sigma \) and \( t = s[r\sigma]_p \). The term \( s_p \) is called a redex. Given a TRS \( (R, E) \), \( \iff \rightarrow_{R,E} \) denotes the symmetric relation \( \iff \rightarrow_{R,E} \cup \rightarrow_{R,E} \cup \iff \). A TRS \( (R, E) \) is Church-Rosser modulo \( E \) if \( \iff \rightarrow^*_{R,E} \) is included in \( \iff \rightarrow^*_{R,E} \circ =_E \iff \rightarrow^*_{R,E} \). When \( =_E \circ \rightarrow_{R,E} \circ =_E \) is terminating, the following properties are equivalent [20]:

1. \( (R, E) \) is Church-Rosser modulo \( E \).
2. for any terms \( t, t' \), \( t \iff \rightarrow^*_{R,E} t' \) if and only if \( t \downarrow =_E t' \downarrow \), where \( t \downarrow \) (resp., \( t' \downarrow \)) denotes any normal form of \( t \) (resp., \( t' \)) w.r.t \( (R, E) \).

A TRS \( (R, E) \) is \( E \)-convergent if \( =_E \circ \rightarrow_{R,E} \circ =_E \) is terminating and \( (R, E) \) is Church-Rosser modulo \( E \). Let \( \Sigma_0 \) be the subsignature of \( \Sigma \) that consists of function symbols occurring in the axioms of \( E \). An \( E \)-convergent TRS \( (R, E) \) is said to be \( E \)-constructed if \( \Sigma_0 \cap \{ l(\epsilon) \mid l \rightarrow r \in R \} = \emptyset \).

An \( E \)-convergent TRS \( (R, E) \) is said to be subterm \( E \)-convergent if for any \( l \rightarrow r \in R \), \( r \) is either a strict subterm of \( l \) or a constant. When \( (R, E) \) is clear from the context, a normal form w.r.t \( (R, E) \) is said to be normalized. A substitution \( \sigma \) is normalized if, for every variable \( x \) in the domain of \( \sigma \), \( x\sigma \) is normalized. An instance \( l\sigma \rightarrow r\sigma \) of a rule \( l \rightarrow r \in R \) is a right-reduced instance if \( \sigma =_E \), \( \text{Var}(r) \) is normalized. A term \( t \) is an innermost redex if no subterm of \( t \) is a redex. An \( E \)-convergent TRS \( (R, E) \) is IR if every innermost redex is \( R, E \)-reducible by a right-reduced instance of a rule in \( R \). An \( E \)-convergent TRS \( (R, E) \) is IR1 if every innermost redex is \( R, E \)-reducible to a normal form in one step.

To simplify the notation, we often use tuples of terms, say \( \bar{u} = (u_1, \ldots, u_n) \), \( \bar{v} = (v_1, \ldots, v_n) \). Applying a substitution \( \sigma \) to \( \bar{u} \) is the tuple \( \bar{u}\sigma = (u_1\sigma, \ldots, u_n\sigma) \). The tuples \( \bar{u} \) and \( \bar{v} \) are said to be \( E \)-equal, denoted by \( \bar{u} =_E \bar{v} \), if \( u_1 =_E v_1, \ldots, u_n =_E v_n \). Similarly, \( \bar{u} \rightarrow^*_R \bar{v} \) if \( u_1 \rightarrow^*_R v_1, \ldots, u_n \rightarrow^*_R v_n \), \( \bar{u} \) is normalized if \( u_1, \ldots, u_n \) are normalized, and \( \bar{u} =^*_E \bar{v} \) is \( u_1 =^*_E v_1 \wedge \cdots \wedge u_n =^*_E v_n \).

2.3 Forward Closure

In this section, we introduce the notion of finite forward closure, following the definition given in [21]. Consider the rule:

\[
\text{ForwardOverlap} \quad g \rightarrow d[l'], \ l \rightarrow r \vdash (g \rightarrow d[r])\sigma
\]

where \( g \rightarrow d[l'], \ l \rightarrow r \in R \), \( l' \) is not a variable and \( \sigma \in CSU_E(l' =^*_E l) \)
For this inference rule, the notion of redundancy is defined with respect to an ordering on terms. We assume the existence of a simplification ordering $>_{\text{E}}$ such that $>_{\text{E}}$ is $E$-compatible, meaning that $s' =_{\text{E}} s >_{\text{E}} t =_{\text{E}} t'$ implies $s' >_{\text{E}} t'$, and $l >_{\text{E}} r$ for any $l \rightarrow r \in R$. *ForwardOverlap* is said to be redundant in $(R, E)$ if for each $g'$ such that $g' =_{E} g\sigma$, $g'$ is $R$-reducible by a right-reduced instance $s\mu \rightarrow t\mu$ of $R$ and either $s\mu <_{E} g\sigma$ or $(s\mu =_{E} g\sigma$ and $t\mu < d[l']\sigma$).

Let $I$ be an inference system generating rewrite rules and whose inferences are possibly redundant, like for instance $I = \{\text{ForwardOverlap}\}$. Given an equational TRS $(R, E)$, the saturation of $(R, E)$ with respect to $I$ is inductively defined as follows:

- $S_{I}^{0}(R) = R$,
- $S_{I}^{k+1}(R) = S_{I}^{k}(R) \cup \{\rho\}$ where the rule $\rho$ is obtained by applying an inference $i$ in $I$ using $(S_{I}^{k}(R), E)$ as equational TRS and such that $i$ is not redundant in $(S_{I}^{k}(R), E)$.

Let $S_{I}(R) = \bigcup_{k \geq 0} S_{I}^{k}(R)$. When $S_{I}(R)$ is finite, $S_{I}(R)$ is called a finite $I$-satisfaction of $(R, E)$. An equational TRS $(R, E)$ is $I$-saturated if $S_{I}(R) = R$.

An equational TRS has a finite forward closure if it has a finite $I$-satisfaction for $I = \{\text{ForwardOverlap}\}$. An equational TRS is forward-closed if it is $I$-saturated for $I = \{\text{ForwardOverlap}\}$.

**Example 1.** Any subterm $E$-convergent TRS has a finite forward closure. Subterm convergent TRSs are often used in the verification of security protocols [1], e.g., \{dec(enc(x, y), y) \rightarrow x\} and \{fst(pair(x, y)) \rightarrow x, \ snd(pair(x, y)) \rightarrow y\}. The equational TRSs \{dec(enc(x, k), k\ast y) \rightarrow x\} and \{rm(x \ast k, k) \rightarrow x\} are subterm $E$-convergent for $E = \text{AC}(\ast) = \{x \ast (y \ast z) = (x \ast y) \ast z, x \ast y = y \ast x\}$.

Forward closure can be connected to the notion of Finite Variant Property (FVP, for short) introduced in [11]. In [7], it was shown that for any TRS $R$, $R$ has the FVP iff it has a finite forward closure. Assuming $E$ is regular, collapse-free and $E$-unification is finitary, we can show more generally that for any $E$-constructed TRS $(R, E)$, $(R, E)$ has the FVP iff it has a finite forward closure. According to this equivalence (see Appendix [A] for a proof), any $E$-constructed TRS with the FVP corresponds to a forward-closed $E$-constructed TRS, and conversely.

In this paper, $(R, E)$ is assumed to be $E$-constructed and so the signature of $(R, E)$ necessarily includes a non-empty set of function symbols that do not occur in the axioms of $E$. Thus, this means that we actually need general $E$-unification, i.e., $E$-unification with free function symbols, instead of $E$-unification. Fortunately, when $E$ is regular and collapse-free, $E$-unification is finitary if and only if general $E$-unification is finitary. This equivalence is a consequence of a classical disjoint combination method for regular and collapse-free theories [31] that allows us to build a general $E$-unification algorithm as a combination of the syntactic unification algorithm and an $E$-unification algorithm.

From now on, the equational theory $E$ is always assumed to be regular and collapse-free when $(R, E)$ is $E$-constructed.
3 Hierarchical Unification

Consider now a union of theories \( R \cup E \) where \( E \) is regular and collapse-free and \((R, E)\) is assumed to be \( E \)-constructed. Thanks to this assumption, \( R \) and \( E \) are “sufficiently separated” and thus we can envision the problem of building an \( R \cup E \)-unification algorithm as a combination of two unification procedures: a mutation-based unification procedure processing some \( R \cup E \)-equalities, and an \( E \)-unification algorithm. The approach we will use for this problem is the hierarchical approach. Informally, the approach works as follows:

- The set of equations is processed to separate the terms over the shared signature, \( \Sigma_0 \), from terms over the non-shared one, \( \Sigma \setminus \Sigma_0 \).
- The mutation-based procedure is then used to simplify the \( \Sigma \setminus \Sigma_0 \)-equations.
- The remaining equations over the shared signature \( \Sigma_0 \) are solved using the \( E \)-unification algorithm.
- The process can repeat. If the process terminates in a solved form then the problem is solvable and a unifier is produced.

A hierarchical unification procedure is parameterized by an \( E \)-unification algorithm and a mutation-based reduction procedure \( U \). It applies some additional rules given in Figure 1: Coalesce, Split, Flatten, and VA are used to separate the terms, \( U \) is used to simplify the \( \Sigma \setminus \Sigma_0 \)-equations, and finally, Solve calls the \( E \)-unification algorithm.

**Fig. 1.** \( H_E \) rules

**Definition 1 (Hierarchical unification procedure).** Assume a \( \Sigma_0 \)-theory \( E \) for which an \( E \)-unification algorithm is known, a \( \Sigma \)-theory \( F \cup E \) for which \( E \)-unification is complete for solving the \( \Sigma_0 \)-fragment of \( F \cup E \)-unification, and an inference system \( U \) satisfying the following assumptions: \( U \) transforms only non-variable flat \( \Sigma \setminus \Sigma_0 \)-equations; \( U \) is sound and complete; and \( U \) is parameterized

\[ \text{Coalesce } \{ x = y \} \cup G \vdash \{ x = y \} \cup (G[x \mapsto y]) \]

where \( x \) and \( y \) are distinct variables occurring both in \( G \).

\[ \text{Split } \{ f(\bar{v}) = t \} \cup G \vdash \{ x = f(\bar{v}), x = t \} \cup G \]

where \( f \in \Sigma \setminus \Sigma_0 \), \( t \) is a non-variable term and \( x \) is a fresh variable.

\[ \text{Flatten } \{ v = f(\ldots,u,\ldots) \} \cup G \vdash \{ v = f(\ldots,x,\ldots), x = u \} \cup G \]

where \( f \in \Sigma \setminus \Sigma_0 \), \( v \) is a variable, \( u \) is a non-variable term, and \( x \) is a fresh variable.

\[ \text{VA } \{ s = t[u] \} \cup G \vdash \{ s = t[x], x = u \} \cup G \]

where \( t \) is \( \Sigma_0 \)-rooted, \( u \) is an alien subterm of \( t \), and \( x \) is a fresh variable.

\[ \text{Solve } G \cup G_0 \vdash \bigvee_{\sigma_0 \in \text{CSU}(G_0)} G \cup \hat{\sigma_0} \]

where \( G \) is a set of \( \Sigma \setminus \Sigma_0 \)-equations, \( G_0 \) is a set of \( \Sigma_0 \)-equations, \( G_0 \) is \( E \)-unifiable and not in tree solved form, \( \hat{\sigma_0} \) is the tree solved form associated with \( \sigma_0 \), and w.l.o.g for any \( x \in \text{Dom}(\sigma_0) \), \( x\sigma_0 \in \text{Var}(G_0) \) if \( x\sigma_0 \) is a variable.
by some finite set \( S \) of \( F \cup E \)-equalities such that the soundness of each inference \( \vdash_U \) follows from at most one equality in \( S \). Under these assumptions, the \( H_E(U) \) inference system is defined as the repeated application of some inference from \( H_E \) (cf. Figure 1) or \( U \), using the following order of priority: Coalesce, Split, Flatten, VA, U, Solve. An \( F \cup E \)-unification problem is in separate form if it is a normal form with respect to \( H_E \backslash \{\text{Solve}\} \).

Note, that when we speak of an inference system, \( U \), this is not just a set of rules but also a strategy for applying those rules. This could include, as in the \( \mathcal{E}_{AC} \) case of Proposition 3, methods for detecting errors such as occur-checks and non-termination [15].

**Proposition 1.** Let \((R, E)\) be any \( E \)-constructed TRS such that an inference system \( U \) following Definition 7 is known for the equational theory \( R \cup E \), in addition to an existing \( E \)-unification algorithm. Then \( E, R \cup E \) and \( U \) satisfy the assumptions of Definition 7 and the \( H_E(U) \) inference system provides a sound and complete \( R \cup E \)-unification procedure if the normal forms w.r.t \( H_E(U) \) are either the dag solved forms or problems that are not \( R \cup E \)-unifiable. If \( H_E(U) \) is terminating, then it is an \( R \cup E \)-unification algorithm.

**Proof.** If \((R, E)\) is \( E \)-constructed, then \( E \)-unification is complete for solving the \( \Sigma_0 \)-fragment of \( R \cup E \)-unification, and so all the assumptions of Definition 1 are satisfied. By construction, \( H_E(U) \) is sound and complete. Since the \( R \cup E \)-unifiable normal forms w.r.t \( H_E(U) \) are assumed to be the dag solved forms, collecting all the dag solved forms reached by \( H_E(U) \) suffices to get a complete set of \( R \cup E \)-unifiers.

3.1 Subterm Collapse-Free Theories

Hierarchical unification algorithms are known for particular subterm collapse-free theories of particular interest for protocol analysis.

**Proposition 2.** ([30][13]) Let \( E \) be the empty \( \Sigma_0 \)-theory where \( \Sigma_0 \) only consists of a binary function symbol \(*\), \( R_D = \{b(x * y) \to b(x) * b(y)\} \) and \( R_{D1} = \{f(x * y, z) \to f(x, z) * f(y, z)\} \). The equational TRSs \((R_D, E)\) and \((R_{D1}, E)\) are \( E \)-constructed. Moreover, \( R_D \cup E \) (resp., \( R_{D1} \cup E \) ) is a subterm collapse-free theory admitting a unification algorithm of the form \( H_E(U_D) \) (resp., \( H_E(U_{D1}) \) ).

**Proof.** Subterm collapse-freeness follows from the fact that both theories are non-size-reducing. The inference system \( U_{D1} \) can be derived following the approach developed in [13] and based on the one initiated in [30] for one-side distributivity. The same approach can be applied for \( R_D \) to get \( U_D \).  

**Proposition 3.** ([13]) Let \( AC = AC(\oplus) \), \( R_E = \{exp(exp(x, y), z) \to exp(x, y \oplus z), exp(x \oplus y, z) \to exp(x, z) \oplus exp(y, z)\} \) and \( R_F = \{enc(enc(x, y), z) \to enc(x, y \oplus z)\} \). The equational TRSs \((R_E, AC)\) and \((R_F, AC)\) are \( AC \)-constructed. Moreover, \( \mathcal{E}_{AC} = R_E \cup AC \) (resp., \( \mathcal{F}_{AC} = R_F \cup AC \) ) is a subterm collapse-free theory admitting a unification algorithm of the form \( H_{AC}(U_E) \) (resp., \( H_{AC}(U_F) \) ).
Proof. In [15] it is shown that both $\mathcal{E}_{AC}$ and $\mathcal{F}_{AC}$ are subterm collapse-free theories. Also in [15] a mutation-based inference system is developed for $R_{E}$ (and $R_{F}$). The procedure reduces the $\Sigma \setminus \Sigma_{0}$ equations into solved forms after which a Solve type rule is used to apply AC-unification. It is shown in [15] that the solve step need only by applied once. Hence, the procedure in [15] is an example of a $H_{AC}(U_{E})$ unification procedure. \qed

3.2 Forward-Closed E-Constructed TRSs

For any forward-closed E-constructed TRS $(R, E)$ such that $E$ is regular and collapse-free, an $R \cup E$-unification algorithm of the form $H_{E}(U)$ can be obtained by defining some inference system $U$, based on the Basic Syntactic Mutation approach initiated for the class of theories closed by paramodulation [23], and already applied in [13] to a particular class of forward-closed equational TRSs.

Let $BSM_{R}$ be the inference system given in Figure 2. One can notice that each inference rule in $BSM_{R}$ generates some boxed terms. This particular annotation of terms, detailed in [23,13], allows us to control the rules application, disregarding needless inferences on boxed terms, in such a way that $BSM_{R}$ is terminating.

\begin{align*}
\text{Imit} & : \bigcup \{ x = f(\bar{v}) \} \cup G \vdash \{ x = f(\bar{y}) \} \cup \bigcup \{ \bar{y} = \bar{v}_{i} \} \cup G \\
\text{where } & \ f \in \Sigma \setminus \Sigma_{0}, \ i > 1, \ \bar{y} \text{ are fresh variables and there are no more equations } x = f(\ldots) \text{ in } G.
\end{align*}

\begin{align*}
\text{MutConflict}_{R} & : \{ x = f(\bar{v}) \} \cup G \vdash \{ x = f(\bar{x}) = \bar{v} \} \cup G \\
\text{where } & \ f \in \Sigma \setminus \Sigma_{0}, \ f(\bar{s}) \rightarrow t \text{ is a fresh instance of a rule in } R, \ f(\bar{v}) \text{ is unboxed, and there} \\
\text{is another equation } & x = u \text{ in } G \text{ with a non-variable term } u \text{ or } x = f(\bar{v}) \text{ occurs in a cycle.}
\end{align*}

\begin{align*}
\text{ImitCycle} & : \{ x = f(\bar{v}) \} \cup G \vdash \{ x = f(\bar{y}) \} \cup G \\
\text{where } & \ f \in \Sigma \setminus \Sigma_{0}, \ f(\bar{v}) \text{ is unboxed, } \bar{y} \text{ are fresh variables and } x = f(\bar{v}) \text{ occurs in a cycle.}
\end{align*}

Fig. 2. $BSM_{R}$ rules

An $R \cup E$-unification algorithm combining $BSM_{R}$ and an $E$-unification algorithm has been developed in [13] for the case of any forward-closed convergent TRS $R$ such that the left-hand sides of $R$ are linear and contain no symbols of $E$. In this paper, we extend [13] to any forward-closed E-constructed TRS $(R, E)$, without any further restriction on $R$.

The soundness and completeness of $BSM_{R}$ is shown by the following lemma.

Lemma 1. Let $(R, E)$ be any forward-closed E-constructed TRS over the signature $\Sigma$. For each equality $u =_{R \cup E} v$ such that $u$ is $\Sigma \setminus \Sigma_{0}$-rooted and $v$ is normalized, one of the following is true:

1. $u = f(\bar{u})$, $v = f(\bar{v})$ and $\bar{u} =_{R \cup E} \bar{v}$. 

9
2. \( u = f(\bar{u}) \), there exist \( f(\bar{s}) \rightarrow t \in R \) and a normalized substitution \( \sigma \) such that \( \bar{u} = R_E \bar{s} \sigma, v = E \tau \sigma \) and \( s_\sigma \tau \sigma \) are normalized.

**Proof.** Let us analyze the possible rewrite proofs \( \rightarrow_{R,E}^* \) of \( u = R_E v \).

First, if there is no step at the root position, then we get \( u = f(\bar{u}) \rightarrow_{R,E}^* R f(\bar{u}) = E v \) where \( \bar{u} \rightarrow_{R,E} \bar{u}' \) and \( \bar{u}' \) are normalized. Since \( f \) is a free symbol for \( E \), we have that \( v = f(\bar{v}) \) and \( \bar{v}' = E \bar{v} \). Hence, \( \bar{u} = R_E \bar{v} \) since \( \bar{u} = R_E \bar{u}' \).

Second, if there is one step at the root position, then we have

\[
u = f(\bar{u}) \rightarrow_{R,E}^* f(\bar{u}') = f(\bar{s}) \sigma \rightarrow_{R,E,\epsilon} t \sigma = E v\]

where \( f(\bar{s}) \rightarrow t \in R, \bar{u} \rightarrow_{R,E} \bar{u}', \bar{u}' \) are normalized, \( \bar{u}' = E s_\sigma \), and so \( \sigma, s_\sigma \) are normalized. Since \( t \sigma = E v \) and \( v \) is normalized, \( t \sigma \) is also normalized. \( \square \)

A unification procedure of the form \( H_E(BSM_R) \) corresponds to the BSC unification procedure given in [13] except that \textbf{Solve} is applied in BSC before \( BSM_R \) rules. However, the termination proof stated for BSC in [13] also holds when \textbf{Solve} is applied after the \( BSM_R \) rules.

**Lemma 2.** Assume \( E \) is any regular and collapse-free theory such that an \( E \)-unification algorithm is known. Let \( (R,E) \) be a forward-closed \( E \)-constructed TRS and \( BSM_R \) the inference system given in Fig. 2. Then \( H_E(BSM_R) \) is an \( R \cup E \)-unification algorithm.

**Example 2.** Consider \( R = \{ h(x) \rightarrow a \times x \}, R' = \{ f(x,y) \rightarrow a'(y) \times x \} \) and \( E = \{ x \times (y * z) = (x \times y) * (x \times z) \} \). The theory \( E \) corresponds to left-distributivity and an \( E \)-unification algorithm is given in [22]. Since \( (R,E) \) and \( (R',E) \) are forward-closed and \( E \)-constructed, \( H_E(BSM_R) \) and \( H_E(BSM_R') \) are unification algorithms for \( R \cup E \) and \( R' \cup E \), respectively. Notice that \( h(x * y) = R_E h(x) \times h(y) \) and \( f(x * y, z) = R_{E \cup E} f(x, z) * f(y, z) \).

**Example 3.** Consider \( R = \{ \pi_1(x,y) \rightarrow x, \pi_2(x,y) \rightarrow y, dec(\text{enc}(x,y), y) \rightarrow x \} \) and \( E = \{ \text{enc}(x,y,z) = \text{enc}(x,z), \text{enc}(y,z) \} \). An \( E \)-unification algorithm can be obtained following the approach developed in [20,16] and can be used in a hierarchical unification procedure of the form \( H_E(BSM_R) \). Since \( (R,E) \) is forward-closed and \( E \)-constructed, \( H_E(BSM_R) \) is an \( R \cup E \)-unification algorithm.

### 4 Combined Hierarchical Unification

We are now interested in combining hierarchical unification algorithms known for \( E \)-constructed TRSs. Given two \( E \)-constructed TRSs, say \( (R_1,E) \) and \( (R_2,E) \), the problem is to study the possible construction of a (combined) hierarchical unification algorithm for \( (R_1 \cup R_2,E) \) using the two hierarchical unification algorithms known for \( (R_1,E) \) and \( (R_2,E) \). We investigate this combination problem for the two classes of \( E \)-constructed TRSs introduced in Section 3. First, we consider a class of \( E \)-constructed TRSs \( (R,E) \) such that \( R \cup E \) is subterm collapse-free. Second, we study the class of forward-closed \( E \)-constructed TRSs \( (R,E) \) such that \( E \) is regular and collapse-free.
4.1 Combining Subterm Collapse-Free Theories

Let us first consider a technical lemma which is useful to get a hierarchical unification procedure.

**Lemma 3.** Let \((R_1, E)\) and \((R_2, E)\) be two \(E\)-constructed TRSs sharing only symbols in \(E\) such that, for \(i = 1, 2\), \(R_i \cup E\) admits a sound and complete unification procedure of the form \(H_E(U_i)\). Assume that \(R_1 \cup R_2 \cup E\) is subterm collapse-free, and for any \(\Sigma_1 \setminus \Sigma_0\)-rooted term \(t_1\) and any \(\Sigma_2 \setminus \Sigma_0\)-rooted term \(t_2\), \(t_1\) cannot be equal to \(t_2\) modulo \(R_1 \cup R_2 \cup E\). Then, \(H_E(U_1 \cup U_2)\) is a sound and complete \(R_1 \cup R_2 \cup E\)-unification procedure.

**Proof.** According to the assumptions, any normal form w.r.t \(H_E(U_1 \cup U_2)\) is \(R_1 \cup R_2 \cup E\)-unifiable iff it is in dag solved form. So, Proposition \([1]\) applies. \(\square\)

We study below a possible way to satisfy the assumptions of Lemma \([3]\).

**Definition 2 (Layer-preservingness).** Let \((R, E)\) be an \(E\)-constructed TRS over the signature \(\Sigma\). A \(\Sigma\)-term \(t\) is said to be \(\Sigma_0\)-capped if there exist a constant-free \(\Sigma_0\)-term \(u\) and a substitution \(\sigma\) such that \(t = u\sigma\), \(\text{Dom}(\sigma) = \text{Var}(u)\) and \(\text{Ran}(\sigma)\) is a set of \(\Sigma \setminus \Sigma_0\)-rooted terms. The TRS \((R, E)\) is said to be layer-preserving if \(R \cup E\) is subterm collapse-free and any normal form of any \(\Sigma \setminus \Sigma_0\)-rooted term is \(\Sigma_0\)-capped.

**Remark 1.** An easy way to get layer-preservingness of \((R, E)\) is to assume that \(R \cup E\) is subterm collapse-free and the right hand-sides of rules in \(R\) are rooted by \(\Sigma \setminus \Sigma_0\)-symbols. In that case the term \(u\) in Definition \([2]\) is simply a variable. The assumption that rules in \(R\) are \(\Sigma \setminus \Sigma_0\)-rooted was used in \([14]\), and layer-preservingness generalizes this assumption.

The property of being \(E\)-constructed and layer-preserving is modular.

**Lemma 4.** Assume \(E\) is subterm collapse-free, for \(i = 1, 2\), \((R_i, E)\) is an \(E\)-constructed layer-preserving TRS whose signature is \(\Sigma_i\), and \(\Sigma_1 \cap \Sigma_2 = \Sigma_0\). If \(\Rightarrow_{E} \circ \Rightarrow_{R_1 \cup R_2} \circ \Rightarrow_{E}\) is terminating, then \((R_1 \cup R_2, E)\) is an \(E\)-constructed layer-preserving TRS, and for any \(\Sigma_1 \setminus \Sigma_0\)-rooted term \(t_1\) and any \(\Sigma_2 \setminus \Sigma_0\)-rooted term \(t_2\), \(t_1\) cannot be equal to \(t_2\) modulo \(R_1 \cup R_2 \cup E\).

**Proof.** To show that \((R_1 \cup R_2, E)\) is layer-preserving, we have to prove that \(R_1 \cup R_2 \cup E\) remains subterm collapse-free. The modularity of subterm collapse-freeness has been shown in \([14]\) when the right-hand sides of \(R_i\) are \(\Sigma_i \setminus \Sigma_0\)-rooted, for \(i = 1, 2\). Actually, a similar proof by contradiction can be performed in the case \(R_i\) is layer-preserving, for \(i = 1, 2\). Let us consider the *height of layers* of a term \(t\), inductively defined as follows:

- \(ht(t) = 0\) if \(t\) is a variable,
- \(ht(t) = 1\) if \(t\) is a non-variable pure term,
- \(ht(t) = 1 + \max\{ht(u) \mid u\text{ is an alien subterm of }t\}\) if \(t\) is not pure.
Assume there exists a term $t$ and a non-empty position $p$ such that $t =_{R_1 \cup E_2 \cup E} t_{lp}$. If the path from $\epsilon$ to $p$ contains only symbols from one theory, say $R_i \cup E_i$, this would lead to a contradiction with the subterm collapse-freeness of $R_i \cup E_i$. Now let $t'$ and $u'$ be the respective normal forms of $t$ and $t_{lp}$ w.r.t $(R_1 \cup R_2, E)$. Since $t' =_E u'$ and $E$ is subterm collapse-free, we have that $t'$ and $u'$ have the same height of layers. By the layer-preserving assumption, $t$ and $t'$ have the same height of layers, which leads to a contradiction since the path from $\epsilon$ to $p$ includes some $\Sigma_1$-symbol(s) and some $\Sigma_2$-symbol(s).

Assume there exist some $\Sigma_1 \setminus \Sigma_0$-rooted term $t_1$ and some $\Sigma_2 \setminus \Sigma_0$-rooted term $t_2$ such that $t_1 =_{R_1 \cup R_2 \cup E} t_2$, then the $R_1 \cup R_2$-normal forms of $t_1$ and $t_2$ must be $E$-equal $\Sigma_0$-terms. The layer-preserving assumption implies that the $R_1 \cup R_2$-normal form of $t_1$ must still contain a symbol in $\Sigma_1 \setminus \Sigma_0$, and this contradicts the fact that it is a $\Sigma_0$-term. □

Remark 2. To satisfy the condition $=_E \circ \to_{R_1 \cup R_2} \circ =_E$ is terminating, it suffices to exhibit an $E$-compatible reduction ordering $>$ such that $l > r$ for any $l \to r \in R_1 \cup R_2$. In that case, $>$ is defined on terms built over $\Sigma_1 \cup \Sigma_2$.

By Lemma 3 and Proposition 1 can be satisfied, and this leads to a hierarchical unification procedure for the combined TRS. In the following, we consider a notion of decreasingness in order to study the termination of this unification procedure.

Definition 3 (Decreasingness). Consider a complexity measure defined as a mapping $C$ from separate forms to natural numbers. An $H_E(U)$ inference system is said to be $C$-decreasing if for any separate form $G \cup G_0$ we have that (1) for any $G'$ such that $G \cup G_0 \vdash_U G' \cup G_0$, the separate form of $G' \cup G_0$ does not increase $C$; (2) for any $G_0$ such that $G \cup G_0 \vdash_{\text{solve}} G \cup G_0'$, then either the separate form of $G \cup G_0$ is in normal form w.r.t $H_E(U)$, or it decreases $C$.

Consequently, $H_E(U)$ is terminating if there exists some $C$ such that $H_E(U)$ is $C$-decreasing.

Theorem 1. Assume $E$ is a subterm collapse-free theory such that an $E$-unification algorithm is known, and $C$ is a complexity measure defined on separate forms. Let $(R_1, E)$ and $(R_2, E)$ be two $E$-constructed TRSs sharing only symbols in $E$ such that, for $i = 1, 2$, $(R_i, E)$ is layer-preserving, and $R_i \cup E$ admits a $C$-decreasing unification algorithm of the form $H_E(U_i)$. If $=_E \circ \to_{R_1 \cup R_2} \circ =_E$ is terminating, then $(R_1 \cup R_2, E)$ is an $E$-constructed TRS such that $(R_1 \cup R_2, E)$ is layer-preserving, and $R_1 \cup R_2 \cup E$ admits a $C$-decreasing unification algorithm of the form $H_E(U_1 \cup U_2)$.

Proof. $(R_1 \cup R_2, E)$ is layer-preserving by Lemma 1. In addition, a $\Sigma_1 \setminus \Sigma_0$-rooted term cannot be equal to a $\Sigma_2 \setminus \Sigma_0$-rooted term modulo $R_1 \cup R_2 \cup E$. Then, Lemma 3 and Proposition 1 can be applied. Hence, $H_E(U_1 \cup U_2)$ provides a sound and complete $R_1 \cup R_2 \cup E$-unification procedure. Moreover, $H_E(U_1 \cup U_2)$ is $C$-decreasing and so it is terminating. □
Example 4. Consider the theories $E_{AC}$ and $F_{AC}$ introduced in Proposition \[3\] and the corresponding hierarchical unification algorithms $H_{AC}(U_E)$ and $H_{AC}(U_F)$ where the mutation rules defining $U_E$ and $U_F$ can be found in \[13\]. Let $SVC$ be the complexity measure defined as follows: given an $R \cup E$-unification problem in separate form $G \cup G_0$, $SVC(G \cup G_0)$ is the number of equivalence classes of variables shared by $G$ and $G_0$ that are variables abstracting $\Sigma \setminus \Sigma_0$-rooted terms.

Let us now check that the unification algorithms $H_{AC}(U_E)$ and $H_{AC}(U_F)$ are both $SVC$-decreasing. On the one hand, it is routine to verify that any (mutation) rule in $U_E$ (resp., $U_F$) does not lead, via a further possible application of $VA$, to new shared variables which are abstracting $\Sigma \setminus \Sigma_0$-rooted terms. Hence, the rules in $U_E$ (resp., $U_F$) cannot increase $SVC$. On the other hand, Solve leads to either a normal form w.r.t $H_{AC}(U_E)$ (resp., $H_{AC}(U_F)$), or it generates some equality $x =? y$ between variables $x$ and $y$ for which there are $\Sigma \setminus \Sigma_0$-equations $x =? s$ and $y =? t$ in $G$. In the last case, the respective equivalence classes of $x$ and $y$ are merged into a single one by applying Solve and so, Solve strictly decreases $SVC$. By Theorem \[4\] we get that $E_{AC} \cup F_{AC}$ admits a $SVC$-decreasing unification algorithm of the form $H_{AC}(U_E \cup U_F)$. Notice this means that we can use the termination strategy used in the individual $H_{AC}(U_E)$ and $H_{AC}(U_F)$ algorithms to obtain a termination strategy for the hierarchical combined algorithm, $H_{AC}(U_E \cup U_F)$. We suspect that this complexity measure, $SVC$, could be useful for proving termination in other theories.

To conclude this section, let us mention the problem of combining two copies of the same $E$-constructed TRS, provided that only the symbols in $E$ are possibly shared. In that very particular case, subterm collapse-freeness is sufficient and there is no need to find a decreasing complexity measure.

**Theorem 2.** Consider $(R, E)$ is an $E$-constructed TRS over the signature $\Sigma$. Assume $R \cup E$ is subterm collapse-free and $R \cup E$ admits a unification algorithm of the form $H_E(U)$. Let $(R', E)$ be a copy of $(R, E)$ obtained by renaming the $\Sigma \setminus \Sigma_0$-symbols. Then, $(R \cup R', E)$ is an $E$-constructed TRS such that $R \cup R' \cup E$ is subterm collapse-free, and $R \cup R' \cup E$ admits a unification algorithm of the form $H_E(U \cup U')$, where $U'$ is obtained from $U$ by applying the same renaming as the one defining $(R', E)$.

**Proof.** If $R \cup R' \cup E$ is not subterm collapse-free, then this would imply that $R \cup E$ is not subterm collapse-free, leading to a contradiction with the assumption. By construction, $H_E(U \cup U')$ is sound and complete. For each inference $P \vdash_{H_E(U \cup U')} Q$, there exists an inference $\iota(P) \vdash_{H_E(U)} \iota(Q)$ where $\iota$ denotes the morphism replacing each symbol $f' \in \Sigma' \setminus \Sigma_0$ by the corresponding function symbol $f \in \Sigma \setminus \Sigma_0$. Thus, the termination w.r.t $H_E(U)$ implies the termination w.r.t $H_E(U \cup U')$. \qed

Example 5. Consider the $E$-constructed TRSs $(R_D, E)$ and $(R_{D1}, E)$ defined in Proposition \[2\] and their copies $R'_D = \{h'(x * y) \rightarrow h'(x) * h'(y)\}$ and $R'_{D1} = \{f'(x * y, z) \rightarrow f'(x, z) * f'(y, z)\}$. The theories $R_D \cup E$ and $R_{D1} \cup E$ are subterm collapse-free and admit unification algorithms of the form $H_E(U_D)$ and
For Example 3 where $E$ is a modular way in unions of forward-closed $E$ and admit unification algorithms of the form $H_E(U_D \cup U_D')$ and $H_E(U_D \cup U_D')$, respectively.

4.2 Combining Forward-Closed $E$-Constructed TRSs

The union of two forward-closed $E$-constructed TRSs remains a forward-closed $E$ constructed TRS. Thus, a hierarchical unification algorithm can be constructed in a modular way in unions of forward-closed $E$-constructed TRSs.

**Theorem 3.** Assume $E$ is a regular and collapse-free theory such that an $E$-unification algorithm is known. Let $(R_1, E)$ and $(R_2, E)$ be two forward-closed $E$-constructed TRSs sharing only symbols in $E$. Then $R_1 \cup R_2 \cup E$ admits a unification algorithm of the form $H_E(BSM_{R_1 \cup R_2})$.

**Proof.** $(R_1 \cup R_2, E)$ is a forward-closed $E$-constructed TRS, and so by Lemma 2 $R_1 \cup R_2 \cup E$ admits a unification algorithm of the form $H_E(BSM_{R_1 \cup R_2})$, which coincides with $H_E(BSM_{R_1 \cup R_2})$. \hfill $\square$

In the following, we investigate the case where $E$ already admits a hierarchical unification algorithm of the form $H_{E'}(U')$ for a subtheory $E'$ of $E$, like in Example 3 where $E$ has a hierarchical unification algorithm of the form $H_{E'}(U')$ for $E' = \emptyset$. In that case, we can consider the following compositionality lemma:

**Lemma 5.** Let $(R, E)$ be an $E$-constructed TRS such that $R \cup E$ admits a unification algorithm of the form $H_E(U)$, and $E$ admits a unification algorithm of the form $H_{E'}(U')$, where $E'$ denotes a subtheory of $E$. Then $R \cup E$ also admits a unification algorithm of the form $H_{E'}(U \cup U')$.

**Example 6.** (Example 3 continued) $R \cup E$ admits a unification algorithm of the form $H_{\emptyset}(BSM_R \cup U')$ where $H_{\emptyset}(U')$ is a hierarchical $E$-unification algorithm.

**Example 7.** Let us consider a theory used in practice to model a group messaging protocol. For this protocol, the theory modeling the intruder can be defined as a combination $R_{ENC} \cup K$ where $K = \{keyexch(x, pk(x'), y, pk(y')) = keyexch(x', pk(x), y', pk(y))\}$ and $(R_{ENC}, K)$ is the forward-closed $K$-constructed TRS where

$$R_{ENC} = \begin{cases} 
    adec(aenc(m, pk(sk)), sk) \rightarrow m \\
    getsig(sign(m, sk)) \rightarrow m \\
    checksign(sign(m, sk), m, pk(sk)) \rightarrow ok \\
    sdec(senc(m, k), k) \rightarrow m
\end{cases}$$

$K$ is a theory closed by paramodulation and so $K$-unification is finitary. By Lemma 2, $R_{ENC} \cup K$ has a hierarchical unification algorithm of the form $H_K(BSM_{R_{ENC}})$. The mutation-based unification algorithm known for theories closed by paramodulation can be reworded as a hierarchical unification algorithm, of the form $H_{\emptyset}(U_K)$ for $K$. By Lemma 5, $H_{\emptyset}(BSM_{R_{ENC}} \cup U_K)$ is another $R_{ENC} \cup K$-unification algorithm.
Applying Lemma 5, we can easily show that a forward-closed $E$-constructed TRS can be combined with a regular and collapse-free $E$-constructed TRS whose equational theory admits a hierarchical unification algorithm.

**Lemma 6.** Assume $E$ is a regular and collapse-free theory such that an $E$-unification algorithm is known. Let $(R_1, E)$ and $(R_2, E)$ be two $E$-constructed TRSs sharing only symbols in $E$ such that $(R_1, E)$ is forward-closed, and $R_2 \cup E$ is a regular and collapse-free theory $E_2$ admitting a unification algorithm of the form $H_E(U_2)$. Then $(R_1, E_2)$ is a forward-closed $E_2$-constructed TRS and $R_1 \cup E_2$ admits a unification algorithm of the form $H_E(BSM_{R_1} \cup U_2)$.

**Proof.** $(R_1, E_2)$ is forward-closed because $(R_1, E)$ is forward-closed and the equational theory $=_E$ coincides with $=_{E_2}$ on $\Sigma_1$-terms. By Lemma 2, $R_1 \cup E_2$ admits a unification algorithm of the form $H_{E_2}(BSM_{R_1})$. According to Lemma 5, $R_1 \cup E_2$ also admits a unification algorithm of the form $H_{E}(BSM_{R_1} \cup U_2)$.

**Example 8.** Let $(R, AC(\odot))$ be a forward-closed $AC(\odot)$-constructed TRS such that $\odot$ is the only function symbol shared by $R \cup AC(\odot)$ and $E_{AC}$ (resp., $F_{AC}$). By Lemma 3, $R \cup E_{AC}$ (resp., $R \cup F_{AC}$) admits a unification algorithm of the form $H_{AC}(BSM_{R} \cup U_{E})$ (resp., $H_{AC}(BSM_{R} \cup U_{F})$). According to Example 4, $E_{AC} \cup F_{AC}$ admits a unification algorithm of the form $H_{AC}(U_{E} \cup U_{F})$. Then, by Lemma 6, $R \cup E_{AC} \cup F_{AC}$ admits a unification algorithm of the form $H_{AC}(BSM_{R} \cup U_{E} \cup U_{F})$.

## 5 Conclusion

We have introduced a hierarchical unification framework as a generic tool to construct unification procedures for (combined) equational theories defined by $E$-constructed TRSs. We have presented new combination results for the simplest case of subterm collapse-free theories, and a natural follow-up would be to study the case of regular and collapse-free theories. A challenging future work is to investigate the general case of arbitrary theories.

Hierarchical unification allows us to handle syntactic theories $R \cup E$ while the $E$-unification algorithm can be arbitrary. According to this observation, we plan to study a weakening of syntacticness, in order to allow theories $R \cup E$ that are just syntactic modulo $E$.

We have also begun the implementation of the above hierarchical combination procedure. To begin with, we are using $E = AC$ as the background theory. However, we will explore expanding this to additional equational theories. In the long term, we want to promote the use of non-disjoint combination procedures [10] and mutation-based procedures [17] in verification engines dedicated to protocol analysis, targeting unification problems as well as some decision problems related to the knowledge of an intruder, such as intruder deduction (a reachability problem) and indistinguishability (an equivalence problem) [18]. The goal is to improve automation of verification methods when theories share for instance $AC$ symbols.
References


A Forward Closure and the Finite Variant Property

Unification in the class of theories with the Finite Variant Property is known to be finitary, thanks to a unification algorithm based on the computation of finite variants [11,19].

Definition 4 ((R,E)-variant). Let (R,E) be an E-convergent TRS. An (R,E)-variant of a term t is a pair (u,θ) such that u =E tθ ↓, the domain of θ is included in Var(t) and θ is normalized. Given two variants (u,θ) and (v,γ) of a term t, (u,θ) is more general than (v,γ), denoted by (u,θ) ≤E (v,γ) if there exists a substitution τ such that uτ =E v and θτ =E γ. A set V_{(R,E)}(t) = \bigcup_{i \in I} \{(u_i,θ_i)\} is a complete set of (R,E)-variants of t if for any (R,E)-variant (v,γ) of t, there is some i ∈ I such that (u_i,θ_i) ≤E (v,γ). (R,E) is said to have the Finite Variant Property (FVP, for short) if any term admits a finite complete set of (R,E)-variants.

The notion of (R,E)-variant can be lifted to any R ∪ E-unification problem G by considering it as a term, where =? and ∧ are viewed as additional function symbols [26], and V_{(R,E)}(G) denotes a finite complete set of (R,E)-variants of G. When (R,E) has the FVP, any R ∪ E-unification problem G reduces to E-unification problems in V_{(R,E)}(G). In many cases, computing the complete set of variants V_{(R,E)}(G) can be prohibitive even with an efficient implementation of folding variant narrowing [19,12]. In the case of an E-convergent TRS (R,E) having the FVP and whose signature is Σ, the finite computation of (R,E)-variants leads to an R ∪ E-unification algorithm, provided that a unification algorithm is known for the Σ-theory E.

As shown in [7,21], forward closure can be connected to the notion of Finite Variant Property. In the case of E-constructed TRSs, this connection can be formally stated as follows.

Lemma 7. Assume E is any regular and collapse-free equational theory such that E-unification is finitary. If (R,E) is any E-constructed TRS, then

- (R,E) is forward-closed iff (R,E) is IR1,
- (R,E) has a finite forward closure iff (R,E) has the FVP.

Proof. We rely on some results that have been shown in [21] for an inference system \mathcal{I} including ForwardOverlap plus the following rule:

Parallel \hspace{1cm} s \rightarrow t, \hspace{0.5cm} l \rightarrow r, \hspace{0.5cm} v = u[l'] \vdash vσ \rightarrow v'

where
\(- s \rightarrow t, l \rightarrow r \in R,\)
\(- v = u[l'] \in E,\)
\(- l' is a non-variable strict subterm of \(u),\)
\(- \sigma \in CSU_E(l = l' \land s = s') u,\)
\(- v' is a normal form of \(v\) w.r.t \((R, E)\),\)
\(- there exist a variable \(x\) of \(t\) and a position \(p\) of \(x\) such that \(l' = E(x)\) \(|_p\).\)

Parallel is said to be redundant in \((R, E)\) if either \(s\) is \(R, E\)-reducible by a right-reduced instance of a rule in \(R\), or for each \(s'\) such that \(s' = E s\), a strict subterm of \(s'\) is \(R, E\)-reducible.

Let \(I = \{\text{Parallel, ForwardOverlap}\}\) and \((R, E)\) an arbitrary \(E\)-convergent TRS. The following statements are proved in [21]:

\(- \) \((R, E)\) is \(IR1\) iff \((R, E)\) is \(I\)-saturated.
\(- \) If \((R, E)\) is \(IR1\) and \(E\)-unification is finitary, then \((R, E)\) has the FVP.
\(- \) If \((R, E)\) has the FVP, then \((R, E)\) has a finite \(I\)-saturation.

Assume \((R, E)\) is \(E\)-constructed and \(E\) is both regular and collapse-free. In this particular case, Parallel does not apply since a \(\Sigma_0\)-rooted term \(l'\) is not \(E\)-unifiable with a \(\Sigma \setminus \Sigma_0\)-rooted term \(l\). Consequently, \(I\)-saturation reduces to forward closure, and the above statements can be reworded as follows:

\(- \) \((R, E)\) is \(IR1\) iff \((R, E)\) is forward-closed.
\(- \) If \((R, E)\) is \(IR1\) and \(E\)-unification is finitary, then \((R, E)\) has the FVP.
\(- \) If \((R, E)\) has the FVP, then \((R, E)\) has a finite forward closure.

To conclude the proof, notice that if \(R'\) is a finite closure of \((R, E)\), then \((R', E)\) is forward-closed and both \((R', E)\) and \((R, E)\) have the FVP. \(\Box\)

B Compositionality Lemma

Lemma 5

Proof. Consider \(\Sigma' = \Sigma_0\) and the \(\Sigma'\)-theory \(E = F' \cup E'\). Assume \(R \cup E\) (resp., \(F' \cup E'\)) has a unification algorithm of the form \(H_E(U)\) (resp., \(H_{E'}(U')\)), where \(U\) (resp., \(U'\)) is sound, complete, and parameterized by some finite set \(S\) (resp., \(S'\)) of \(R \cup E\)-equalities (resp., \(F' \cup E'\)-equalities) such that the soundness of each inference \(\vdash_U\) (resp., \(\vdash_{U'}\)) follows from at most one equality in \(S\) (resp., \(S'\)).

Since \(E\)-unification is complete for solving the \(\Sigma'\)-fragment of \(R \cup E\)-unification, \(U'\) is also sound and complete for \(R \cup F' \cup E'\). Hence, the inference system \(U \cup U'\) is sound and complete. Moreover, \(S \cup S'\) is a finite set of \(R \cup F' \cup E'\)-equalities such that the soundness of each inference \(\vdash_{U \cup U'}\) follows from at most one equality in \(S \cup S'\).

Since \(E\)-unification is complete for solving the \(\Sigma'\)-fragment of \(R \cup E\)-unification and \(E'\)-unification is complete for solving the \(\Sigma_0'\)-fragment of \(E\)-unification, we have that \(E'\) is also complete for solving the \(\Sigma_0'\)-fragment of \(R \cup F' \cup E'\)-unification.
Consequently, $E'$, $R \cup F' \cup E'$ and $U \cup U'$ satisfy all the assumptions of Definition 1, and so $H_{E'}(U \cup U')$ is well-defined. Since $H_{E'}(U \cup U')$ corresponds to an “unfolding” of $H_E(U)$, it is terminating, sound and complete, just like $H_E(U)$. Thus, $H_{E'}(U \cup U')$ is a unification algorithm for $R \cup E = R \cup F' \cup E'$. □
slepice: Towards a Verified Implementation of Type Theory in Type Theory

František Farkaš
IMDEA Software Institute, Spain
frantisek.farka@imdea.org

Abstract. Dependent types have proven a useful technique for development of verified software. Despite the existence of many systems based in dependent type theory, mostly interactive theorem provers but also programming languages, there is no system that would itself be implemented using dependent types. Recently, a new approach to type inference and term synthesis for type theory with dependent types emerged that separates the process into an analysis phase that is carried out in type theory, and a search phase that is carried out in a logic programming engine. We describe an architecture of type inference and term synthesis engine for a language with dependent types that is based on the new approach and that is feasible to implement using dependently typed language. We demonstrate the architecture by describing slepice, its particular implementation.

Keywords: Dependent types · Type inference · Horn Clause logic · Term synthesis · Proof-relevant resolution

1 Introduction

Dependent type theory has gained its place as foundations for construction of software verification tools in the form of interactive theorem provers and, more recently, in a form of programming languages with dependent types. Yet, current systems that are based in dependent type theory are not themselves implemented using such technology; to give an example, Coq [4] is implemented in OCaml, Agda [15] and Idris [5] in Haskell. At the same time, the need for formally verified tools has been advocated, e.g. by the CompCert project [14]. Such a tool allows to guarantee that there are no compilation-introduced bugs, which is only desirable given that processing languages with dependent types is a complex task.

A canonical example of type theory with dependent types is LF [13]. LF possesses decidable type checking, the metatheory is well-understood, and is strong enough to serve as a basis for a programming language [16]. Urban et al. [20] developed a formalisation of the metatheory of LF that provides an implementation of type checking via code generation. However, even this detailed development is carried out in Isabelle/HOL rather than in a constructive type theory and, as a consequence, the authors study only quasi-decidability of the typing judgement.
For any implementation that is to be of a practical use type checking alone is not sufficient. The amount of type annotations and the number of proof obligations becomes unmanageable very quickly. Some amount of automation is necessary, namely type inference to reconstruct omitted type annotations, and term synthesis to infer omitted proofs. In what follows we use refinement to refer to type inference and term synthesis at once. A detailed account of issues connected with type inference in LF (as implemented in Beluga system [17]) was given by Pientka [16]. Providing such an implementation in constructive type theory is non-trivial since the problem is in general undecidable [7]. Similar issues arise with term synthesis, that accounts for proof automation.

Further, languages that are used in practice are based in stronger type theory than LF. Abel et al. [1] recently formalised a proof of decidability of equality in type theory in Agda. The type theory they consider is an idealised version of Agda itself and the proof can, in principle, be used to extract an implementation of an algorithm for deciding equality in Agda. However, their formalisation depends on inductive-recursive definitions to specify logic-relation that is necessary to proceed by induction and to show that the required assumptions are structurally smaller when proving the main result. Such approach is far from vanilla type theory.

A common objection against the need of having a verified implementation of the refinement engine builds upon Appel’s approach [3] to proof-carrying code—only a kernel that handles type checking is verified while any refinement is handled by a non-verified code. Final type checking by the verified kernel ensures that refinement provides well-formed code. While the approach keeps implementation of such a tool tractable it also has several drawbacks. Among other things, it leads to duplication of code as some functionality is implemented twice, first time in the kernel and second time in the non-verified code. These issues were discussed in a greater detail by Guidi et al. [12]. But more importantly, this leads to a practice when such compiler is the de facto specification of the language—there is no formal specification of the language and even if there were the refinement is not verified to adhere to it. Only the kernel is.

Recently, Farka et al. [9] proposed a new, two-stage approach to refinement. In this approach, a refinement problem consist of a signature $S$ and a term $M$ with metavariables that stand for omitted types and terms (proof obligations). The signature $S$ is translated to a logic program $P$ using refinement calculus and the term $M$ to a goal $G$ while synthesising a type $A$ of $M$. Then, proof-relevant resolution is employed and the goal $G$ is resolved by the program $P$ while computing an answer substitution $\theta$ and a proof term $e$. The answer substitution $\theta$ provides solution to the refinement problem, that is as a refined term $\theta M$ and its type $\theta A$. The computed proof term $e$ is interpreted as a derivation $D = (e, \cdot)_{\theta A}^{\text{der}}$ of well-formedness judgement $S; \cdot \vdash \theta M : \theta A$, that is well-formedness of the solution to the refinement problem. Verification of well-formedness of the refined term then proceeds by straightforward induction on the derivation of the well-formedness judgement. A schematic diagram is listed in Figure 1. In this paper,
we describe an architecture of a refinement engine that is based on the approach, and its particular implementation slepice.

First, a refinement problem is parsed resulting in a pair of inductive objects, abstract syntax representations of a signature and a term. Then, a translation of the signature and the term into a logic program and a goal is formulated as a decidability of the refinement calculus; the calculus is decidable in the sense that either a program and a goal can be constructed or the term is ill-typed. The proof is constructive and proceeds by induction on the structure of the abstract syntax representation of the term. The proof is used to either obtain a program and a goal, if these exists, or to reject ill-typed terms.

The reason that the translation can proceed by simple induction is that all parts that either require a complicated argument, like decidability of equality, or that are in general undecidable, like terms to be substituted for metavariables, are postponed in a form of goals. A resolution engine is used to resolve the goal with the generated program. Guidi et al. [12] investigate a similar approach with λ-Prolog that is solely based in resolution and argue that resolution is suitable to provide an implementation of type checker and elaborator that is comparable to the state-of-art tools. However, their approach does not give a verified implementation. Unlike Guidi et al., we employ proof-relevant resolution. Proof-relevant resolution provides a proof-term that captures a successful resolution of the generated goal. We state a property that the refined term that is obtained from an interpretation of the proof term is well-formed. The proof proceeds by induction on well-formedness derivation that is obtained from the proof-term as well. Proof of the property constitutes a procedure that obtains the refined term.

Finally, formal specification of LF and the refinement calculus gives a basis for the implementation. Data definitions as well as definitions of well-formedness judgements in the type theory are obtained from the formal specification. One can see the refinement as a rudimentary form of elaboration. The refinement calculus then constitutes a formal semantics of the surface language. Further, the generated logic program has in fact two parts; there is a fixed part that is the same for each generated logic program and that constitutes inference rules of the type theory, and there is the part that is given by a particular signature. The static part is directly obtained from the specification as well.

In this paper, we give an account of a system that implements a proof of concept of a refinement engine using the architecture we just described. The im-

Fig. 1. Refinement by proof-relevant resolution
plementation can be found online\(^1\). We use existing tools to instantiate different parts of the described architecture to obtain a verified implementation of type theory in type theory. Namely, we use the Ott tool \([18]\) to specify the grammar, the typing judgement, and the refinement calculus. Ott is also used to generate parser of the source language from the grammar. We use Coq to formally state decidability of the refinement calculus and the interpretation. We use ELPI \([8]\) to mimic proof-relevant resolution. We discuss a particular way to do this and why is it possible in Section 5. Finally, we need to admit that our implementation falls somewhat short of the ideal architecture that is fully hosted by a dependently typed language. The Coq theorem prover does not execute the code directly but uses extraction to OCaml. The definitions and parser generated by Ott are not generated as Coq code but as OCaml code. The ELPI code is interfaced via OCaml as well. To our defence, the amount of OCaml code necessary is fairly small and deals exclusively with interfacing of the components and interaction with the user.

**Contributions** The contributions of this paper is twofold; we

- describe an architecture for a elaboration and type inference engine of a dependently typed language that allows self-hosting, and
- we report on an implementation that uses such architecture and hence manifests feasibility of the approach.

## 2 Specification

In this section, we describe LF \([13]\) that is extended with term- and type-level metavariables, the well-typed fragment of the extended language, and the target logic. The strong point of our approach is that the description is carried out as a formal specification and that definitions in a theorem prover (Coq in our case) and in executable code (OCaml in our case) are generated from the specification. This approach forces a correspondence between formal specification of the language and the implementation. We use Ott tool to formalise the specification. Note that, beside any theorem prover or executable code, a human-readable description is obtained from the formal specification as well\(^2\).

We present the extended language of LF using *de Bruijn* indices representation of variables. We use natural numbers for de Bruijn indices in \(I\), we use identifier \(i\) for individual elements of \(I\) and we denote successor by \(\sigma(-)\). We assume countably infinite disjoint sets \(C\) of *term constants*, and \(B\) of *type constants*. We denote elements of \(C\) by \(c, c', \text{etc.}\) and elements of \(B\) by \(\alpha, \beta, \text{etc.}\). We assume disjoint countable sets of *term-level metavariables* \(?_\text{t}\) and *type-level metavariables* \(?_\text{v}\). For technical reasons, we also assume a countable set of metavariables \(?_\text{p}\) ordered by \(\prec\).

---

1. [github.com/frantisekfarka/slepice](http://github.com/frantisekfarka/slepice)
2. cf. the generated documentation *doc/slepice.pdf* in the implementation
Definition 1 (Extended LF). The syntax of extended terms, extended types, and extended kinds as well as extended signatures and extended contexts is:

\[ T \ni A, B ::= B | T t | \Pi T . T | ?_B \mid ?_T \]  
\[ t \ni M, N ::= C | I | X . t | t t | ?_V \mid ?_T \]  
\[ K \ni L ::= \text{type} | \Pi T . K \]  
\[ Sgn \ni S ::= \cdot | Sgn, C : T | Sgn, B : K \]  
\[ Ctx \ni \Gamma ::= \cdot | Ctx, T \]  

The extended terms include function abstraction \( \lambda A . M \) and application \( M N \). The extended types include \( \Pi \)-type elimination \( AM \) and formation \( \Pi A . B \). We do not include type-level abstraction as it can be safely erased from LF without compromising the expressive power of the calculus \[11\]. We use parenthesis in the rest of the paper in the usual way. An excerpt of Ott source that formalises extended types and terms as well the generated Coq code is listed in Figure 2. Note that the formalisation specifies syntax sugar for parenthesis that is not reflected in the Coq definition. In the actual implementation, there are also some decorations that allows us to extract parser and pretty printer. We omit the decorations here for the sake of readability.

We also give syntactic objects of LF proper as a fragment of the extended language. The formalisation is carried out as a subgrammar of the extended language. The actual representation in the generated theorem prover code is by predicates over extended objects.
Definition 2 (LF). The syntax of terms, types, and kinds as well as signatures and contexts is:

\[
T \ni A, B ::= B | Tt | IT.T \quad \text{types} \\
t \ni M, N ::= C | I | \lambda T.t | tt \quad \text{terms} \\
K \ni L ::= \text{type} | IT.K \quad \text{kinds} \\
Sgn \ni S ::= \cdot | Sgn, C : T | Sgn, B : K \quad \text{signatures} \\
Ctxt \ni \Gamma ::= \cdot | Ctxt, T \quad \text{contexts}
\]

De Bruijn indices are manipulated by shifting; shifting takes a term, type, or kind and an index \(i\) and increments all indices greater than \(i\) by one.

Definition 3. Term and type shifting, denoted by \(- \uparrow^i\) is defined as follows:

\[
\begin{align*}
\epsilon \uparrow^i & \equiv \epsilon \\
\alpha \uparrow^0 & \equiv \sigma \epsilon \\
0 \uparrow^\sigma & \equiv 0 \\
(\sigma \epsilon) \uparrow^{\sigma'} & \equiv \sigma(\epsilon \uparrow') \\
(\lambda A.M) \uparrow^i & \equiv \lambda (A \uparrow^i). (M \uparrow^{\sigma_i}) \\
(MN) \uparrow^i & \equiv (M \uparrow^i)(N \uparrow^i) \\
\end{align*}
\]

Substitution is defined recursively on the structure of objects using index shifting. A substitution of a term \(M\) for an arbitrary index \(i\) increases the index when traversing under a binder.

Definition 4. Term, type, and kind substitution, denoted by \([-N/i]\) is defined as follows:

\[
\begin{align*}
c[N/i] & \equiv c \\
\alpha[N/i] & \equiv \alpha \\
\epsilon[N/i'] & \equiv \begin{cases} N & i = i' \\
\lambda A.M[N/i] & \equiv \lambda (A[N/i]).(M[N\uparrow\sigma_i]) \\
(M_1 M_2)[N/i] & \equiv (M_1[N/i])(M_2[N/i]) \\
\end{cases}
\]

Since shifting with greater index than 0 and substitution for other indices than 0 is not necessary in the inference rules we introduce the following abbreviations:

\[
A \uparrow^0 \equiv A \quad M \uparrow^0 \equiv M \quad A[N] \equiv A[N/0] \quad M[N] \equiv M[N/0]
\]

Well-formed objects of the internal language are given by means of several judgements; \(\vdash S\) for well-formed signatures, \(\vdash \Gamma\) for well-formed contexts, \(\vdash S ; \Gamma \vdash L : \text{type}\) for well-formed kinds, \(\vdash S ; \Gamma \vdash A : L\) for well-formed types of a kind, \(\vdash S ; \Gamma \vdash M : A\) for well-formed terms of a type, \(\vdash S ; \Gamma \vdash A = B : L\) for conversion of types, and \(\vdash S ; \Gamma \vdash M = N : A\) for conversion of terms. In Figure 3 we define judgements for well formed types and terms. We omit definitions of the remaining judgements as these can be found in the documentation generated from the formal specification.
3 Refinement calculus

In this section, we set up the refinement calculus. The refinement calculus formalises the semantics of type inference and term synthesis in the extended language. It can be seen as a rudimentary form of elaboration of a surface language into a core language. In our case the surface language is the extended language that allows to omit some type annotations and proof obligations (i.e. terms) by introducing metavariables instead. The internal language is the language in Definition 2 that does not contain any metavariables.

The refinement calculus relates objects of the extended language and goals in proof-relevant Horn-clause logic. At the same time, it relates signatures of the language and logic programs. First, we give a syntax of the target logic:

**Definition 5.** The syntax of Atomic formulæ, Horn formulæ, programs, proof terms, and goals is:

```
Pc ⊢ c ::= κ_{\text{type}} | κ_C | κ_{\text{id}} | κ_{\text{prim}} | ···

Pt ⊢ e ::= c | e e

At ⊢ ::= \top | \text{eq}_K(K,K,Ctx) | \text{eq}_T(T,T,K,Ctx)
           | \text{eq}_t(t,t,T,Ctx) | \text{type}(T,K,Ctx)
           | \text{term}(t,T,Ctx) | T \uparrow I \equiv T | T[t/I] \equiv T

HC ⊢ ::= At \leftarrow At \land \ldots \land At

P ⊢ P ::= \cdot | P, Pc : HC

G ⊢ G ::= G \land G \parallel At | ?_v : At
```

Proof term constants are distinct names used to identify Horn clauses in programs. Proof terms are applicative terms freely generated from proof term con-
Fig. 4. Refinement of types
Definition 6. Let $\?_T$, $\?_T^\prime$, and $\?_T^{\prime\prime}$ be technical variables. The freshness judgement $\?_T^{\prime\prime} \#_T \?_T$ is defined as follows:

$$\overline{\?_T^{\prime\prime} \#_T \?_T}$$

We introduce an abbreviation for repeated freshness judgements:

$$\overline{\?_T^{\prime\prime} \#_T \?_T^{\prime}\?_T^{\prime\prime} \#_T \?_T^{\prime\prime}\?_T^{\prime\prime\prime} \?_T}$$

Finally, we give a specification of the refinement judgement. This judgement formalises semantics of type inference in the extended grammar. There are mutually defined judgements $S; \Gamma; A \vdash_{\?_T} (G \mid L)$ for refinement of types, and $S; \Gamma; M \vdash_{\?_T} (G \mid A)$ for refinement of terms. The arguments on the left hand side of the dash, that is a signature $S$, an extended context $\Gamma$, an extended type $A$ or an extended term $M$, and a technical variable $\?_T$, are seen as inputs. The arguments on the right hand side, that is a technical variable $\?_T$, a goal $G$, and an extended kind $L$ or an extended type $A$ are seen as outputs. The judgements are defined in Figures 4 and 5. We use $A_1 \equiv A_2$ and $A_1[N] \equiv A_2$ to abbreviate atoms $A_1 \equiv A_2$ and $A_1[N/0] \equiv A_2$, respectively and similarly for terms in allusion to abbreviations we introduced for shifting and substitution.

We show decidability of the term and type refinement judgements in the next section and this also justifies our identification of arguments of the judgement as inputs and outputs. A goal that is produced by refinement translation is solved by a logic program. The program is obtained from a signature. We define judgement $S; \Gamma; P \vdash_{\?_T}$. A signature $S$ and a technical variable $\?_T$ are seen as inputs and the technical variable $\?_T$ and a program $P$ are seen as outputs. The judgement is defined in Figure 6. The empty signature is refined into the initial program $P_e$ that captures static inference rules in in Figure 3. The remaining two rules then extend program with an instance of the inference rules $\text{tcon}$ and $\text{con}$ respectively and instantiate shifting and substitution with type and term constants.
We use the formal specification of refinement judgements in Figures 4, 5, and 6 to obtain definitions on Coq that are used for stating the decidability results. We illustrate the extracted definitions on an excerpt of Coq code in Figure 7. The definitions in executable OCaml code are extracted directly from the specification in order as we extract parser of the input language from the specification as well. Coq definitions are then explicitly mapped to extracted OCaml definitions in Coq code extraction.

\[
\begin{align*}
S, a : L &\vdash \gamma_f P, \kappa_a : type(a, L, ?) \leftarrow, \kappa_{\text{a}[\text{?}]} : (a[?/?\text{?}]) \equiv a \leftarrow \\
S, c : A &\vdash \gamma_f P, \kappa_c : \text{term}(c, A, ?) \leftarrow, \kappa_{\text{c}[\text{?}]} : (c[?/?\text{?}]) \equiv c \leftarrow
\end{align*}
\]

Fig. 6. Refinement of signatures

4 Decidability of Refinement

In this section we prove decidability of the refinement judgements that relate extended types and terms to goals and signatures to programs. The proofs are formalised in Coq theorem prover and serve, after code extraction, as functions that perform the core generation of goals and programs.
The first intermediate result we need to prove in our formalisation is that equality of syntactic objects of the extended language is decidable.

**Proposition 1.**

1. Let $A$, $B$ be extended types. Then either $A = B$ or $A \neq B$.
2. Let $M$, $N$ be extended terms. Then either $M = N$ or $M \neq N$.
3. Let $L$, $L'$ be extended kinds. Then either $L = L'$ or $L \neq L'$.

**Proof.** Parts 1 and 2 proceed by mutual induction on the type and the term. Part 3 proceeds by induction using part 1.

We also need to show that whether a type of a certain kind or a term constant of a certain type is bound in a signature is decidable.

**Lemma 1.**

- Let $S$ be a signature, and $c$ a term constant. Then either there is a type $A$ such that $c : A \in S$ or, for all $A$, $c : A \in S$ is impossible.
- Let $S$ be a signature, and $\alpha$ a type constant. Then either there is an extended kind $L$ such that $\alpha : L \in S$ or, for all $L$, $\alpha : L \in S$ is impossible.

**Proof.** By induction on signature using decidability of equality of terms and types (Proposition 1).

Now we could state the main theorem that the refinement judgement for terms and types is decidable. However, there is a caveat. The refinement judgements for terms and for types are mutually defined and hence the extracted Coq definitions are mutually defined as well as we demonstrate in Figure 7. A proof by naive induction fails as Coq cannot establish that recursive calls are structurally smaller. We devise mutually recursive inductive types that we call structure of extended types and extended terms and a mapping from extended types and extended terms to the respective structure.

**Definition 7.** The syntax of structure of extended types and structure of extended terms is:

$$
S_T \ni s_A := \cdot | \Pi S_T.S_T | S_T.S_t \quad \text{structure of extended types}
$$

$$
S_t \ni s_M := \cdot | \Pi S_t.S_t | S_t.S_t \quad \text{structure of extended terms}
$$

**Definition 8.** We define mappings $(-)^s : T \rightarrow S_T$ and $(-)^s : t \rightarrow S_t$ by

\[
\begin{align*}
(\alpha)^s &= \cdot \\
(\Pi A.B)^s &= \Pi(A)^s.(B)^s \\
(A.M)^s &= (A)^s(M)^s \\
(?_A)^s &= \cdot \\
(?_T)^s &= \cdot \\
(\alpha)^s &= \cdot \\
(\Pi A.M)^s &= \Pi(A)^s.(M)^s \\
(M.N)^s &= (M)^s(N)^s \\
(?_M)^s &= \cdot \\
(?_T)^s &= \cdot
\end{align*}
\]
Note that by an abuse of notation we do not distinguish between the names of the mapping from types and the mapping from terms. The more general statement of decidability of refinement is stated using the structure.

**Theorem 1 (Decidability of Refinement).**

- Let $s_M$ be a structure, $S$ a signature, $\Gamma$ an extended context, and $M$ an extended term. If $(M)^s = s_M$, then either there is a goal $G$ and an extended type $A$ such that $S; \Gamma; M \vdash (G | A)$ or, for any goal $G$ and any type $A$, $S; \Gamma; M \vdash (G | A)$ is impossible.

- Let $s_A$ be a structure, $S$ a signature, $\Gamma$ an extended context, and $A$ an extended type. If $(A)^s = s_A$, then either there is a goal $G$ and a kind $L$ such that $S; \Gamma; A \vdash (G | L)$ or, for any goal $G$ and any kind $L$, $S; \Gamma; A \vdash (G | L)$ is impossible.

**Proof.** By mutual induction on structure of the term $s_M$ and structure of the type $s_A$ using Proposition 1 and Lemma 1.

The intended statement of the refinement theorem for terms and types then follows as a corollary.

**Corollary 1 (Goal construction).**

- Let $S$ be a signature, $\Gamma$ an extended context, and $M$ an extended term. Either there is a goal $G$ and an extended type $A$ such that $S; \Gamma; M \vdash (G | A)$ or, for any goal $G$ and any type $A$, $S; \Gamma; M \vdash (G | A)$ is impossible.

- Let $S$ be a signature, $\Gamma$ an extended context, and $A$ an extended type. Either there is a goal $G$ and an extended kind $L$ such that $S; \Gamma; A \vdash (G | L)$ or, for any goal $G$ and any kind $L$, $S; \Gamma; A \vdash (G | L)$ is impossible.

Similarly, we state a decidability result for refinement of signatures that will allow us to obtain programs that resolve goals generated from extended types and terms.

**Theorem 2 (Refinement of Signature).** Let $S$ be a signature. Either there is a program $P$ such that $S \vdash P$ or, for any $P$, $S \vdash P$ is impossible.

**Proof.** By induction on signature $S$.

Formalisation of proofs of the above theorems provides a procedures that take terms and types and generate goals and that take signature and generate program. OCaml signatures of the extracted code that correspond to the above theorems are listed in Figure 8. Signatures $Sgn$ are extracted as the type $\text{sgn}$, extended contexts $Ctx$ as $\text{ctx}$. Structure of extended types $S_T$ is extracted as $\text{sTy}$, extended types $T$ as $\text{eTy}$, similarly for terms and kinds. Type level metavariables $?_B$ and term level metavariables $?_V$ are extracted as $\text{lvar}$ and technical metavariables as $\text{tvar}$, goals $G$ and programs $P$ as $\text{goal}$ and $\text{prog}$ respectively.
5 Proof-relevant Resolution

In this section we describe our realisation of proof-relevant resolution and interpretation of answer substitutions and computed proof terms. As a resolution engine in our implementation we resort to ELPI \[8\]. Although ELPI is not proof-relevant resolution engine, it is sufficient for our purposes. In this work we are not interested in finer details of the resolution mechanism (cf. \[9\], \[10\]) and we can obtain sound results by a simple syntactic transformation. In this paper, we omit details of the transformation and focus on interpretation of the computed assignment to type and term level metavariables and on interpretation of computed proof terms. In the following, we assume that the proof relevant resolution for a generated goal \(G\) and a program \(P\) either computes an answer substitution \(\theta\) and, for each atomic subgoal, a proof-term \(e\) or fails.

First, we extend application of computed substitution to extended types and extended terms in the usual way.

**Definition 9.** We define application of a substitution \(\theta\) by

\[
\begin{align*}
\theta(\alpha) &= \alpha & \theta(c) &= c \\
\theta(\Pi A.B) &= \Pi \theta(A).\theta(B) & \theta(\iota) &= \iota \\
\theta(A.M) &= \theta(A).\theta(M) & \theta(\lambda A.M) &= \lambda \theta(A).\theta(M) \\
\theta(\?_A) &= \theta(\?_A) & \theta(\?_M) &= \theta(\?_M) \\
\theta(\?_T) &= \?_T & \theta(\?_T) &= \?_T
\end{align*}
\]

By Definition 5 of syntax of the target logic proof terms are computed for atomic (sub-)goals. We define an interpretation of proof terms that construct a derivation of a well-formedness judgement from such a proof term. We use \(S; \Gamma \vdash I\) to jointly refer to the judgements of LF in the usual way.

**Definition 10.** Let \(S\) be a signature, and \(\Gamma\) a context such that \(S \vdash \Gamma\) and let \(S; \Gamma \vdash I\) be a judgement, and \(e\) a proof term. The interpretation of the proof
term \((e, \Gamma)\}_{\text{der}}^\kappa\) is defined as follows:

\[
(\kappa, \Gamma)_{\alpha:L}^\text{der} = \frac{S \vdash \alpha : L}{S; \Gamma \vdash \alpha : L} \quad \text{T-CON}
\]

\[
(\kappa_{\Pi\text{-intro}} e_1 e_2, \Gamma)_{\Pi A.B : \text{type}}^\text{der} = \frac{(e_1, \Gamma)_{\alpha:A;B_1.L}^\text{der} (e_2, \Gamma)_{\alpha:B_2}^\text{der} \quad (e_3, \Gamma)_{\alpha:B_1 = B_2; \text{type}}^\text{der}}{S; \Gamma \vdash \Pi A.B : \text{type}} \quad \text{T-\Pi\text{-intro}}
\]

\[
(\kappa_{\Pi\text{-elim}} e_1 e_2 e_3, \Gamma)_{\lambda M.A.M[L[M]}^\text{der} = \frac{(e_1, \Gamma)_{\alpha:M}^\text{der} (e_2, \Gamma)_{\alpha:A}^\text{der} (e_3, \Gamma)_{\alpha:B_1 = B_2; \text{type}}^\text{der}}{S; \Gamma \vdash \lambda A.M : L[M]} \quad \text{T-\Pi\text{-elim}}
\]

\[
(\kappa_{\text{con}} e_1 e_2, \Gamma)_{\alpha:L}^\text{der} = \frac{S \vdash \alpha : L}{S; \Gamma \vdash \alpha : L} \quad \text{T-con}
\]

\[
(\kappa_{\text{zero}} e_1 e_2, \Gamma)_{\alpha:L}^\text{der} = \frac{S \vdash \alpha : L}{S; \Gamma \vdash \alpha : L} \quad \text{T-zero}
\]

\[
(\kappa_{\text{succ}} e_1 e_2, \Gamma)_{\alpha:L}^\text{der} = \frac{S \vdash \alpha : L}{S; \Gamma \vdash \alpha : L} \quad \text{T-succ}
\]

\[
(\kappa_{\text{der}} e_1 e_2, \Gamma)_{\alpha:L}^\text{der} = \frac{S \vdash \alpha : L}{S; \Gamma \vdash \alpha : L} \quad \text{T-der}
\]

\[
(\kappa_{\text{type}} e_1 e_2, \Gamma)_{\alpha:L}^\text{der} = \frac{S \vdash \alpha : L}{S; \Gamma \vdash \alpha : L} \quad \text{T-type}
\]

\[
(\kappa_{\text{term}} e_1 e_2, \Gamma)_{\alpha:L}^\text{der} = \frac{S \vdash \alpha : L}{S; \Gamma \vdash \alpha : L} \quad \text{T-term}
\]

The definition is easiest to understand as a definition of function \((-\, -)^\text{der}\) that constructs a derivation by pattern matching on its first argument, a proof term. The cases are discriminated by the head symbol of the proof term—each corresponding to one inference rule of the internal language—and there is a subderivation to be computed, the function call itself recursively on the appropriate subterms of the proof term.

The above definition lists only cases of proof-terms with head symbols that correspond to inference rules in Figure 3. We omit the remaining cases for well-formedness of contexts and equality judgements for the sake of brevity and since we do not list these rules in the paper. These omitted cases are straightforward and are properly handled in the formalisation.

In Lemma 1, we have already proven that whether a type constant is bound in a signature as a particular kind, that is whether \(\alpha : L \in S\) is decidable. We extend this result to decidability of all judgements involved in Definition 10. Hence we can verify whether proof-relevant resolution produces well-formed types and terms by manifesting a derivation of the well-formedness judgement.

**Theorem 3.**

- Let \(e\) be a proof term, \(S\) a signature, \(\Gamma\) a context, \(M, N\) terms, and \(A\) a type. Then either \((e)_{M = N : A}^\text{der}\) is well-formed or \((e)_{M = N : A}^\text{der}\) is impossible.
- Let \(e\) be a proof term, \(S\) a signature, \(\Gamma\) a context, \(A, B\) types, and \(L\) a kind. Then either \((e)_{\alpha : B ; L}^\text{der}\) is well-formed or \((e)_{\alpha : B ; L}^\text{der}\) is impossible.

**Proof.**

- By induction on \(e\) using part 2.
- By induction on \(e\) using part 1.
**Theorem 4.** Let $e$ be a proof term, $\theta$ a substitution of metavariables, $S$ a signature, $M$ an extended term, and $A$ an extended type.

Then either $(e)_{\theta M, \theta A}^{\text{der}}$ is well-formed or $(e)_{\theta M, \theta A}^{\text{der}}$ is impossible.

**Proof.** By induction using Lemma 1 and Theorem 3.

This theorem concludes our exposition of the interpretation of proof terms that are computed by proof-relevant resolution. When the formalised proof is extracted into OCaml it provides a procedure for verification of solution computed by proof-relevant resolution and hence manifests soundness of the system.

## 6 Related Work

Type inference and term synthesis as discussed in this paper is mechanically obtained from a specification of a type system in the form of typing judgements. Such approach does not exist in the literature yet. However, the importance of such treatment of type inference and term synthesis can be clearly argued based on the work currently being carried out for languages as Coq and Agda. The main relevant project is MetaCoq [19, 2]. The project aims to provide certified metaprogram facilities for Coq.

Building on MetaCoq, Sozeau et al. provide a verified implementation of type checker. They as well need to carry out certain amount of type inference. However, the amount is limited by the fact that they work only with a kernel of Coq (in our terms the internal language), i.e. a limited internal language that has already been elaborated, and by the fact that they assume that the metatheory is sound and hence the language is strongly normalising (and, as a result, typechecking is decidable).

In Agda, there is work being currently done on type-save metaprograming, albeit it is in less mature state than in Coq. Cockx [6] has introduced type-safe rewriting rules, a type of reflection that is restricted to equality. Due to the restriction, there is no need for type inference and term synthesis. We conjecture that for full-scale metaprograming it will be necessary as is the case with Coq.

## 7 Conclusion

Our formalisation of type inference and term synthesis for LF is carried out in Off, which is used to generate the OCaml code, a parser of the input and Coq definition. We utilise type inference and term synthesis by translation to proof-relevant resolution and formally prove decidability of the translation, give interpretations to the computed proof terms and show their soundness.

Although our implementation is not fully carried out in a dependently typed language, that is Coq in our case, the amount of OCaml code that is necessary is very small. Such code is necessary only for interfacing different components of the system. The portion of hand-written OCaml code is very small and we believe this makes our approach superior to current implementations of dependently typed languages. We believe that the architecture we just introduced can
serve as viable basis both for obtaining reference implementations from formal specifications of a programming languages and, with properly optimised resolution phase, as a basis for a type inference engine.

References


Resourceful Program Synthesis from Graded Linear Types

Jack Hughes and Dominic Orchard
School of Computing, University of Kent

Abstract. Linear types provide a way to constrain programs by specifying that some values must be used exactly once. Recent work on graded modal types augments and refines this notion, enabling fine-grained, quantitative specification of data use in programs. The information provided by graded modal types appears to be useful for type-directed program synthesis, where these additional constraints can be used to prune the search space of candidate programs. We explore one of the major implementation challenges of a synthesis algorithm in this setting: how does the synthesis algorithm efficiently ensure that resource constraints are satisfied throughout program generation? We provide two solutions to this resource management problem, adapting Hodas and Miller’s input-output model of linear context management to a graded modal linear type theory. We evaluate the performance of both approaches via their implementation as a program synthesis tool for the programming language Granule, which provides linear and graded modal typing.

1 Introduction

Type-directed program synthesis is a long-studied technique rooted in automated theorem proving [27]. A type-directed synthesis algorithm can be constructed as an inversion of type checking, starting from a type and inductively synthesising well-typed subterms, pruning the search space via typing. Via the Curry-Howard correspondence [20], we can view this as proof search in a corresponding logic, where the goal type is a proposition and the synthesised program is its proof. Recent work has extended type-directed synthesis to refinement types [32], cost specifications [25], differential privacy [33], and example-guided synthesis [11,31].

Automated proof search techniques have been previously adapted to linear logics, accounting for resource-sensitive reasoning [7–9,19,29]. By removing the structural rules of contraction and weakening, linear logic allows propositions to be treated as resources that must be used exactly once [16]. Non-linear propositions are captured via the ‘exponential’ modality !. Linearity introduces a new dimension to proof search and program synthesis: how do we inductively generate terms whilst pruning the search space of those which violate linearity? For example, consider the following inductive synthesis rule, mirroring Gentzen’s sequent calculus [14], which synthesises a term of type $A \otimes B$:

\[
\begin{align*}
\Gamma_1 \vdash A & \Rightarrow t_1 \\
\Gamma_2 \vdash B & \Rightarrow t_2
\end{align*}
\]

\[
\Gamma_1, \Gamma_2 \vdash A \otimes B \Rightarrow \langle t_1, t_2 \rangle \quad \text{PAIR}
\]
Reading the rule bottom up: from a context of assumptions $\Gamma_1, \Gamma_2$ we can synthesise the pair $\langle t_1, t_2 \rangle$ from the product type $A \otimes B$ provided that we can inductively synthesise the subterms of the pair, using $\Gamma_1$ for the left side and $\Gamma_2$ for the right.

But how do we partition a context of free variables $\Gamma$ into $\Gamma_1$ and $\Gamma_2$ such that $\Gamma_1$ contains only those variables needed by $t_1$ and $\Gamma_2$ only those for $t_2$? A naive approach is to try every possible partition of $\Gamma$. However, this becomes unmanageable as the number of possible partitions is $2^{\lvert \Gamma \rvert}$, i.e., exponential in the number of assumptions. This issue has been explored in automated theorem proving for linear logic, and is termed the resource management problem [7].

To address this, Hodas and Miller described an input-output context management scheme for linear logic programming [19], further developed by Cervesato et al. [7]. In this approach, synthesis rules take the form $\Gamma \vdash A \Rightarrow t; \Delta$ with an input context $\Gamma$ and an output context $\Delta$ which contains all the hypotheses of $\Gamma$ that were not used in the proof $t$ of $A$ (akin to the notion of left over typing for linear type systems [2,34]). This output context is then used as the input context to subsequent subgoals. In the case of $A \otimes B$, synthesis has the form:

$$
\frac{\Gamma \vdash A \Rightarrow t_1; \Delta_1 \quad \Delta_1 \vdash B \Rightarrow t_2; \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow \langle t_1, t_2 \rangle; \Delta_2} \text{ PAIR \_ LEFTOVER}
$$

The non-determinism of how to divide $\Gamma$ is resolved by using the entire context as the input for the synthesis of the first subterm $t_1$ from type $A$. If this succeeds, the context $\Delta_1$ is returned containing the resources not needed to construct $t_1$. These remaining resources provide the input context to synthesise $t_2$ from $B$, which in turn returns an output context $\Delta_2$ containing the resources not used by the pair $\langle t_1, t_2 \rangle$. We extend this approach, which we term subtractive resource management, to graded modal types and present its dual: additive resource management. In the additive approach, the output context describes what resources were used to synthesise a term, rather than what may still be used.

Graded modal types comprise an indexed family of modal operators whose indices have structure capturing program properties [30]. In the context of linear logic, graded modalities generalise the indexed modality of Bounded Linear Logic [17] $!_{r}A$ where $r \in \mathbb{N}$ captures the upper bound $r$ on the number of times $A$ is used. Generalising such indices to an arbitrary (pre-ordered) semiring yields a type system which can be instantiated to track various properties via the graded modality, a technique which is increasingly popular [4,12,13,15,22,23,30,34].

Our primary contribution is the extension of the input-output model of resource management for linear program synthesis to graded modal types. Our input and output contexts contain both linear and graded assumptions. Graded assumptions are annotated with a grade: an element of a pre-ordered semiring describing the variable’s use. For example, grades drawn from $\mathbb{N}$ yield a system akin to BLL which counts the number of times a variable is used, where a graded assumption $x : [A]_2$ means $x$ can be used twice. An example instantiation of our subtractive pair introduction rule is then as follows:

$$
\frac{\Gamma, x : [A]_2 \vdash A \Rightarrow x; \Gamma, x : [A]_1 \quad \Gamma, x : [A]_1 \vdash A \Rightarrow x; \Gamma, x : [A]_0}{\Gamma, x : [A]_2 \vdash A \otimes A \Rightarrow (x,x); \Gamma, x : [A]_0} \text{ PAIR \_ LEFTOVER}
$$
The initial input context contains graded assumption \( x : [A]_2 \). The first premise synthesises the term \( x \), returning an output context which contains the assumption \( x \) with grade 1, indicating that \( x \) has been used once and can be used one more time. The next premise synthesises the second part of the pair as \( x \) using its remaining use. In the final output context, \( x \) is graded by 0, preventing it from being used to synthesise subsequent terms.

We adapt the input-output model of linear logic synthesis to subtractive and additive approaches in the presence of graded modal types, pruning the search space via the quantitative constraints of grades. We develop a type-directed synthesis tool for Granule, a functional language which combines indexed, linear, and graded modal types. Granule supports various graded modalities, and its type checker leverages the Z3 SMT solver to discharge constraints on grades.

We develop a type-directed synthesis tool for Granule, a functional language which combines indexed, linear, and graded modal types. Granule supports various graded modalities, and its type checker leverages the Z3 SMT solver to discharge constraints on grades. As type-based synthesis follows the structure of types, it is necessary to solve equations on grades during synthesis, for which we make use of Granule’s SMT integration. Such calls to an external prover are costly, and thus efficiency of resource management is a key concern.

Section 2 introduces our core type theory (a subset of Granule’s type system) based on the linear \( \lambda \)-calculus extended with graded modal types, pairs, and sums. Section 3 describes the two core synthesis calculi (subtractive and additive) as augmented inversions of the typing rules, as well as a variant of additive synthesis. Section 4 describes the implementation and gives a quantitative comparison of the synthesis techniques on a suite of benchmark programs. The main finding is that the additive approach is often more efficient than the subtractive, presenting a departure from the literature on linear logic theorem proving which is typically subtractive.

Throughout, we will tend towards using types-and-programs terminology rather than propositions-and-proofs. Through the Curry-Howard correspondence, one can switch smoothly to viewing our approach as proof search in logic.

## 2 Graded linear \( \lambda \)-calculus

Our focus is a linear \( \lambda \)-calculus akin to a simply-typed linear functional language with graded modalities, resembling the core languages of Gaboardi et al. and Brunel et al., and a simply-typed subset of Granule.

Types comprise linear functions, multiplicative conjunction (product types \( \otimes \) and unit 1), additive disjunction (sum types \( \oplus \)), and a graded modality \( \Box_r \):

\[
A, B ::= A \to B \mid A \otimes B \mid A \oplus B \mid 1 \mid \Box_r A
\]

(types)

where \( \Box_r A \) is an indexed family of type operators where \( r \) ranges over the elements of some pre-ordered semiring \( (\mathcal{R}, *, 1, +, 0, \sqsubseteq) \) parameterising the calculus (where * and + are monotonic with respect to the pre-order \( \sqsubseteq \)).

---

1. [https://github.com/granule-project/granule/releases/tag/v0.7.8.0](https://github.com/granule-project/granule/releases/tag/v0.7.8.0)
The syntax of terms provides the elimination and introduction forms:
\[
\begin{align*}
t &: = x | \lambda x.t | \ let \ t_1 t_2 | [t] | \ let \ [x] = t_1 \ let \ t_2 | \ let \ (x_1, x_2) = t_1 \ in \ t_2 \\
& | () | \ let \ () = t_1 \ let \ t_2 | \ inl \ t | \ inr \ t | \ case \ t_1 \ of \ inl \ x_1 \ to \ t_2 | \ inr \ x_2 \ to \ t_3 \quad (\text{terms})
\end{align*}
\]

We use the syntax () for the inhabitant of multiplicative unit 1. Pattern matching via a let is used to eliminate products and unit types; for sum types, case is used to distinguish the constructors. The construct \([t]\) introduces a graded modal type \(\square, A\) by ‘promoting’ a term \(t\) to the graded modality, and let \([x] = t_1 \ in \ t_2\) eliminates a graded modal value \(t_1\), binding a graded variable \(x\) in scope of \(t_2\).

Typing judgments are of the form \(\Gamma \vdash t : A\), where \(\Gamma\) ranges over contexts:
\[
\Gamma ::= \emptyset \mid \Gamma, x : A \mid \Gamma, x : [A]_r \quad (\text{contexts})
\]

Thus, a context may be empty \(\emptyset\), extended with a linear assumption \(x : A\) or extended with a graded assumption \(x : [A]_r\). For linear assumptions, structural rules of weakening and contraction are disallowed. Graded assumptions may be used non-linearly according to the constraints given by their grade, the semiring element \(r\). Throughout, comma denotes disjoint context concatenation.

Various operations on contexts are used to capture non-linear data flow via grading. Firstly, context addition provides an analogue to contraction, combining contexts that have come from typing multiple subterms in a rule. Context addition, written \(\Gamma_1 + \Gamma_2\), is undefined if \(\Gamma_1\) and \(\Gamma_2\) overlap in their linear assumptions. Otherwise graded assumptions appearing in both contexts are combined via the semiring \(+\) of their grades.

**Definition 1 (Context addition).** For all \(\Gamma_1, \Gamma_2\) context addition is defined as follows by ordered cases matching inductively on the structure of \(\Gamma_2\):
\[
\Gamma_1 + \Gamma_2 = \begin{cases} 
\Gamma_1 & \Gamma_2 = \emptyset \\
((\Gamma'_1, \Gamma''_1) + \Gamma'_2), x : [A]_{r+s} & \Gamma_2 = \Gamma'_2, x : [A]_s \land \Gamma_1 = \Gamma'_1, x : [A]_r, \Gamma''_1 \\
(\Gamma_1 + \Gamma'_2), x : A & \Gamma_2 = \Gamma'_2, x : A \land x : A \notin \Gamma_1 
\end{cases}
\]

In the typing of case expressions, the least-upper bound of the two contexts used to type each branch is used, defined:

**Definition 2 (Partial least-upper bounds of contexts).** For all \(\Gamma_1, \Gamma_2\):
\[
\Gamma_1 \sqcup \Gamma_2 = \begin{cases} 
(\emptyset \sqcup \Gamma''_2), x : [A]_{r+s} & \Gamma_1 = \emptyset \land \Gamma_2 = \emptyset \\
(\Gamma'_1 \sqcup (\Gamma'_2, \Gamma''_2)), x : A & \Gamma_1 = \Gamma'_1, x : A \land \Gamma_2 = \Gamma'_2, x : A, \Gamma''_2 \\
(\Gamma'_1 \sqcup (\Gamma'_2, \Gamma''_2)), x : [A]_{r+s} & \Gamma_1 = \Gamma'_1, x : [A]_r \land \Gamma_2 = \Gamma'_2, x : [A]_s, \Gamma''_2 
\end{cases}
\]

where \(r \sqcup s\) is the least-upper bound of grades \(r\) and \(s\) if it exists, derived from \(\sqcup\).

As an example of the partiality of \(\sqcup\), if one branch of a case uses a linear variable, then the other branch must also use it to maintain linearity overall, otherwise the upper-bound of the two contexts for these branches is not defined.
\[
\begin{array}{c}
\text{VAR} \quad x : A \vdash x : A \\
\text{Abs} \quad \Gamma \vdash \lambda x : B \rightarrow \Gamma \vdash \lambda x . t : A \rightarrow B \\
\text{APP} \quad \Gamma \vdash t_1 : A \rightarrow B \quad \Gamma \vdash t_2 : A \\
\Gamma, [\Delta]_0 \vdash t : A \\
\text{Weak} \quad \Gamma, x : A \vdash t : B \\
\Gamma, x : [A] \vdash t : B \\
\text{DER} \quad r \ast [\Gamma] \vdash [t] : [r \ast A] \\
\text{Pr} \quad \Gamma \vdash t : A \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash t_1 : \Box r A \\
I_1 \vdash x : [A] \vdash t_2 : B \\
\text{Let} \quad \Gamma_1 \vdash t_1 : \Gamma_1 \vdash \gamma : r \ast \Gamma_1 \\
\text{Let1} \quad \text{LET} \quad \Gamma_1 \vdash [] \vdash 1 \\
\Gamma_1 \vdash t_1 : \Gamma_1 \vdash t_2 : A \\
\Gamma_1 \vdash t_1 : \Gamma_1 \vdash t_2 : B \\
\text{PAIR} \quad \Gamma \vdash t_1 : [A] \vdash t_2 : A \\
\Gamma \vdash t_1 : [A] \vdash t_2 : A \\
\text{APPROX} \quad \Gamma \vdash t_1 : [A]_s \vdash t_2 : A \\
\Gamma \vdash t_1 : [A]_s \vdash t_2 : A \\
\Gamma \vdash t_1 : [A]_s \vdash t_2 : A \\
\text{INR} \quad \Gamma \vdash t_1 : [A] \vdash t_2 : A \oplus B \\
\Gamma \vdash t_1 : [A] \vdash t_2 : A \oplus B \\
\text{CASE} \quad \Gamma \vdash t_1 : [A] \vdash t_2 : C \\
\Gamma \vdash t_1 : [A] \vdash t_2 : C \\
\end{array}
\]

Figure 1 defines the typing rules. Linear variables are typed in a singleton context (VAR). Abstraction (Abs) and application (APP) follow the rules of the linear \(\lambda\)-calculus. Rules for multiplicative products (pairs) and additive co-products (sums) are routine, where pair introduction (PAIR) adds the contexts used to type the pair’s constituent subterms. Pair elimination (LETPAIR) binds a pair’s components to two linear variables in the scope of the body \(t_2\). The INL and INR rules handle the typing of constructors for the sum type \(A \oplus B\). Elimination of sums (CASE) takes the least upper bound (defined above) of the contexts used to type the two branches of the case.

The WEAK rule captures weakening of assumptions graded by 0 (where \([\Delta]_0\) denotes a context containing only graded assumptions graded by 0). Dereliction (DER), allows a linear assumption to be converted to a graded assumption with grade 1. Grade approximation is captured by the APPROX rule, which allows a grade \(s\) to be converted to another grade \(r\), providing that \(r\) approximates \(s\), where the relation \(\sqsubseteq\) is the pre-order provided with the semiring. Introduction and elimination of the graded modality is provided by the Pr and LET rules respectively. The Pr rule propagates the grade \(r\) to the assumptions through scalar multiplication of \(\Gamma\) by \(r\) where every assumption in \(\Gamma\) must already be graded (written \([\Gamma]\) in the rule), defined:

**Definition 3 (Scalar context multiplication).**

\[
r \ast \emptyset = \emptyset \\
r \ast ([\Gamma], x : [A]_s) = (r \ast \Gamma), x : [A]_{(r \ast s)}
\]

The LET rule eliminates a graded modal value \(\Box r A\) into a graded assumption \(x : [A]_s\) with a matching grade in the scope of the let body.

We now give three examples of different graded modalities.
Example 1. The natural number semiring with discrete ordering \((\mathbb{N}, \ast, 1, +, 0, \equiv)\) provides a graded modality that counts exactly how many times non-linear values are used. As a simple example, the \(\mathcal{S}\) combinator is typed and defined:

\[
\begin{align*}
  s &: (A \rightarrow (B \rightarrow C)) \rightarrow (A \rightarrow B) \rightarrow (\square_2 A \rightarrow C) \\
  s &= \lambda x . \lambda y . \lambda z . \text{let } [z] = z' \text{ in } (x z)(y z)
\end{align*}
\]

The graded modal value \(z'\) captures the ‘capability’ for a value of type \(A\) to be used twice. This capability is made available by eliminating \(\square\) (via \text{let}) to the variable \(z\), which is graded \(z : [A]_2\) in the scope of the body.

Example 2. Exact usage analysis is less useful when control-flow is involved, e.g., eliminating sum types where each control-flow branch uses variables differently. The above \(\mathbb{N}\)-semiring can be imbued with a notion of approximation via less-than-equal ordering, providing upper bounds. A more expressive semiring is that \(\mathbb{N}\) of natural number intervals \([30]\), given by pairs \((\leq)\) than-equal ordering, providing upper bounds. A more expressive semiring is that \(\mathbb{N}\) of natural number intervals \([30]\), given by pairs \((\leq)\) than-equal ordering, providing upper bounds. A more expressive semiring is that \(\mathbb{N}\) of natural number intervals \([30]\), given by pairs \((\leq)\) than-equal ordering, providing upper bounds. A more expressive semiring is that \(\mathbb{N}\) of natural number intervals \([30]\), given by pairs \((\leq)\) than-equal ordering, providing upper bounds.

Example 3. Graded modalities can capture a form of information-flow security, tracking the flow of labelled data through a program \([30]\), with a lattice-based semiring on \(\mathcal{R} = \{\text{Unused} \subseteq \mathcal{H} \subseteq \text{Lo}\}\) where \(0 = \text{Unused}, 1 = \mathcal{H}, + = \mathcal{H}\) and if \(r = \text{Unused}\) or \(s = \text{Unused}\) then \(r * s = \text{Unused}\) otherwise \(r * s = \mathcal{H}\). This allows the following well-typed program, eliminating a pair of \(\text{Lo}\) and \(\mathcal{H}\) security values, picking the left one to pass to a continuation expecting a \(\text{Lo}\) input:

\[
\begin{align*}
  \text{noLeak} &: (\square_\text{Lo} A \otimes \square_\mathcal{H} A) \rightarrow (\square_\text{Lo} (A \otimes 1) \rightarrow B) \rightarrow B \\
  \text{noLeak} &= \lambda z . \lambda u . \text{let } (x', y') = z \text{ in } \text{let } x = x' \text{ in } \text{let } [y] = y' \text{ in } [u ([x, ()])]
\end{align*}
\]

Metatheory. The admissibility of substitution is a key result that holds for this language \([30]\), which is leveraged in soundness of the synthesis calculi.

Lemma 1 (Admissibility of substitution). Let \(\Delta \vdash t' : A\), then:

- \(\text{(Linear)}\) If \(\Gamma, x : A, \Gamma' \vdash t : B\) then \(\Gamma + \Delta + \Gamma' \vdash [t'/x]t : B\)
- \(\text{(Graded)}\) If \(\Gamma, x : [A]_r, \Gamma' \vdash t : B\) then \(\Gamma + (r * \Delta) + \Gamma' \vdash [t'/x]t : B\)

3 The synthesis calculi

We present two synthesis calculi with subtractive and additive resource management schemes, extending an input-output approach to graded modal types. The
structure of the synthesis calculi mirrors a cut-free sequent calculus, with left and right rules for each type constructor. Right rules synthesise an introduction form for the goal type. Left rules eliminate (deconstruct) assumptions so that they may be used inductively to synthesise subterms.

3.1 Subtractive Resource Management

Our subtractive approach follows the philosophy of earlier work on linear logic proof search \[7, 19\], structuring synthesis rules around an input context of the available resources and an output context of the remaining resources that can be used to synthesise subsequent subterms. Synthesis rules are read bottom-up, with judgments \( \Gamma \vdash A \Rightarrow - t; \Delta \) meaning from the goal type \( A \) we can synthesise a term \( t \) using assumptions in \( \Gamma \), with output context \( \Delta \). We describe the rules in turn to aid understanding. Appendix \[X\] collects the rules for reference.

Variable terms can be synthesised from linear or graded assumptions by rules:

\[
\begin{align*}
\text{LinVar}^- & : \Gamma, x : A \vdash A \Rightarrow - x; \Gamma \\
\text{GrVar}^- & : \exists s, r \sqsupseteq s + 1 \Gamma, x : [A]_r \vdash A \Rightarrow - x; \Gamma, x : [A]_s
\end{align*}
\]

On the left, a variable \( x \) may be synthesised for the goal \( A \) if a linear assumption \( x : A \) is present in the input context. The input context without \( x \) is then returned as the output context, since \( x \) has been used. On the right, we can synthesise a variable \( x \) for \( A \) we have a graded assumption of \( x \) matching the type. However, the grading \( r \) must permit \( x \) to be used once here. Therefore, the premise states that there exists some grade \( s \) such that grade \( r \) approximates \( s + 1 \). The grade \( s \) represents the use of \( x \) in the rest of the synthesised term, and thus \( x : [A]_s \) is in the output context. For the natural numbers semiring, this constraint is satisfied by \( s = r - 1 \) whenever \( r \neq 0 \), e.g., if \( r = 3 \) then \( s = 2 \). For intervals, the role of approximation is more apparent: if \( r = [0...3] \) then this rule is satisfied by \( s = [0...2] \) where \( s + 1 = [0...2] + [1...1] = [1...3] \sqsubseteq [0...3] \). Thus, this premise constraint avoids the need for an additive inverse. In the implementation, the constraint is discharged via an SMT solver, where an unsatisfiable result terminates this branch of synthesis.

In typing, \( \lambda \)-abstraction binds linear variables to introduce linear functions. Synthesis from a linear function type therefore mirrors typing:

\[
\begin{align*}
\forall x : A \vdash B \Rightarrow t; \Delta, x \notin \Delta | \quad \Gamma \vdash A \Rightarrow - \lambda x. t; \Delta
\end{align*}
\]

Thus, \( \lambda x. t \) can be synthesised given that \( t \) can be synthesised from \( B \) in the context of \( \Gamma \) extended with a fresh linear assumption \( x : A \). To ensure that \( x \) is used linearly by \( t \) we must therefore check that it is not present in \( \Delta \).

The left-rule for linear function types then synthesises applications (as in \[19\]):

\[
\begin{align*}
\forall x_2 : B \vdash C \Rightarrow t_1; \Delta_1, x_2 \notin \Delta_1 | \quad \Delta_1 \vdash A \Rightarrow - t_2; \Delta_2 \\
\Gamma, x_1 : A \Rightarrow - B \vdash C \Rightarrow - [(x_1 t_2)/x_2] t_1; \Delta_2
\end{align*}
\]
The rule synthesises a term for type $C$ in a context that contains an assumption $x_1 : A \multimap B$. The first premise synthesises a term $t_1$ for $C$ under the context extended with a fresh linear assumption $x_2 : B$, i.e., assuming the result of $x_1$. This produces an output context $\Delta_1$ that must not contain $x_2$, i.e., $x_2$ is used by $t_1$. The remaining assumptions $\Delta_1$ provide the input context to synthesise $t_2$ of type $A$: the argument to the function $x_1$. In the conclusion, the application $x_1 t_2$ is substituted for $x_2$ inside $t_1$, and $\Delta_2$ is the output context.

Note that this rule synthesises the application of a function given by a linear assumption. What if we have a graded assumption of function type? Rather than duplicating every left rule for both linear and graded assumptions, we mirror the dereliction typing rule (converting a linear assumption to graded) as:

$$
\Gamma, x : [A]_s, y : A \vdash B \Rightarrow t; \Delta, x : [A]_{s'}, y \not\in |\Delta| \quad \exists s, r \geq s + 1 \quad \text{DER}^{-}
$$

Dereliction captures the ability to reuse a graded assumption being considered in a left rule. A fresh linear assumption $y$ is generated that represents the graded assumption’s use in a left rule, and must be used linearly in the subsequent synthesis of $t$. The output context of this premise then contains $x$ graded by $s’$, which reflects how $x$ was used in the synthesis of $t$, i.e. if $x$ was not used then $s’ = s$. The premise $\exists s, r \geq s + 1$ constrains the number of times dereliction can be applied so that it does not exceed $x$’s original grade $r$.

For a graded modal goal type $\Box r A$, we synthesise a promotion $[t]$ if we can synthesise the ‘unpromoted’ $t$ from $A$:

$$
\Gamma \vdash A \Rightarrow t; \Delta \quad \text{R}\Box^{-}
$$

Recall that typing of a promotion $[t]$ scales all the graded assumptions used to type $t$ by $r$. Therefore, to compute the output context we must “subtract” $r$-times the use of the variables in $t$. However, in the subtractive model $\Delta$ tells us what is left, rather than what is used. Thus we first compute the context subtraction of $\Gamma$ and $\Delta$ yielding the variables usage information about $t$:

**Definition 4 (Context subtraction).** For all $\Gamma_1, \Gamma_2$ where $\Gamma_2 \subseteq \Gamma_1$:

$$
\Gamma_1 - \Gamma_2 = \begin{cases} 
\Gamma_1 & \Gamma_2 = \emptyset \\
(\Gamma'_1, \Gamma''_1) - \Gamma'_2 & \Gamma_2 = \Gamma'_2, x : A \quad \Gamma_1 = \Gamma'_1, x : A, \Gamma''_1 \\
((\Gamma'_1, \Gamma''_1) - \Gamma'_2), x : [A]_q & \Gamma_2 = \Gamma'_2, x : [A]_s \quad \Gamma_1 = \Gamma'_1, x : [A]_r, \Gamma''_1 \\
\wedge \exists q, r \geq q + s \quad \forall q' \cdot r \geq q' + s & \implies q \equiv q'
\end{cases}
$$

As in graded variable synthesis, context subtraction existentially quantifies a variable $q$ to express the relationship between grades on the right being “subtracted” from those on the left. The last conjunct states $q$ is the greatest element (wrt. to the pre-order) satisfying this constraint, i.e., for all other $q' \in R$ satisfying the subtraction constraint then $q \equiv q'$ e.g., if $r = [2...3]$ and $s = [0...1]$.
then \(q = [2...2]\) instead of, say, \([0...1]\). This **maximality** condition is important for soundness (that synthesised programs are well-typed).

Thus for \(R \square^\top\), \(\Gamma - \Delta\) is multiplied by the goal type grade \(r\) to obtain how these variables are used in \(t\) after promotion. This is then subtracted from the original input context \(\Gamma\) giving an output context containing the left-over variables and grades. Context multiplication requires that \(\Gamma - \Delta\) contains only graded variables, preventing the incorrect use of linear variables from \(\Gamma\) in \(t\).

Synthesis of graded modality elimination, is handled by the \(L \square^-\) left rule:

\[
\frac {\Gamma, x_2 : [A]_r \vdash B \Rightarrow t; \Delta, x_2 : [A]_s \quad 0 \nleq s \quad L \square^-} {\Gamma, x_1 : \square_r A \vdash B \Rightarrow t \quad \text{let} \ [x_2] = x_1 \ \text{in} \ t ; \Delta}
\]

Given an input context comprising \(\Gamma\) and a linear assumption \(x_1\) of graded modal type, we can synthesise an unboxing of \(x_1\) if we can synthesise a term \(t\) under \(\Gamma\) extended with a graded assumption \(x_2 : [A]_r\). This returns an output context that must contain \(x_2\) graded by \(s\) with the constraint that \(s\) must approximate \(0\). This enforces that \(x_2\) has been used as much as stated by the grade \(r\).

The right and left rules for products, units, and sums, are then fairly straightforward following the subtractive resource model:

\[
\frac {\Gamma \vdash A \Rightarrow t_1; \Delta_1 \quad \Delta_1 \vdash B \Rightarrow t_2; \Delta_2 \quad \text{R} \gamma^-} {\Gamma \vdash A \otimes B \Rightarrow (t_1, t_2); \Delta}
\]

\[
\frac {\Gamma, x_1 : A, x_2 : B \vdash C \Rightarrow t_1; \Delta \quad x_1 \notin |\Delta| \quad x_2 \notin |\Delta| \quad \text{L} \gamma^-} {\Gamma, x_1 : A \otimes B \vdash C \Rightarrow \text{let} \ (x_1, x_2) = x_1 \ \text{in} \ t_1 ; \Delta}
\]

\[
\frac {\Gamma \vdash t; \Delta} {\Gamma, x : 1 \vdash C \Rightarrow t = x \ \text{in} \ t ; \Delta \quad \text{L} 1^-}
\]

\[
\frac {\Gamma \vdash A \Rightarrow t; \Delta} {\Gamma \vdash \text{inl} \ t ; \Delta \quad \text{R} \oplus_1^-}
\]

\[
\frac {\Gamma \vdash A \Rightarrow t; \Delta} {\Gamma \vdash \text{inr} \ t ; \Delta \quad \text{R} \oplus_2^-}
\]

\[
\frac {\Gamma, x_1 : A + B \vdash C \Rightarrow t_1; \Delta_1 \quad \Gamma, x_3 : B + C \vdash t_2; \Delta_2 \quad x_2 \notin |\Delta_1| \quad x_3 \notin |\Delta_2| \quad \text{L} \oplus^-} {\Gamma, x_1 : A + B \vdash C \Rightarrow \text{case} \ x_1 \ \text{of} \ \text{inl} \ x_2 \rightarrow t_1 \ | \ \text{inr} \ x_3 \rightarrow t_2 ; \Delta_1 \cap \Delta_2}
\]

The \(L \oplus^-\) rule syntheises the left and right branches of a case statement that may use resources differently. The output context therefore takes the **greatest lower bound** (\(\sqcap\)) of \(\Delta_1\) and \(\Delta_2\). We elide definition of context \(\sqcap\) as it has the same shape as \(\sqcup\) for contexts (Definition 2), just replacing \(\sqcup\) with \(\sqcap\) on grades.

As an example of \(\sqcap\), consider the semiring of intervals over natural numbers and two judgements that could be used as premises for the \((L \oplus^-)\) rule:

\[
\frac {\Gamma, y : [A']_3 \vdash t_1 ; x_2 : A + C \Rightarrow t_1; \Delta \quad \Gamma, y : [A']_3 \vdash t_2 ; x_3 : B + C \Rightarrow t_2 ; y : [A']_3 \quad \text{R} \oplus^-} {\Gamma, y : [A']_3 \vdash \text{case} \ x_1 \ \text{of} \ \text{inl} \ x_2 \rightarrow t_1 \ | \ \text{inr} \ x_3 \rightarrow t_2 ; \Delta_1 \cap \Delta_2}
\]

where \(t_1\) uses \(y\) such that there are 2-5 uses remaining and \(t_2\) uses \(y\) such that there are 3-4 uses left. To synthesise \(\text{case} \ x_1 \ \text{of} \ \text{inl} \ x_2 \rightarrow t_1 \ | \ \text{inr} \ x_3 \rightarrow t_2\) the output context must be pessimistic about what resources are left, thus we take
the greatest-lower bound yielding the interval [2..4] here: we know y can be used at least twice and at most 4 times in the rest of the synthesised program.

This completes subtractive synthesis. We conclude with a key result, that synthesised terms are well-typed at the type from which they were synthesised:

Lemma 2 (Subtractive synthesis soundness). For all \( \Gamma \) and \( A \) then:

\[
\Gamma \vdash A \Rightarrow t; \Delta \implies \Gamma - \Delta \vdash t : A
\]

i.e. \( t \) has type \( A \) under context \( \Gamma - \Delta \), that contains just those linear and graded variables with grades reflecting their use in \( t \). Appendix [\ref{appendix}] provides the proof.

3.2 Additive Resource Management

We now propose a dual additive resource management approach. Additive synthesis also uses the input-output context approach, but where output contexts describe exactly which assumptions were used to synthesise a term, rather than which assumptions are still available. Additive synthesis rules are read bottom-up, with \( \Gamma \vdash A \Rightarrow^+ t; \Delta \) meaning that from the type \( A \) we synthesise a term \( t \) using exactly the assumptions \( \Delta \) that originate from the input context \( \Gamma \).

We unpack the rules, starting with variables:

\[
\Gamma, x : A \vdash A \Rightarrow^+ x; x : A \ \text{LINVAR}^+ \quad \Gamma, x : [A]_r \vdash A \Rightarrow^+ x; x : [A]_1 \ \text{GRVAR}^+
\]

For a linear assumption, the output context contains just the variable that was synthesised. For a graded assumption \( x : [A]_r \), the output context contains the assumption graded by 1. To synthesise a variable from a graded assumption, we must check that the use is compatible with the grade. The subtractive approach handled this rule (GRVARخص) by a constraint \( \exists s. r \sqsubseteq s + 1 \). Here however, the point at which we check that a graded assumption has been used according to the grade takes place in the L\( \Box^+ \) rule, where graded assumptions are bound:

\[
\Gamma, x_2 : [A]_r \vdash B \Rightarrow^+ t; \Delta \quad \text{if } x_2 : [A]_s \in \Delta \text{ then } s \sqsubseteq r \text{ else } 0 \sqsubseteq r \quad \Gamma, x_1 : \Box_r A \vdash B \Rightarrow^+ \text{let } [x_2] = x_1 \in t; \quad (\Delta \setminus x_2), x_1 : \Box_r A \quad \text{L\( \Box^+ \)}
\]

Here, \( t \) is synthesised under a fresh graded assumption \( x_2 : [A]_r \). This produces an output context containing \( x_2 \) with some grade \( s \) that describes how \( x_2 \) is used in \( t \). An additional premise requires that the original grade \( r \) approximates either \( s \) if \( x_2 \) appears in \( \Delta \) or 0 if it does not, ensuring that \( x_2 \) has been used correctly. For the \( \mathbb{N} \)-semiring with equality as the ordering, this would ensure that a variable has been used exactly the number of times specified by the grade.

Right and left rules for \( \text{R}^-\) have a similar shape to the subtractive calculus:

\[
\Gamma, x : A \vdash B \Rightarrow^+ t; \Delta, x : A \quad \Gamma \vdash A \Leftarrow B \Rightarrow^+ \lambda x.t; \Delta \quad \text{R}^-\text{R}^+
\]

\[
\Gamma, x_2 : B \vdash C \Rightarrow^+ t_1; \Delta_1, x_2 : B \quad \Gamma \vdash A \Rightarrow^+ t_2; \Delta_2 \quad \Gamma, x_1 : A \Leftarrow B \vdash C \Rightarrow^+ [(x_1 t_2)/x_2] t_1; \quad (\Delta_1 + \Delta_2), x_1 : A \Leftarrow B \quad \text{L}^-\text{R}^+
\]
Synthesising an abstraction (R→^+) requires that x : A is in the output context of the premise, ensuring that linearity is preserved. Likewise for application (L→^+), the output context of the first premise must contain the linearly bound x_2 : B and the final output context must contain the assumption being used in the application x_1 : A → o B. This output context computes the context addition (Def. 1) of both output contexts of the premises Δ_1 + Δ_2. If Δ_1 describes how assumptions were used in t_1 and Δ_2 respectively for t_2, then the addition of these two contexts describes the usage of assumptions for the entire subprogram.

Recall, context addition ensures that a linear assumption may not appear in both Δ_1 and Δ_2, preventing us from synthesising terms that violate linearity.

As in the subtractive approach, we avoid duplicating left rules to match graded assumptions by giving a synthesising version of dereliction:

\[
\frac{Γ, x : [A]_s, y : A ⊢ B}{Γ; x : [A], y : A ⊢ B} \text{ DER}^+
\]

The fresh linear assumption y : A must appear in the output context of the premise, ensuring it is used. The final context therefore adds to Δ an assumption of x graded by 1, accounting for this use of x (temporarily renamed to y).

Synthesis of a promotion is considerably simpler in the additive approach. In subtractive resource management it was necessary to calculate how resources were used in the synthesis of t before then applying the scalar context multiplication by the grade r and subtracting this from the original input Γ. In additive resource management, however, we can simply apply the multiplication directly to the output context Δ to obtain how our assumptions are used in [t]:

\[
\frac{Γ ⊢ A \Rightarrow^+ t; Δ}{Γ; x : [A]_s \Rightarrow^+[t]; r * Δ} \text{ R}^+
\]

As in the subtractive approach, the right and left rules for products, units, and sums follow fairly straightforwardly from the resource scheme:

\[
\frac{Γ ⊢ A \Rightarrow^+ t_1; Δ, y : A \Rightarrow^+ t_2; Δ}{Γ; x : [A]_s \Rightarrow^+ [x/y]t; Δ + x : [A]_s} \text{ L}^+
\]

\[
\frac{Γ, x_1 : A, x_2 : B \Rightarrow^+ t; Δ, x_1 : A, x_2 : B}{Γ; x_3 : A \otimes B \Rightarrow^+ \text{ let } (x_1, x_2) = x_3 \text{ in } t_2; Δ, x_3 : A \otimes B} \text{ L}^+
\]

\[
\frac{Γ ⊢ A \Rightarrow^+ t; Δ, y : A \otimes B \Rightarrow^+ \text{ inl } t; Δ}{Γ; x : 1 \Rightarrow^+ \text{ let } () = x \text{ in } t; Δ, x : 1} \text{ L}^+
\]

\[
\frac{Γ, x_2 : A \Rightarrow^+ t_1; Δ, x_2 : A}{Γ; x_3 : B \Rightarrow^+ \text{ case } x_2 \text{ of inl } x_2 \rightarrow t_1; \text{ inr } x_3 \rightarrow t_2; (Δ \sqcup Δ_2), x_1 : A \oplus B} \text{ L}\oplus^+
\]

Rule (L⊕^+) takes the least-upper bound of the premise’s output contexts (Def. 2).
Lemma 3 (Additive synthesis soundness). For all $\Gamma$ and $A$:

$$\Gamma 
\vdash A \Rightarrow^+ \; t; \; \Delta \implies \Delta \vdash t : A$$

Appendix E gives the proof.

Additive pruning As seen above, the additive approach delays checking whether a variable is used according to its linearity/grade until it is bound. We hypothesise that this can lead additive synthesis to explore many ultimately ill-typed (or ill-resourced) paths for too long. Subsequently, we define a “pruning” variant of any additive rules with multiple sequenced premises. For $(R \otimes^+)$ this is:

$$\Gamma \vdash A \Rightarrow^+ t_1; \; \Delta_1 \quad \Gamma - \Delta_1 \vdash B \Rightarrow^+ t_2; \; \Delta_2 \quad R' \otimes^+$$

Instead of passing $\Gamma$ to both premises, $\Gamma$ is the input only for the first premise. This premise outputs context $\Delta_1$ that is subtracted from $\Gamma$ to give the input context of the second premise. This provides an opportunity to terminate the current branch of synthesis early if $\Gamma - \Delta_1$ does not contain the necessary resources to attempt the second premise. The $(L \Rightarrow^+)$ rule is similarly adjusted.

Lemma 4 (Additive pruning synthesis soundness). For all $\Gamma$ and $A$:

$$\Gamma \vdash A \Rightarrow^+ t; \; \Delta \implies \Delta \vdash t : A$$

Appendix E gives the proof.

3.3 Focusing

The two calculi provide a foundation for a synthesis algorithm. However, in their current form, both synthesis calculi are highly non-deterministic: for each rule there are multiple rules which may be applied to synthesise the premise(s).

We apply the idea of focusing to derive two focusing calculi which are equivalent to the former in expressivity, but with a reduced degree of non-determinism in the rules that may be applied. Focusing is a proof search technique based on the idea that some rules are invertible, i.e. whenever the premises of a rule are derivable, the conclusion is also derivable. Rules with this property can be applied eagerly in the synthesis of a term. When we arrive at a goal whose applicable rules are not invertible, we focus on either the goal type or a particular assumption by applying a chain of non-invertible rules until we reach a goal to which invertible rules can be applied. Appendix D.1 gives the focusing versions of the two calculi, which form the basis of our implementation. The proofs for the soundness of these focusing calculi can be found in Appendix E.
4 Evaluation

Prior to evaluation, we made the following hypotheses about the relative performance of the additive versus subtractive approaches:

1. Additive synthesis should make fewer calls to the solver, with lower complexity theorems (fewer quantifiers). Dually, subtractive synthesis makes more calls to the solver with higher complexity theorems (more quantifiers);
2. For complex problems, additive synthesis will explore more paths as it cannot tell whether a variable is not well-resourced until closing a binder; additive pruning and subtractive will explore fewer paths as they can fail sooner.
3. A corollary of the above two: simple examples will likely be faster in additive mode, but more complex examples will be faster in subtractive mode.

Methodology

We implemented our approach as a synthesis tool for Granule, integrated with its core tool. Granule features ML-style polymorphism (rank-0 quantification) but we do not address polymorphism here. Instead, programs are synthesised from type schemes treating universal type variables as logical atoms.

Constraints on resource usage are handled via Granule’s existing symbolic engine, which compiles constraints on grades (for various semirings) to the SMT-lib format for Z3 [28]. We use the LogicT monad for backtracking search [24].

To evaluate our synthesis tool we developed a suite of benchmarks comprising Granule type schemes for a variety of operations using linear and graded modal types. We divide our benchmarks into several classes of problem:

- Hilbert: the Hilbert-style axioms of intuitionistic logic (including SKI combinators), with appropriate \( \mathbb{N} \) and \( \mathbb{N} \)-intervals grades where needed (see, e.g., S combinator in Example 1 or coproduct elimination in Example 2).
- Comp: various translations of function composition into linear logic: multiplicative, call-by-value and call-by-name using ! [16], I/O using ! [26], and coKleisli composition over \( \mathbb{N} \) and arbitrary semirings: e.g. \( \forall r, s \in \mathbb{R} \):
  \[
  \text{comp-coK}_{\mathbb{R}} : \Box_r((\Box_s A \rightarrow B) \rightarrow (\Box_r B \rightarrow C)) \rightarrow \Box_{r+s} A \rightarrow C
  \]
- Dist: distributive laws of various graded modalities over functions, sums, and products [21], e.g., \( \forall r \in \mathbb{N} \), or \( \forall r \in \mathbb{R} \) in any semiring, or \( r = 0 \ldots \infty \):
  \[
  \text{pull}_{\Box} : (\Box_r A \oplus \Box_r B) \rightarrow \Box_r (A \oplus B) \quad \text{push}_{\Box} : \Box_r (A \rightarrow B) \rightarrow \Box_r A \rightarrow \Box_r B
  \]
- Vec: map operations on vectors of fixed size encoded as products, e.g.:
  \[
  \text{vmap}_5 : (\Box_4 A \rightarrow B) \rightarrow (((A \otimes A) \otimes A) \otimes A) \rightarrow (((B \otimes B) \otimes B) \otimes B)
  \]
- Misc: includes Example 3 (information-flow security) and functions which must share or split resources between graded modalities, e.g.:
  \[
  \text{share} : \Box_4 A \rightarrow \Box_6 A \rightarrow \Box_2 (((((A \otimes A) \otimes A) \otimes A) \rightarrow B) \rightarrow (B \otimes B)
  \]

Appendix C lists the type schemes for these synthesis problems (32 in total).

We found that Z3 is highly variable in its solving time, so timing measurements are computed as the mean of 20 trials. We used Z3 version 4.8.8 on a Linux laptop with an Intel i7-8665u @ 4.8 Ghz and 16 Gb of RAM.
Additive (pruning)

<table>
<thead>
<tr>
<th>Problem</th>
<th>Additive</th>
<th>Additive (pruning)</th>
<th>Subtractive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu T$ (ms)</td>
<td>N</td>
<td>$\mu T$ (ms)</td>
</tr>
<tr>
<td>Hilbert</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⊓Intro ✓</td>
<td>6.69 (0.05)</td>
<td>2 ✓</td>
<td>9.66 (0.23)</td>
</tr>
<tr>
<td>⊓Elim ✓</td>
<td>0.22 (0.01)</td>
<td>0 ✓</td>
<td>0.05 (0.00)</td>
</tr>
<tr>
<td>⊔Intro ✓</td>
<td>0.08 (0.00)</td>
<td>0 ✓</td>
<td>0.07 (0.00)</td>
</tr>
<tr>
<td>⊓Elim ✓</td>
<td>7.26 (0.30)</td>
<td>2 ✓</td>
<td>13.25 (0.58)</td>
</tr>
<tr>
<td>SKI ✓</td>
<td>8.12 (0.25)</td>
<td>2 ✓</td>
<td>24.98 (1.19)</td>
</tr>
<tr>
<td>Comp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⊓Intro ✓</td>
<td>28.31 (3.00)</td>
<td>5 ✓</td>
<td>41.86 (0.38)</td>
</tr>
<tr>
<td>cbn ✓</td>
<td>13.12 (0.84)</td>
<td>3 ✓</td>
<td>26.24 (0.27)</td>
</tr>
<tr>
<td>cbv ✓</td>
<td>19.68 (0.98)</td>
<td>5 ✓</td>
<td>34.15 (0.98)</td>
</tr>
<tr>
<td>ocoK_R ✓</td>
<td>33.37 (2.01)</td>
<td>2 ✓</td>
<td>27.37 (0.78)</td>
</tr>
<tr>
<td>ocoK_S ✓</td>
<td>27.59 (0.67)</td>
<td>2 ✓</td>
<td>21.62 (0.59)</td>
</tr>
<tr>
<td>mult ✓</td>
<td>0.29 (0.02)</td>
<td>0 ✓</td>
<td>0.12 (0.00)</td>
</tr>
<tr>
<td>Dist</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⊓-  ✓</td>
<td>12.96 (0.48)</td>
<td>2 ✓</td>
<td>32.28 (1.32)</td>
</tr>
<tr>
<td>⊓-N  ✓</td>
<td>24.83 (1.01)</td>
<td>2 ×</td>
<td>32.18 (0.80)</td>
</tr>
<tr>
<td>⊓-R  ✓</td>
<td>28.17 (1.01)</td>
<td>2 ×</td>
<td>29.72 (0.90)</td>
</tr>
<tr>
<td>⊓-l ✓</td>
<td>7.87 (0.23)</td>
<td>2 ✓</td>
<td>16.54 (0.43)</td>
</tr>
<tr>
<td>⊓-N  ✓</td>
<td>22.13 (0.70)</td>
<td>2 ×</td>
<td>30.30 (1.02)</td>
</tr>
<tr>
<td>⊓-R  ✓</td>
<td>22.18 (0.60)</td>
<td>2 ×</td>
<td>31.24 (1.46)</td>
</tr>
<tr>
<td>⊓-l ✓</td>
<td>6.53 (0.16)</td>
<td>2 ✓</td>
<td>10.01 (0.25)</td>
</tr>
<tr>
<td>⊓-N  ✓</td>
<td>29.16 (0.82)</td>
<td>2 ×</td>
<td>28.71 (0.67)</td>
</tr>
<tr>
<td>⊓-R  ✓</td>
<td>29.31 (1.81)</td>
<td>2 ×</td>
<td>27.44 (0.60)</td>
</tr>
<tr>
<td>Vec</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vec5 ✓</td>
<td>4.72 (0.07)</td>
<td>1 ✓</td>
<td>14.93 (0.21)</td>
</tr>
<tr>
<td>vec10 ✓</td>
<td>5.51 (0.36)</td>
<td>1 ✓</td>
<td>20.81 (0.77)</td>
</tr>
<tr>
<td>vec15 ✓</td>
<td>9.75 (0.25)</td>
<td>1 ✓</td>
<td>22.09 (0.24)</td>
</tr>
<tr>
<td>vec20 ✓</td>
<td>13.40 (0.46)</td>
<td>1 ✓</td>
<td>30.18 (0.20)</td>
</tr>
<tr>
<td>Misc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>split ⊓ ✓</td>
<td>3.79 (0.04)</td>
<td>1 ✓</td>
<td>5.10 (0.16)</td>
</tr>
<tr>
<td>split ⊔ ✓</td>
<td>14.07 (1.01)</td>
<td>3 ✓</td>
<td>46.27 (2.04)</td>
</tr>
<tr>
<td>share ✓</td>
<td>292.02 (11.37)</td>
<td>44 ✓</td>
<td>100.85 (2.44)</td>
</tr>
<tr>
<td>exam ✓</td>
<td>8.09 (0.46)</td>
<td>2 ✓</td>
<td>26.03 (1.21)</td>
</tr>
</tbody>
</table>

Table 1. Results. $\mu T$ in ms to 2 d.p. with standard sample error in brackets

Results and analysis For each synthesis problem, we recorded whether synthesis was successful or not (denoted ✓ or ×), the mean total synthesis time ($\mu T$), the mean total time spent by the SMT solver ($\mu SMT$), and the number of calls made to the SMT solver (N). Table 1 summarises the results with the fastest case for each benchmark highlighted. For all benchmarks that used the SMT solver, the solver accounted for 91.73% – 99.98% of synthesis time, so we report only the mean total synthesis time $\mu T$. We set a timeout of 120 seconds.

Additive versus subtractive As expected, the additive approach generally synthesises programs faster than the subtractive. Our first hypothesis (that the additive approach in general makes fewer calls to the SMT solver) holds for almost all benchmarks, with the subtractive approach often far exceeding the number made by the additive. This is explained by the difference in graded variable synthesis between approaches. In the additive, a constant grade 1 is given for graded assumptions in the output context, whereas in the subtractive, a fresh grade variable is created with a constraint on its usage which is checked immediately.
As the total synthesis time is almost entirely spent in the SMT solver (more than 90%), solving constraints is by far the most costly part of synthesis leading to the additive approach synthesising most examples in a shorter amount of time.

Graded variable synthesis in the subtractive case also results in several examples failing to synthesise. In some cases, e.g., the first three comp benchmarks, the subtractive approach times-out as synthesis diverges with constraints growing in size due to the maximality condition and absorbing behaviour of \([0\ldots\infty]\) interval. In the case of \(coK-R\) and \(coK-N\), the generated constraints have the form \(\forall r.\exists s.r \sqsubseteq s + 1\) which is not valid \(\forall r \in \mathbb{N}\) (e.g., when \(r = 0\)), which suggests that the subtractive approach does not work well for polymorphic grades. As further work, we are considering an alternate rule for synthesising promotion with constraints of the form \(\exists s.s = s' * r\), i.e., a multiplicative inverse constraint.

In more complex examples we see evidence to support our second hypothesis. The share problem requires a lot of graded variable synthesis which is problematic for the additive approach, for the reasons described in the second hypothesis. In contrast, the subtractive approach performs better, with \(\mu T = 193.3ms\) as opposed to additive’s 292.02ms. However, additive pruning outperforms both.

Additive pruning The pruning variant of additive synthesis (where subtraction takes place in the premises of multiplicative rules) had mixed results compared to the default. In simpler examples, the overhead of pruning (requiring SMT solving) outweighs the benefits obtained from reducing the space. However, in more complex examples which involve synthesising many graded variables (e.g. share), pruning is especially powerful, performing better than the subtractive approach. However, additive pruning failed to synthesise two examples which are polymorphic in their grade (\(\otimes-N\)) and in the semiring/graded-modality (\(\otimes-R\)).

Overall, the additive approach outperforms the subtractive and is successful at synthesising more examples, including ones polymorphic in grades and even the semiring itself. Given that the literature on linear logic theorem proving is typically subtractive, this is an interesting result. Going forward, a mixed approach between additive and additive pruning may be possible, selecting the algorithm, or even the rules, depending on the class of problem. Exploring this, and further optimisations and improvements, is further work.

5 Discussion

Further related work Before Hodas and Miller [19], the problem of resource non-determinism was first identified by Harland and Pym [18]. Their solution delays splitting of contexts at a multiplicative connective. They later explored the implementation details of this approach, proposing a solution where proof search is formulated in terms of constraints on propositions. The logic programming language Lygon [1] implements this approach.

Our approach to synthesis implements a backward style of proof search: starting from the goal, recursively search for solutions to subgoals. In contrast to this, forward reasoning approaches attempt to reach the goal by building subgoals
from previously proved subgoals until the overall goal is proved. Pfenning and Chaudhuri consider forward approaches to proof search in linear logic using the inverse method \cite{inverse}, where the issue of resource non-determinism that is typical to backward approaches is absent \cite{8,9}.

Non-idempotent intersection types systems have a similar core structure resembling the linear $\lambda$-calculus with quantitative aspects akin to grading \cite{6}. It therefore seems likely that the approaches of this paper could be applied in this setting and used, for example, as way to enhance or even improve existing work on the inhabitation problem for non-idempotent intersection types \cite{5}: a synthesised term gives a proof of inhabitation. This is left as further work, including formalising the connection between non-idempotent intersections and grading.

Next steps and Conclusions Our synthesis algorithms are now part of the Granule toolchain with IDE support, allowing programmers to insert a “hole” in a term and, after executing a keyboard shortcut, Granule tries to synthesise the type of the hole, pretty-printing generated code and inserting it at the cursor.

There are various extensions which we are actively pursuing, including synthesis for arbitrary user-defined indexed data types (GADTs), polymorphism, and synthesis of recursive functions. We plan to study various optimisations to the approaches considered here, as well as reducing the overhead of starting the SMT solver each time by instead running an “online” SMT solving procedure. We also plan to evaluate the approach on the extended linear logical benchmarks of Olarte et al. \cite{29}. Although our goal is to create a practical program synthesis tool for common programming tasks rather than a general purpose proof search tool, the approach here also has applications to automated theorem proving.

Acknowledgements Thanks to Benjamin Moon, Harley Eades III and the anonymous reviewers for their helpful comments. This work is supported by an EPSRC Doctoral Training Award and EPSRC grant EP/T013516/1 (Verifying Resource-like Data Use in Programs via Types).

References

1. Logic programming with linear logic, \url{http://www.cs.rmit.edu.au/lygon/}, accessed 19th June 2020


A Collected Rules of the Subtractive Calculus

\[
\begin{align*}
& \frac{\Gamma, x : A \vdash A \Rightarrow x; \Gamma}{\text{LinVar}^{-}} \quad \frac{\exists s, r \ni s + 1}{\text{GrVar}^{-}} \\
& \frac{\Gamma, x : [A] \vdash A \Rightarrow x; \Gamma, x : [A], s}{\text{DER}^{-}} \\
& \frac{\Gamma, x : [A], y : A \vdash B \Rightarrow t; \Delta, x : [A], y \notin [\Delta] \quad \exists s, r \ni s + 1}{\Gamma, x : [A], y : A \vdash B \Rightarrow [x/y]t; \Delta, x : [A], y} \\
& \frac{\Gamma, x : A \vdash B \Rightarrow t; \Delta \quad x \notin [\Delta]}{\Gamma \vdash A \Rightarrow B \Rightarrow \lambda x.t; \Delta} \quad \frac{\Gamma \vdash A \Rightarrow B \Rightarrow \lambda x.t; \Delta}{R\Rightarrow} \\
& \frac{\Gamma, x_2 : B \vdash C \Rightarrow t_1; \Delta_1 \quad x_2 \notin [\Delta_1]}{\Gamma, x_1 : A \Rightarrow B \vdash C \Rightarrow \text{let } x_2 = x_1 \text{ in } t_1; \Delta_2} \quad \frac{\Gamma, x_1 : A \Rightarrow B \vdash C \Rightarrow \text{let } x_2 = x_1 \text{ in } t_1; \Delta_2}{L\Rightarrow} \\
& \frac{\Gamma \vdash A \Rightarrow t_1; \Delta_1 \quad \Delta_1 \vdash A \Rightarrow t_2; \Delta_2}{\Gamma \vdash A \Rightarrow B \Rightarrow t_1; t_2; \Delta_2} \quad \frac{\Gamma \vdash A \Rightarrow B \Rightarrow t_1; t_2; \Delta_2}{R\Rightarrow} \\
& \frac{\Gamma, x_1 : A, x_2 : B \vdash C \Rightarrow t_2; \Delta \quad x_1 \notin [\Delta] \quad x_2 \notin [\Delta]}{\Gamma, x_1 : A \otimes B \vdash C \Rightarrow \text{let } (x_1, x_2) = x_3 \text{ in } t_2; \Delta} \quad \frac{\Gamma, x_1 : A \otimes B \vdash C \Rightarrow \text{let } (x_1, x_2) = x_3 \text{ in } t_2; \Delta}{L\Rightarrow} \\
& \frac{\Gamma \vdash A \Rightarrow t; \Delta}{\Gamma \vdash A \otimes B \Rightarrow \text{inl } t; \Delta} \quad \frac{\Gamma \vdash B \Rightarrow t; \Delta}{\Gamma \vdash A \otimes B \Rightarrow \text{inr } t; \Delta} \quad \frac{\Gamma \vdash A \Rightarrow \text{inl } t; \Delta}{R\Rightarrow} \\
& \frac{\Gamma, x_2 : A \vdash C \Rightarrow t_1; \Delta_1 \quad \Gamma, x_3 : B \vdash C \Rightarrow t_2; \Delta_2 \quad x_2 \notin [\Delta_1] \quad x_3 \notin [\Delta_2]}{\Gamma, x_1 : A \otimes B \vdash C \Rightarrow \text{case } x_1 \text{ of inl } x_2 \rightarrow t_1 | \text{inr } x_3 \rightarrow t_2; \Delta_1 \cap \Delta_2} \quad \frac{\Gamma, x_1 : A \otimes B \vdash C \Rightarrow \text{case } x_1 \text{ of inl } x_2 \rightarrow t_1 | \text{inr } x_3 \rightarrow t_2; \Delta_1 \cap \Delta_2}{L\Rightarrow} \\
& \frac{\Gamma, x : A \vdash C \Rightarrow t; \Delta}{\Gamma \vdash \text{let } x \equiv \text{in } t; \Delta} \quad \frac{\Gamma \vdash C \Rightarrow \text{let } x \equiv \text{in } t; \Delta}{R1^{-}} \\
& \frac{\Gamma \vdash 1 \Rightarrow (); \Gamma}{R1^{-}} \\
& \frac{\Gamma \vdash 1 \Rightarrow (); \Gamma, x : 1 \vdash C \Rightarrow \text{let } x \equiv \text{in } t; \Delta}{L1^{-}}
\end{align*}
\]
B Collected Rules of the Additive Calculus

\[
\begin{align*}
\Gamma, x : A \vdash A \Rightarrow x; x : A &\quad \text{LinVAR}^+ \\
\Gamma, x : [A]_r \vdash A \Rightarrow x; x : [A]_l &\quad \text{GrVAR}^+ \\
\begin{array}{c}
\Gamma, x : [A]_r, y : A \vdash B \Rightarrow t; \Delta, y : A \\
\Gamma, x : [A]_l \vdash B \Rightarrow [x/y]t; \Delta + x : [A]_l
\end{array} &\quad \text{DER}^+ \\
\Gamma, x : A \vdash B \Rightarrow t; \Delta, x : A &\quad \text{R}^+ \\
\Gamma \vdash A \rightarrow B \Rightarrow \lambda x.t; \Delta &\quad \text{R}^- \\
\Gamma, x_2 : B \vdash C \Rightarrow t_1; \Delta_1, x_2 : B &\quad \Gamma \vdash A \Rightarrow t_2; \Delta_2 \\
\Gamma, x_1 : A \rightarrow B \vdash C \Rightarrow [(x_1 t_2)/x_2] t_1; (\Delta_1 + \Delta_2), x_1 : A \rightarrow B &\quad \text{L}^- \\
\Gamma \vdash A \Rightarrow t; \Delta &\quad \Gamma \vdash \Box A \Rightarrow [t]; r * \Delta &\quad \text{R}^+ \\
\Gamma, x_2 : [A]_r \vdash B \Rightarrow t; \Delta &\quad \text{if } x_2 : [A]_r \in \Delta \text{ then } s \sqsubseteq r \text{ else } 0 \sqsubseteq r &\quad \text{L}^+ \\
\Gamma, x_1 : \Box A \vdash B \Rightarrow t &\quad \text{let } [x_2] = x_1 \text{ in } t; (\Delta \setminus x_2), x_1 : \Box A &\quad \text{L}^+ \\
\Gamma \vdash A \Rightarrow t_1; \Delta_1, \Gamma \vdash B \Rightarrow t_2; \Delta_2 &\quad \Gamma \vdash A \otimes B \Rightarrow (t_1, t_2); \Delta_1 + \Delta_2 &\quad \text{R}^\otimes \\
\Gamma, x_1 : A, x_2 : B \vdash C \Rightarrow t_2; \Delta, x_1 : A, x_2 : B &\quad \Gamma, x_3 : A \otimes B \vdash C \Rightarrow [(x_1 t_2)/x_3] t_1; \Delta, x_3 : A \otimes B &\quad \text{L}^\otimes \\
\Gamma \vdash A \Rightarrow t_1; \Delta &\quad \Gamma \vdash A \otimes B \Rightarrow t &\quad \text{let } [x] \in t_1 &\quad \text{R}^\otimes_2 \\
\Gamma, x_2 : A \vdash C \Rightarrow t_1; \Delta_1, x_2 : A &\quad \Gamma, x_3 : B \vdash C \Rightarrow t_2; \Delta_2, x_3 : B &\quad \Gamma, x_1 : A \oplus B \vdash C \Rightarrow \text{case } x_1 \text{ of } \text{inl } t_2 \rightarrow t_1 \text{ inr } x_3 \rightarrow t_2; (\Delta_1 \cup \Delta_2), x_1 : A \oplus B &\quad \text{L}^{\oplus} \\
\Gamma \vdash 1 \Rightarrow (); \emptyset &\quad \Gamma, x : 1 \vdash C \Rightarrow \text{let } () = x \text{ in } t; \Delta, x : 1 &\quad \text{L}^1 \\
\end{align*}
\]

B.1 Alternative pruning rules for pair introduction and application

\[
\begin{align*}
\Gamma, x_2 : B \vdash C \Rightarrow t_1; \Delta_1, x_2 : B &\quad \Gamma \vdash \Delta_1 \Rightarrow t_2; \Delta_2 \\
\Gamma, x_1 : A \rightarrow B \vdash C \Rightarrow [(x_1 t_2)/x_2] t_1; (\Delta_1 + \Delta_2), x_1 : A \rightarrow B &\quad \text{L}^- \Rightarrow \\
\Gamma \vdash A \Rightarrow t_1; \Delta_1 &\quad \Gamma \vdash \Delta_1 \Rightarrow B \Rightarrow t_2; \Delta_2 &\quad \Gamma \vdash A \otimes B \Rightarrow \langle t_1, t_2 \rangle; \Delta_1 + \Delta_2 &\quad \text{R}^\otimes \\
\end{align*}
\]

20
C  List of benchmark synthesis problems

In the following $\Box A$ is shorthand for $\Box_{[0,\infty]} A$ (graded modality with indices drawn from intervals over $\mathbb{N} \cup \infty$).

| Hilbert |  
| --- | --- |
| ⊤Intro | $\top_i : \forall a, b. \ a \to b \to (a \otimes b)$ |
| ⊤Elim | $\top_{e1} : \forall a, b. \ (a \otimes \Box_0 b) \to a$ |
| ⊤Intro | $\top_{i1} : \forall a, b. \ a \to a \oplus b$ |
| ⊤Elim | $\top_{e2} : \forall a, b. \ (\Box_0 a \otimes b) \to b$ |
| SKI | $\Box : \forall a, b, c. \ (a \to c) \to \Box (b \to c) \to (a \otimes b) \to c$ |

| Comp |  
| --- | --- |
| mult | $\circ : \forall a, b, c. \ (a \to b) \to (b \to c) \to (a \to c)$ |
| 0/1 | $\circ_{0/1} : \forall a, b, c. \ (\Box a \to \Box b) \to (\Box (\Box b \to \Box c) \to (\Box (\Box a \to c)$ |
| CBN | $\circ_{CBN} : \forall a, b, c. \ (\Box a \to b) \to (\Box (\Box b \to c) \to (\Box a \to c$ |
| CBV | $\circ_{CBV} : \forall a, b, c. \ (\Box a \to \Box b) \to (\Box (\Box b \to c) \to (\Box a \to c$ |
| coK-R | $\forall R, r, s \in R, a, b, c. \ (\Box_r a \to b) \to (\Box_r b \to c) \to (\Box_{r+s} a \to c$ |
| coK-N | $\forall N : \forall R, s \in N, a, b, c. \ (\Box_r a \to b) \to (\Box_r b \to c) \to (\Box_{r+s} a \to c$ |

| Dist |  
| --- | --- |
| ⊤-N | pull_0 : $\forall r : N, a, b. \ (\Box a \otimes \Box b) \to \Box (a \otimes b)$ |
| ⊤-1 | pull_1 : $\forall a, b. \ (\Box a \otimes \Box b) \to \Box (a \otimes b)$ |
| ⊥-R | pull_0 : $\forall R, r \in R, a, b. \ (\Box_r a \otimes \Box_r b) \to \Box_r (a \otimes b)$ |
| ⊤-N | pull_0 : $\forall r : N, a, b. \ (\Box a \otimes \Box b) \to \Box_r (a \otimes b)$ |
| ⊤-1 | pull_0 : $\forall r : N, a, b. \ (\Box a \otimes \Box b) \to \Box_r (a \otimes b)$ |
| ⊥-R | push_0 : $\forall r : R, a, b. \ (\Box_r a \to b) \to \Box_r a \to \Box_r b$ |
| ⊤-N | push_0 : $\forall r : N, a, b. \ (\Box_r a \to b) \to \Box_r a \to \Box_r b$ |
| Vec |  
| vec10 | $\forall a, b. \ as above but for 10-tuples$ |
| vec15 | $\forall a, b. \ as above but for 15-tuples$ |
| vec20 | $\forall a, b. \ as above but for 20-tuples$ |

| Misc |  
| --- | --- |
| split | $\forall a, b, c. \ (\Box_r a \otimes \Box_s b) \to (a \otimes c) \to (a \otimes (\Box_r a \otimes \Box_s b) \to (c \otimes (\Box_r a \otimes \Box_s b))$ |
| split | $\forall a, b, c. \ (\Box_r a \otimes \Box_s b) \to (a \otimes c) \to (a \otimes (\Box_r a \otimes \Box_s b) \to (c \otimes (\Box_r a \otimes \Box_s b))$ |
| share | $\forall a, b, c. \ (\Box_r a \otimes \Box_s b) \to (a \otimes c) \to (a \otimes (\Box_r a \otimes \Box_s b) \to (c \otimes (\Box_r a \otimes \Box_s b))$ |

| Exm. |  
| --- | --- |
| noLeak | $\forall a, b. \ (\Box_{a+b} \otimes \Box_{a+b}) \to (\Box_{a+b} (a \otimes 1) \to b) \to b$ |
## D Focusing Forms of the Synthesis Calculi

### D.1 Subtractive Resource Management

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premises</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RightAsync</strong></td>
<td>(\Gamma; \omega, x : A \vdash B \Rightarrow t; \Delta \quad x \not\in</td>
<td>\Delta</td>
</tr>
<tr>
<td>(\Gamma; \omega \vdash A \Rightarrow \lambda x.t; \Delta)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LeftAsync</strong></td>
<td>(\Gamma; \omega, x_1 : A, x_2 : B \vdash C \Rightarrow t_2; \Delta \quad x_1 \not\in</td>
<td>\Delta</td>
</tr>
<tr>
<td>(\Gamma; \omega, x_1 : A \otimes B \vdash C \Rightarrow t_2; \Delta_2 \quad x_2 \not\in</td>
<td>\Delta</td>
<td>)</td>
</tr>
<tr>
<td>(\Gamma; x_2 : A \vdash C \Rightarrow \Gamma_1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\Gamma; \omega, x_1 : A + B \vdash C \Rightarrow \Gamma_1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\Gamma; \omega, x_1 : \square A \vdash C \Rightarrow t; \Delta, \Gamma_1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Focus</strong></td>
<td>(\Gamma; x : A \vdash C \Downarrow \Rightarrow t; \Delta \quad C \text{ not atomic} )</td>
<td>(\text{FOCUS}^{-})</td>
</tr>
<tr>
<td>(\Gamma; x : A \vdash C \Downarrow \Rightarrow t; \Delta)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RightSync</strong></td>
<td>(\Gamma; \emptyset \vdash A \Downarrow \Rightarrow t_1; \Delta)</td>
<td>(R_{\Downarrow}^{-})</td>
</tr>
<tr>
<td>(\Gamma; \emptyset \vdash A \otimes B \Downarrow \Rightarrow (t_1, t_2); \Delta)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\Gamma; \emptyset \vdash A \Downarrow \Rightarrow t; \Delta)</td>
<td>(L_{\Downarrow}^{-})</td>
<td></td>
</tr>
<tr>
<td>(\Gamma; \emptyset \vdash \square A \Downarrow \Rightarrow \Gamma; \emptyset \vdash C \Downarrow \Rightarrow t; \Delta)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LeftSync</strong></td>
<td>(\Gamma; x_2 : B \Downarrow \Rightarrow t_1; \Delta_1 \quad x_2 \not\in</td>
<td>\Delta</td>
</tr>
<tr>
<td>(\Gamma; x_1 : A \Rightarrow B \Downarrow \Rightarrow \Gamma; \emptyset \vdash C \Downarrow \Rightarrow t; \Delta)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\exists s, r \sqsupseteq s + 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LinVar</strong></td>
<td>(\Gamma; x : [A]<em>{r} \Downarrow \Rightarrow A \Rightarrow x; \Gamma; x : [A]</em>{s})</td>
<td>(\text{GRVar}^{-})</td>
</tr>
<tr>
<td><strong>GrVar</strong></td>
<td>(\Gamma; x : A \Downarrow \Rightarrow C \Downarrow \Rightarrow t; \Delta)</td>
<td></td>
</tr>
</tbody>
</table>

22
D.2 Additive Resource Management

**RightAsync**

\[\Gamma; \Omega, x : A \vdash B \triangleright \Rightarrow t; \Delta, x : A \vdash C \Rightarrow \Gamma; \Omega; x : \Delta \vdash \Delta, x : \Delta \quad \text{not right async} \quad \text{R}_\triangleright^+\]

\[\Gamma; \Omega, x : A, x_2 : B \vdash C \Rightarrow t_2; \Delta, x_1 : A, x_2 : B \quad \text{L}_\triangleright^+\]

**LeftAsync**

\[\Gamma; \Omega, x_1 : A \otimes B \vdash C \Rightarrow t_2; \Delta, x_1 : A, x_2 : B \quad \text{R}_\otimes^+\]

\[\Gamma; \Omega, x_2 : A \vdash \Gamma; \Omega, x_3 : B \vdash C \Rightarrow t_2; \Delta, x_1 : A \otimes B \quad \text{L}_\otimes^+\]

\[\Gamma; \Omega, x_1 : A \otimes B \vdash \Gamma; \Omega, x_2 : \text{let } (x_1, x_2) = x_3 \text{ in } t_2; \Delta, x_3 : A \otimes B \quad \text{L}_\otimes^+\]

\[\Gamma; \Omega, x_2 : [A]_r \vdash \Gamma; \Omega, x_1 : \text{let } x_2 = x_1 \text{ in } t_2; \Delta, x_1 : [A]_r \quad \text{L}_1^+\]

**Focus**

\[\Gamma; \emptyset \vdash C \triangleright \Rightarrow t; \Delta \quad \text{C not atomic} \quad \text{FOCUSR}^+\]

\[\Gamma; \emptyset \vdash C \triangleright \Rightarrow t; \Delta \quad \text{FOCUSR}^+\]

\[\Gamma; x : A \triangleright \triangleright \Omega \vdash C \Rightarrow \Gamma; \emptyset \vdash C \triangleright \Rightarrow t; \Delta \quad \text{FOCUSL}^+\]

**RightSync**

\[\Gamma; \emptyset \vdash A \triangleright \Rightarrow t_1; \Delta_1 \quad \text{R}_\triangleright^+\]

\[\Gamma; \emptyset \vdash A \otimes B \triangleright \Rightarrow (t_1, t_2); \Delta_1 + \Delta_2 \quad \text{R}_{\otimes}^+\]

\[\Gamma; \emptyset \vdash B \triangleright \Rightarrow t_2; \Delta \quad \text{R}_\triangleright^+\]

**LeftSync**

\[\Gamma; x_2 : B \vdash C \Rightarrow t_1; \Delta_1, x_2 : B \quad \text{L}_\otimes^+\]

\[\Gamma; x_1 : A \rightarrow B \vdash C \Rightarrow [(x_1 t_2); x_2]; \Delta_1 + \Delta_2, x_1 : A \rightarrow B \quad \text{L}_\otimes^+\]

\[\Gamma; x : A \vdash A \Rightarrow x_1 : x_2 : A \quad \text{GRVAR}^+\]

\[\Gamma; x : [A]_r \vdash A \Rightarrow x_1 : x_2 : [A]_1 \quad \text{GRVAR}^+\]

\[\Gamma; \emptyset \vdash A \triangleright \Rightarrow t; \Delta \quad \text{A not atomic and not left sync} \quad \text{L}_{\triangleright}^+\]
Alternative Pruning Rules for Pair Introduction and Application

\[
\frac{\Gamma; x_2 : B \vdash C \Rightarrow t_1; \Delta_1, x_2 : B \quad \Gamma - \Delta_1; \emptyset \vdash A \Rightarrow^+ t_2; \Delta_2}{\Gamma; x_1 : A \Rightarrow B \vdash C \Rightarrow^+ t_1; (x_1 \cdot t_2)/x_2; (\Delta_1 + \Delta_2), x_1 : A \Rightarrow B}
\]

\[
\frac{\Gamma; x_1 : A \Rightarrow B \vdash C \Rightarrow^+ t_1; \Delta_1 \quad \Gamma - \Delta_1; \emptyset \vdash B \Rightarrow^+ t_2; \Delta_2}{\Gamma; x_1 : A \Rightarrow B \vdash C \Rightarrow^+ \langle t_1, t_2 \rangle; (\Delta_1 + \Delta_2)}
\]

\[
\frac{\Gamma; \emptyset \vdash A \Rightarrow^+ t_1; \Delta_1 \quad \Gamma - \Delta_1; \emptyset \vdash B \Rightarrow^+ t_2; \Delta_2}{\Gamma; \emptyset \vdash A \otimes B \Rightarrow^+ \langle t_1, t_2 \rangle; (\Delta_1 + \Delta_2)}
\]

E Soundness proofs

This appendix gives the proofs of Lemma 2 and Lemma 3, along with soundness results for the variant systems: additive pruning and subtractive division.

We first state and prove some intermediate results about context manipulations which are needed for the main lemmas.

Definition 5 (Context approximation). For contexts $\Gamma_1, \Gamma_2$ then:

\[
\begin{array}{c}
\emptyset \sqsubseteq \emptyset \\
\Gamma_1 \sqsubseteq \Gamma_2 \\
\Gamma_1, x : A \sqsubseteq \Gamma_2, x : A \\
\Gamma_1 \sqsubseteq \Gamma_2 \\
r \subseteq s \\
\Gamma_1, x : [A]_r \sqsubseteq \Gamma_2, x : [A]_s \\
\Gamma_1 \sqsubseteq \Gamma_2 \\
\emptyset \subseteq s \\
\Gamma_1 \sqsubseteq \Gamma_2, x : [A]_s
\end{array}
\]

This is actioned in type checking by iterative application of APPROX.

Lemma 5 ($\Gamma + (\Gamma' - \Gamma'') \sqsubseteq (\Gamma + \Gamma') - \Gamma''$).

Proof. Induction over the structure of both $\Gamma'$ and $\Gamma''$. The possible forms of $\Gamma'$ and $\Gamma''$ are considered in turn:

1. $\Gamma' = \emptyset$ and $\Gamma'' = \emptyset$

   We have:

   \[
   (\Gamma + \emptyset) - \emptyset = \Gamma + (\emptyset - \emptyset)
   \]

   From definitions 1 and 3, we know that on the left hand side:

   \[
   (\Gamma + \emptyset) - \emptyset = \Gamma + \emptyset
   = \Gamma
   \]

   and on the right-hand side:

   \[
   \Gamma + (\emptyset - \emptyset) = \Gamma + \emptyset
   = \Gamma
   \]

   making both the left and right hand sides equivalent:

   \[
   \Gamma = \Gamma
   \]
2. $\Gamma' = \Gamma', x : A$ and $\Gamma'' = \emptyset$

We have

$$(\Gamma + \Gamma', x : A) - \emptyset = \Gamma + (\Gamma, x : A - \emptyset)$$

From definitions 1 and 4, we know that on the left hand side we have:

$$(\Gamma + \Gamma', x : A) - \emptyset = (\Gamma, \Gamma'), x : A - \emptyset$$

and on the right hand side:

$$\Gamma + (\Gamma, x : A - \emptyset) = \Gamma + \Gamma', x : A$$

$$= (\Gamma, \Gamma'), x : A$$

making both the left and right hand sides equal:

$$(\Gamma, \Gamma'), x : A = (\Gamma, \Gamma'), x : A$$

3. $\Gamma' = \Gamma', x : A$ and $\Gamma'' = \Gamma'', x : A$

We have

$$(\Gamma + \Gamma', x : A) - \Gamma'' = \Gamma + (\Gamma', x : A - \Gamma'')$$

From definitions 1 and 4, we know that on the left hand side we have:

$$(\Gamma + \Gamma', x : A) - \Gamma'' = (\Gamma, \Gamma), x : A - \Gamma'', x : A$$

and on the right hand side:

$$\Gamma + (\Gamma', x : A - \Gamma'') = \Gamma + (\Gamma', x : A) - \Gamma''$$

$$= (\Gamma, \Gamma') - \Gamma''$$

making both the left and right hand sides equivalent:

$$\Gamma, \Gamma' - \Gamma'' = \Gamma, \Gamma' - \Gamma''$$

4. $\Gamma' = \Gamma', x : [A]_r$ and $\Gamma'' = \emptyset$

We have

$$(\Gamma + \Gamma', x : [A]_r) - \emptyset = \Gamma + (x : [A]_r - \emptyset)$$

From definitions 1 and 4, we know that on the left hand side we have:

$$(\Gamma + \Gamma', x : [A]_r) - \emptyset = (\Gamma + \Gamma', x : [A]_r)$$

and on the right hand side:

$$\Gamma + (\Gamma', x : [A]_r - \emptyset) = \Gamma + (\Gamma', x : [A]_r)$$

$$= (\Gamma, \Gamma'), x : [A]_r$$

making both the left and right hand sides equivalent:

$$(\Gamma, \Gamma'), x : [A]_r = (\Gamma, \Gamma'), x : [A]_r$$
5. $\Gamma' = \Gamma', x : [A]_r$ and $\Gamma'' = \Gamma'', x : [A]_s$

Thus we have (for the LHS of the inequality term):

$$\Gamma + (\Gamma', x : [A]_r - \Gamma'', x : [A]_s)$$

which by context subtraction yields:

$$\Gamma + (\Gamma', x : [A]_r - \Gamma'', x : [A]_s) = \Gamma + (\Gamma' - \Gamma''), x : [A]_{q'}$$

where:

$$\exists q'.r \supseteq q' + s \quad \forall \hat{q}'.r \supseteq \hat{q}' + s \implies q' \supseteq \hat{q}' \quad (2)$$

And for the LHS of the inequality, from definitions [1] and [4] we have:

$$(\Gamma + \Gamma', x : [A]_r) - \Gamma'', x : [A]_s = (\Gamma + \Gamma'), x : [A]_r - \Gamma'', x : [A]_s$$

$$= ((\Gamma + \Gamma') - \Gamma''), x : [A]_r - x : [A]_s$$

$$= ((\Gamma + \Gamma') - \Gamma''), x : [A]_{q'}$$

where:

$$\exists q.r \supseteq q + s \quad \forall \hat{q}.r \supseteq \hat{q} + s \implies q \supseteq \hat{q} \quad (1)$$

Applying $\exists q.r \supseteq q + s$ to maximality (2) (at $\hat{q}' = q$) then yields that $q \subseteq q'$.

Therefore, applying induction, we derive:

$$\frac{((\Gamma + \Gamma') - \Gamma'') \subseteq ((\Gamma + \Gamma') - \Gamma''), x : [A]_{q'} \subseteq ((\Gamma + \Gamma') - \Gamma''), x : [A]_{q'}}$$

satisfying the lemma statement.

**Lemma 6** ($(\Gamma - \Gamma') + \Gamma' \subseteq \Gamma$).

**Proof.** The proof follows by induction over the structure of $\Gamma'$. The possible forms of $\Gamma'$ are considered in turn:

1. $\Gamma' = \emptyset$

   We have:

   $$(\Gamma - \emptyset) + \emptyset = \Gamma$$

   From definition [4] we know that:

   $$\Gamma - \emptyset = \Gamma$$

   and from definition [1] we know:

   $$\Gamma + \emptyset = \Gamma$$

   giving us:

   $$\Gamma = \Gamma$$

   26
2. \( \Gamma' = \Gamma'', x : A \)

and let \( \Gamma = \Gamma', x : A \).

\[(\Gamma', x : A - \Gamma'', x : A) + \Gamma'', x : A = A\]

From definition \( \Gamma' = \Gamma'' \), \( x : A \) we know that:

\[(\Gamma', x : A - \Gamma'', x : A) + \Gamma'', x : A = ((\Gamma' - \Gamma'') + \Gamma''), x : A \]

induction = \( \Gamma', x : A \)

\[= \Gamma \]

thus satisfying the lemma statement by equality.

3. \( \Gamma' = \Gamma'', x : [A]_r \)

and let \( \Gamma = \Gamma', x : [A]_s \).

We have:

\[(\Gamma', x : [A]_s - \Gamma'', x : [A]_r) + \Gamma'', x : [A]_r \]

From definition \( \Gamma' = \Gamma'' \), \( x : [A]_r \) we know that:

\[(\Gamma', x : [A]_s - \Gamma'', x : [A]_r) + \Gamma'', x : [A]_r = (\Gamma' - \Gamma''), x : [A]_q + \Gamma'', x : [A]_r \]

\[= ((\Gamma' - \Gamma'') + \Gamma''), x : [A]_q + r \]

where \( s \supseteq q + r \) and \( \forall q'.s \supseteq q' + r \implies q \supseteq q' \).

Then by induction we derive the ordering:

\[ ((\Gamma' - \Gamma'') + \Gamma''), x : [A]_q + r \subseteq \Gamma', x : [A]_s \]

which satisfies the lemma statement.

**Lemma 7 (Context negation).** For all contexts \( \Gamma \):

\[ \emptyset \subseteq \Gamma - \Gamma \]

**Proof.** By induction on the structure of \( \Gamma \):

- \( \Gamma = \emptyset \) Trivial.
- \( \Gamma = \Gamma', x : A \) then \( (\Gamma', x : A) - (\Gamma', x : A) = \Gamma' - \Gamma' \) so proceed by induction.
- \( \Gamma = \Gamma', x : [A]_r \) then \( \exists q, (\Gamma', x : [A]_r) - (\Gamma', x : [A]_r) = (\Gamma' - \Gamma'), x : [A]_q \)

such that \( r \supseteq q + r \) and \( \forall q'.r \supseteq q' + r \implies q \supseteq q' \).

Instantiating maximality with \( q' = 0 \) and reflexivity then we have \( 0 \subseteq q \).

From this, and the inductive hypothesis, we can construct:

\[ \emptyset \subseteq (\Gamma - \Gamma') \quad 0 \subseteq q \]

\[ \emptyset \subseteq (\Gamma - \Gamma'), x : [A]_q \]
Lemma 8. For all contexts $\Gamma_1$, $\Gamma_2$, where $[\Gamma_2]$ (i.e., $\Gamma_2$ is all graded) then:

$$\Gamma_2 \subseteq \Gamma_1 - (\Gamma_1 - \Gamma_2)$$

Proof. By induction on the structure of $\Gamma_2$.

- $\Gamma_2 = \emptyset$
  Then $\Gamma_1 - (\Gamma_1 - \emptyset) = \Gamma_1 - \Gamma_1$.
  By Lemma 7, then $\emptyset \subseteq (\Gamma_1 - \Gamma_1)$ satisfying this case.

- $\Gamma_2 = \Gamma'_2, x : [A]_s$
  By the premises $\Gamma_1 \subseteq \Gamma_2$ then we can assume $x \in \Gamma_1$ and thus (by context rearrangement) $\Gamma'_1, x : [A]_r$.
  Thus we consider $(\Gamma'_1, x : [A]_r) - ((\Gamma'_1, x : [A]_r) - (\Gamma'_2, x : [A]_s))$.

$$= (\Gamma'_1, x : [A]_r) - ((\Gamma'_1 - \Gamma'_2), x : [A]_q)$$

where (1) $\exists q. r \supseteq q + s$ with (2) $(\forall q.r \supseteq q + s \implies q \supseteq q')$
and (3) $\exists q'. r \supseteq q' + q$ with (4) $(\forall q'. r \supseteq q' + s \implies q' \supseteq q')$.

Apply (1) to (4) by letting $q' = s$ and by commutativity of $+$ then we get that $q' \supseteq s$.
By induction we have that

$$\Gamma'_1 \subseteq \Gamma'_1 - (\Gamma'_1 - \Gamma'_2)$$

(ih)

Thus we get that:

$$\Gamma'_1, x : [A]_s \subseteq (\Gamma'_1 - (\Gamma'_1 - \Gamma'_2)), x : [A]_q$$

- $\Gamma_2 = \Gamma'_2, x : A$ Trivial as it violates the grading condition of the premise.

Lemma 2 (Subtractive synthesis soundness). For all $\Gamma$ and $A$ then:

$$\Gamma \vdash A \Rightarrow t; \Delta \implies \Gamma - \Delta \vdash t : A$$

i.e. $t$ has type $A$ under context $\Gamma - \Delta$, that contains just those linear and graded variables with grades reflecting their use in $t$. Appendix E provides the proof.

Proof. Structural induction over the synthesis rules. Each of the possible synthesis rules are considered in turn.

1. Case $\text{LinVar}^-$
   In the case of linear variable synthesis, we have the derivation:

$$\Gamma, x : A \vdash A \Rightarrow x; \Gamma \vdash \text{LinVar}^-$$
By the definition of context subtraction, \((\Gamma, x : A) - \Gamma = x : A\), thus we can construct the following typing derivation, matching the conclusion:

\[
\frac{}{x : A \vdash x : A} \text{VAR}
\]

2. Case \text{GrVAR}^-

Matching the form of the lemma, we have the derivation:

\[
\exists s, r \supseteq s + 1 \\
\frac{\Gamma, x : [A]_r \vdash A \Rightarrow x; \Gamma, x : [A]_s}{\Gamma, x : [A]_r \vdash x : A} \text{GrVAR}^-
\]

By the definition of context subtraction, \((\Gamma, x : [A]_r) - (\Gamma, x : [A]_s) = x : [A]_q\)

where (1) \(\exists q, r \supseteq q + s\) and \(\forall q', r \supseteq q' + s \Rightarrow q \supseteq q'\).

Applying maximality (1) with \(q = 1\) then we have that \(1 \subseteq q\) (*).

Thus, from this we can construct the typing derivation, matching the conclusion:

\[
\frac{}{x : [A]_1 \vdash x : A} \text{VAR} \\
\frac{x : [A]_1 \vdash x : A}{x : [A]_q \vdash x : A} \text{APPROX}
\]

3. Case \text{R} \Rightarrow \_-

We thus have the derivation:

\[
\frac{\Gamma, x : A \vdash B \Rightarrow t; \Delta \quad x \notin |\Delta|}{\Gamma \vdash A \Rightarrow B \Rightarrow \lambda x.t; \Delta} \text{R} \Rightarrow \_-
\]

By induction we then have that:

\((\Gamma, x : A) - \Delta \vdash t : B\)

Since \(x \notin |\Delta|\) then by the definition of context subtraction we have that \((\Gamma, x : A) - \Delta = (\Gamma - \Delta), x : A\). From this, we can construct the following derivation, matching the conclusion:

\[
\frac{(\Gamma - \Delta), x : A \vdash t : B}{\Gamma - \Delta \vdash \lambda x.t : A \Rightarrow B} \text{ABS}
\]

4. Case \text{L} \Rightarrow \_-

Matching the form of the lemma, the application derivation is:

\[
\frac{\Gamma, x_2 : B \vdash C \Rightarrow t_1; \Delta_1 \quad x_2 \notin |\Delta_1| \quad \Delta_1 \vdash \Rightarrow t_2; \Delta_2}{\Gamma, x_1 : A \Rightarrow B \vdash C \Rightarrow [(x_1 t_2)/x_2]t_1; \Delta_2} \text{L} \Rightarrow \_-
\]

By induction, we have that:

\[
\frac{(\Gamma, x_2 : B) - \Delta_1 \vdash t_1 : C}{(\Delta_1 - \Delta_2) \vdash t_2 : A} \quad (\text{ih1})
\]

\[
\frac{(\Delta_1 - \Delta_2) \vdash \lambda x.t : A}{(\Delta_1 - \Delta_2) \vdash \lambda x.t : A} \quad (\text{ih2})
\]
By the definition of context subtraction and since \( x_2 \not\in |\Delta_1| \) then (ih1) is equal to:

\[
(\Gamma - \Delta_1), x_2 : B \vdash t_1 : C
\]

(ih1')

We can thus construct the following typing derivation, making use of the admissibility of linear substitution (Lemma 1):

\[
\begin{array}{c}
(\Gamma - \Delta_1), x_2 : B \vdash C \vdash t_1 : C \\
\hline
\Gamma - \Delta_1 \vdash \lambda x_2.t_1 : B \Rightarrow \Gamma \Rightarrow \Delta_1 - \Delta_2 \vdash t_2 : A \\
\hline
\Gamma - \Delta_1 \vdash \lambda x_2.t_1 : B \Rightarrow (\Delta_1 - \Delta_2), x_1 : A \Rightarrow B \vdash \Gamma \Rightarrow t_2 : B
\end{array}
\]

From Lemma 5 we have that

\[
((\Gamma - \Delta_1) + (\Delta_1 - \Delta_2)), x_1 : A \Rightarrow B \sqsubseteq (((\Gamma - \Delta_1) + \Delta_1) - \Delta_2), x_1 : A \Rightarrow B
\]

and from Lemma 6 that:

\[
(((\Gamma - \Delta_1) + \Delta_1) - \Delta_2), x_1 : A \Rightarrow B \sqsubseteq (\Gamma - \Delta_2), x_1 : A \Rightarrow B
\]

which, since \( x_1 \) is not in \( \Delta_2 \) (as \( x_1 \) is not in \( \Gamma \)) \( (\Gamma - \Delta_2), x_1 : A \Rightarrow B = (\Gamma, x_1 : A \Rightarrow B) - \Delta_2 \). Applying these inequalities with APP then yields the lemma’s conclusion \( (\Gamma, x_1 : A \Rightarrow B) - \Delta_2 \vdash [x_1 \ t_2]/x_2 t_1 : C \).

5. Case R\(\Box\)

The synthesis rule for boxing can be constructed as:

\[
\Gamma \vdash A \Rightarrow \Box \vdash t : \Delta \\
\hline
\Gamma \vdash \Box \Rightarrow t : \Delta \vdash \Box \Rightarrow \Box \Rightarrow r \Rightarrow (\Gamma - \Delta)
\]

By induction on the premise we get:

\[
\Gamma - \Delta \vdash t : A
\]

Since we apply scalar multiplication in the conclusion of the rule to \( \Gamma - \Delta \) then we know that all of \( \Gamma - \Delta \) must be graded assumptions.

From this, we can construct the typing derivation:

\[
\begin{array}{c}
[\Gamma - \Delta] \vdash t : A \\
\hline
r \Rightarrow [\Gamma - \Delta] \vdash \Box \Rightarrow r \Rightarrow \Box \Rightarrow A
\end{array}
\]

Via Lemma 8 we then have that \( (r \Rightarrow \Gamma - \Delta) \subseteq (\Gamma - (r \Rightarrow (\Gamma - \Delta))) \) thus, we can derived:

\[
\begin{array}{c}
[\Gamma - \Delta] \vdash t : A \\
\hline
r \Rightarrow [\Gamma - \Delta] \vdash \Box \Rightarrow [\Gamma - (r \Rightarrow (\Gamma - \Delta))] \vdash \Box \Rightarrow \Box \Rightarrow A
\end{array}
\]

Satisfying the goal of the lemma.
6. Case L

The synthesis rule for unboxing has the form:

\[
\frac{\Gamma, x_2 : [A]_r \vdash \Rightarrow t; \Delta, x_2 : [A]_s \quad 0 \subseteq s}{\Gamma, x_1 : \square_r A \vdash \Rightarrow \text{let } [x_2] = x_1 \text{ in } t; \Delta} \quad \text{L}^{-}\]

By induction on the premise we have that:

\[
(\Gamma, x_2 : [A]_r) - (\Delta, x_2 : [A]_s) \vdash t : B
\]

By the definition of context subtraction we get that 0 ⊑ r.

We also have that 0 ⊑ s.

By monotonicity with 0 ⊑ q (reflexivity) and 0 ⊑ s then q ⊑ q + s.

From this, we can construct the typing derivation:

\[
\frac{x_1 : \square_r A \vdash x_1 : \square_r A \quad (\Gamma - \Delta), x_2 : [A]_q \vdash \Rightarrow t : B}{\Gamma - \Delta, x_1 : \square_r A \vdash \text{let } [x_2] = x_1 \text{ in } t : B} \quad \text{VAR}
\]

Which matches the goal.

7. Case R

The synthesis rule for pair introduction has the form:

\[
\frac{\Gamma \vdash \Rightarrow t_1; \Delta_1 \vdash \Rightarrow t_2; \Delta_2 \vdash \Delta_2}{\Gamma \vdash \Rightarrow (t_1, t_2); \Delta_2} \quad \text{R}^{-}\]

By induction we get:

\[
\Gamma - \Delta_1 \vdash t_1 : A \quad \text{(ih1)}
\]

\[
\Delta_1 - \Delta_2 \vdash t_2 : B \quad \text{(ih2)}
\]

From this, we can construct the typing derivation:

\[
\frac{(\Gamma - \Delta_1) + (\Delta_1 - \Delta_2) \vdash \Rightarrow (t_1, t_2) : A \otimes B}{(\Gamma - \Delta_1) + (\Delta_1 - \Delta_2) \vdash \Rightarrow (t_1, t_2) : A \otimes B} \quad \text{PAIR}
\]

From Lemma 5, we have that:

\[
(\Gamma - \Delta_1) + (\Delta_1 - \Delta_2) \subseteq ((\Gamma - \Delta_1) + \Delta_1) - \Delta_2
\]

and from Lemma 6, that:

\[
((\Gamma - \Delta_1) + \Delta_1) - \Delta_2 \subseteq \Gamma - \Delta_2
\]

From which we then apply APPROX to the above derivation, yielding the goal \(\Gamma - \Delta_2 \vdash (t_1, t_2) : A \otimes B\).
8. Case $L\otimes^-$

The synthesis rule for pair elimination has the form:

$$
\Gamma, x_1 : A, x_2 : B \vdash C \Rightarrow t_2; \Delta \quad x_1 \notin |\Delta| \quad x_2 \notin |\Delta|

\frac{}{\Gamma, x_3 : A \otimes B \vdash C \Rightarrow \text{let} (x_1, x_2) = x_3 \text{ in } t_2; \Delta} \quad L\otimes^-
$$

By induction we get:

$$(\Gamma, x_1 : A, x_2 : B) - \Delta \vdash t_2 : C$$

since $x_1 \notin |\Delta| \land x_2 \notin |\Delta|$ then $(\Gamma, x_1 : A, x_2 : B) - \Delta = (\Gamma - \Delta), x_1 : A, x_2 : B$. From this, we can construct the following typing derivation, matching the conclusion:

$$
\frac{x_3 : A \otimes B \vdash x_3 : A \otimes B}{(\Gamma - \Delta), x_1 : A, x_2 : B \vdash t_2 : C} \quad \text{VAR}
$$

$$
\frac{(\Gamma - \Delta), x_1 : A, x_2 : B \vdash t_2 : C}{(\Gamma - \Delta), x_3 : A \otimes B \vdash \text{let} (x_1, x_2) = x_3 \text{ in } t_2 : C} \quad \text{CASE}
$$

which matches the conclusion since $(\Gamma - \Delta), x_3 : A \otimes B = (\Gamma, x_3 : A \otimes B) - \Delta$ since $x_3 \notin |\Delta|$ by its disjointness from $\Gamma$.

9. Case $R\oplus_1^-$ and $R\oplus_2^-$

The synthesis rules for sum introduction are straightforward. For $R\oplus_1^-$ we have the rule:

$$
\Gamma \vdash A \Rightarrow t; \Delta

\frac{}{\Gamma \vdash A \oplus B \Rightarrow \text{inl } t; \Delta} \quad R\oplus_1^-
$$

By induction we have:

$$
\Gamma - \Delta \vdash t : A
$$

(ii1)

from which we can construct the typing derivation, matching the conclusion:

$$
\frac{\Gamma - \Delta \vdash t : A}{\Gamma - \Delta \vdash \text{inl } t : A} \quad R\oplus_1^-
$$

Matching the goal. And likewise for $R\oplus_2^-$.  

10. Case $L\oplus^-$

The synthesis rule for sum elimination has the form:

$$
\Gamma, x_2 : A \vdash C \Rightarrow t_1; \Delta_1 \quad \Gamma, x_3 : B \vdash C \Rightarrow t_2; \Delta_2 \quad x_2 \notin |\Delta_1| \quad x_3 \notin |\Delta_2|

\frac{}{\Gamma, x_1 : A \oplus B \vdash C \Rightarrow \text{case } x_1 \text{ of inl } x_2 \rightarrow t_1 | \text{inr } x_3 \rightarrow t_2; \Delta_1 \cap \Delta_2} \quad L\oplus^-
$$

By induction:

$$(\Gamma, x_2 : A) - \Delta_1 \vdash t_1 : C \quad \text{(ih)}$$

$$(\Gamma, x_3 : B) - \Delta_2 \vdash t_2 : C \quad \text{(ih)}$$

From this we can construct the typing derivation, matching the conclusion:

$$
\frac{x_1 : A \oplus B \vdash t_1 : A \oplus B}{(\Gamma - \Delta_1), x_2 : A \vdash t_2 : C} \quad \text{VAR}
$$

$$
\frac{(\Gamma - \Delta_2), x_3 : B \vdash t_3 : C}{(\Gamma - \Delta_1 \cap \Delta_2), x_1 : A \oplus B \vdash \text{case } t_1 \text{ of inl } x_2 \rightarrow t_2 | \text{inr } x_3 \rightarrow t_3 : C} \quad \text{CASE}
$$

$$
(\Gamma, x_1 : A \oplus B) - (\Delta_1 \cap \Delta_2) \vdash \text{case } t_1 \text{ of inl } x_2 \rightarrow t_2 | \text{inr } x_3 \rightarrow t_3 : C
$$
11. Case $R1^{-}$

\[
\frac{\Gamma \vdash 1 \Rightarrow \bot; \Gamma}{\bigcirc}
\]

By Lemma 7 we have that $\emptyset \subseteq \Gamma - \Gamma$ then we have:

\[
\frac{\emptyset \vdash () : 1}{\bigcirc} \text{APPROX}
\]

Matching the goal

12. Case $L1^{-}$

\[
\frac{\Gamma \vdash C \Rightarrow t; \Delta}{\bigcirc}
\]

By induction we have:

\[
\frac{\Gamma - \Delta \vdash t : C}{\text{ih}}
\]

Then we make the derivation:

\[
\frac{x : 1 \vdash x : 1 \text{VAR}}{(\Gamma - \Delta), x : 1 \vdash \text{let} () = x \text{in} t : C \text{LET1}}
\]

where the context is equal to $(\Gamma, x : 1) - \Delta$.

13. Case $\text{der}^{-}$

\[
\frac{\Gamma, x : [A] ; y : A \vdash B \Rightarrow t; \Delta, x : [A] ; y \notin |\Delta| \qquad \exists s, r \equiv s + 1 \text{DER}^{-}}{\Gamma, x : [A], r \vdash B \Rightarrow [x/y] t; \Delta, x : [A] ; r \text{DER}^{-}}
\]

By induction:

\[
(\Gamma, x : [A] ; y : A) - (\Delta, x : [A] ; y) \vdash t : B \text{ (ih)}
\]

By the definition of context subtraction we have (since also $y \notin |\Delta|$)

\[
(\Gamma, x : [A] ; y : A) - (\Delta, x : [A] ; y)
= (\Gamma - \Delta), x : [A] ; y : A
\]

where $\exists q, s \sqsupseteq q + s'$ (1) and $\forall q, s \sqsupseteq q + s' \Rightarrow q \sqsupseteq \hat{q}$ (2)
The goal context is computed by:

\[(\Gamma, x : [A]_r) - (\Delta, x : [A]_{s'})\]

\[= (\Gamma - \Delta), x : [A]_{q'}\]

where \(r \supseteq q' + s'\) (3) and \(\forall \hat{q'}, r \supseteq q' + s' \Rightarrow q' \supseteq \hat{q'}\) (4)

From the premise of \(\text{der}^-\) we have \(r \supseteq (s + 1)\).

Using this last result we derive:

\[
\frac{(\Gamma - \Delta), x : [A]_{q}, y : A \vdash t : B}{\text{DER}} \quad \frac{(\Gamma - \Delta), x : [A]_{q+1} \vdash [x/y]t : B}{\text{CONTRACTION}} \quad \frac{(\Gamma - \Delta), x : [A]_{q'} \vdash [x/y]t : B}{\text{APPROX}} (8)
\]

Which matches the goal.

**Lemma 3 (Additive synthesis soundness).** For all \(\Gamma\) and \(A\):

\[\Gamma \vdash A \Rightarrow t; \Delta \quad \Rightarrow \quad \Delta \vdash t : A\]

Appendix E gives the proof.

**Proof.** 1. Case \(\text{LinVar}^+\)

In the case of linear variable synthesis, we have the derivation:

\[
\frac{\Gamma, x : A \vdash t : B}{\text{LinVar}^+}
\]

Therefore we can construct the following typing derivation, matching the conclusion:

\[
\frac{x : A \vdash x : A}{\text{VAR}}
\]

2. Case \(\text{GrVar}^+\)

Matching the form of the lemma, we have the derivation:

\[
\frac{\Gamma, x : [A]_r \vdash t : B}{\text{GrVar}^+}
\]

From this we can construct the typing derivation, matching the conclusion:

\[
\frac{x : A \vdash x : A}{\text{VAR}} \quad \frac{x : [A]_r \vdash x : [A]_1}{\text{DER}}
\]

34
3. Case $\text{R} \rightarrow^{*}$

We thus have the derivation:

$$
\begin{align*}
\Gamma, x : A &\vdash B \Rightarrow^* t; \Delta, x : A \\
\Gamma &\vdash A \rightarrow B \Rightarrow^* \lambda x.t; \Delta
\end{align*}
$$

By induction on the premise we then have:

$$\Delta, x : A \vdash t : B$$

From this, we can construct the typing derivation, matching the conclusion:

$$\begin{align*}
\Delta, x : A \vdash t : B \\
\Delta \vdash \lambda x.t : A \rightarrow B
\end{align*}$$

4. Case $\text{L} \rightarrow^{*}$

Matching the form of the lemma, the application derivation can be constructed as:

$$
\begin{align*}
\Gamma, x_2 : B &\vdash C \Rightarrow^* t_1; \Delta_1, x_2 : B \\
\Gamma &\vdash A \Rightarrow^* t_2; \Delta_2 \\
\Gamma, x_1 : A \rightarrow B &\vdash C \Rightarrow^* [(x_1 t_2)/x_2]t_1; (\Delta_1 + \Delta_2), x_1 : A \rightarrow B
\end{align*}
$$

By induction on the premises we then have the following typing judgments:

$$\Delta_1, x_2 : B \vdash t_1 : C$$

$$\Delta_2 \vdash t_2 : A$$

We can thus construct the following typing derivation, making use of the admissibility of linear substitution (Lemma [1]):

$$
\begin{align*}
x_1 : A \rightarrow B &\vdash x_1 : A \rightarrow B \quad \text{VAR} \\
\Delta_2 &\vdash t_2 : A \\
\Delta_1, x_2 : B &\vdash t_1 : C
\end{align*}
$$

(L. 1)

5. Case $\text{R} \Box^{+}$

The synthesis rule for boxing can be constructed as:

$$
\begin{align*}
\Gamma &\vdash A \Rightarrow^* t; \Delta \\
\Gamma \vdash \Box_r A \Rightarrow^* [t]; r * \Delta
\end{align*}
$$

By induction we then have:

$$\Delta \vdash t : A$$

In the conclusion of the above derivation we know that $r * \Delta$ is defined, therefore it must be that all of $\Delta$ are graded assumptions, i.e., we have that $[\Delta]$ holds. We can thus construct the following typing derivation, matching the conclusion:

$$
\begin{align*}
[\Delta] &\vdash t : A \\
r \ast [\Delta] &\vdash [t] : \Box_r A
\end{align*}
$$
6. Case \( \text{DER}^+ \)

From the dereliction rule we have:

\[
\Gamma, x : [A]_s, y : A \vdash t \Rightarrow \Delta, y : A \quad \text{DER}^+
\]

By induction we get:

\[
\Delta, y : A \vdash t : B \quad \text{(ih)}
\]

Case on \( x \in \Delta \)

- \( x \in \Delta \), i.e., \( \Delta = \Delta', x : [A]_{s'} \).

Then by admissibility of contraction we can derive:

\[
\Delta', x : [A]_{s'}, y : [A]_1 \vdash t : B \quad \text{DER}
\]

\[
(\Delta', x : [A]_{s'}) + x : [A]_1 \vdash [x/y]t : B
\]

Satisfying the lemma statement.

- \( x \not\in \Delta \). Then again from the admissibility of contraction, we derive the typing:

\[
\Delta, y : A \vdash t : B
\]

which is well defined as \( x \not\in \Delta \) and gives the lemma conclusion.

7. Case \( \text{L} \square^+ \)

The synthesis rule for unboxing has the form:

\[
\Gamma, x_2 : [A]_r \vdash B \Rightarrow t; \Delta \quad \text{if} \ x_2 : [A]_s \in \Delta \text{ then } s \subseteq r \text{ else } 0 \subseteq r \quad \text{L} \square^+
\]

By induction we have that:

\[
\Delta \vdash t : B \quad \text{(ih)}
\]

Case on \( x_2 : [A]_s \in \Delta \)

- \( x_2 : [A]_s \in \Delta \), i.e., \( s \subseteq r \).

From this, we can construct the typing derivation, matching the conclusion:

\[
x_1 : \square_r A \vdash x_1 : \square_r A \quad \text{VAR}
\]

\[
\Delta, x_2 : [A]_r \vdash t : B \quad \text{LET}\square
\]

- \( x_2 : [A]_s \not\in \Delta \), i.e., \( 0 \subseteq r \).

From this, we can construct the typing derivation, matching the conclusion:

\[
x_1 : \square_r A \vdash x_1 : \square_r A \quad \text{VAR}
\]

\[
\Delta, x_2 : [A]_r \vdash t : B \quad \text{WEAK}
\]

\[
\Delta, x_3 : \square_r A \vdash \text{LET} x_3 = x_1 \text{ in } t : B \quad \text{APPROX}
\]
8. Case $\text{R} \otimes^+$

The synthesis rule for pair introduction has the form:

\[
\frac{\Gamma \vdash t_1 : A \quad \Delta \vdash t_2 : B}{\Gamma \vdash \langle t_1, t_2 \rangle : A \otimes B} \quad \text{R} \otimes^+
\]

By induction on the premises we have that:

\[
\begin{array}{ll}
\Delta_1 \vdash t_1 : A & \quad \text{(ih1)} \\
\Delta_2 \vdash t_2 : B & \quad \text{(ih2)} \\
\end{array}
\]

From this, we can construct the typing derivation, matching the conclusion:

\[
\frac{\Delta_1 \vdash t_1 : A \quad \Delta_2 \vdash t_2 : B}{\Delta_1 + \Delta_2 \vdash \langle t_1, t_2 \rangle : A \otimes B} \quad \text{PAIR}
\]

9. Case $\text{L} \otimes^+$

The synthesis rule for pair elimination has the form:

\[
\frac{\Gamma, x_1 : A, x_2 : B \vdash t_2 : C}{\Gamma, x_3 : A \otimes B \vdash \text{let} (x_1, x_2) = x_3 \text{ in } t_2 : C} \quad \text{L} \otimes^+
\]

By induction on the premises we have that:

\[
\begin{array}{ll}
\Delta_1 \vdash t_1 : A & \quad \text{(ih1)} \\
\Delta_2 \vdash t_2 : B & \quad \text{(ih2)} \\
\end{array}
\]

From this, we can construct the typing derivation, matching the conclusion:

\[
\frac{\Delta_1 \vdash t_1 : A \quad \Delta_2 \vdash t_2 : B}{\Delta, x_3 : A \otimes B \vdash \text{let} (x_1, x_2) = x_3 \text{ in } t_2 : C} \quad \text{LETPAIR}
\]

10. Case $\text{R} \oplus^+_1$ and $\text{R} \oplus^+_2$

The synthesis rules for sum introduction are straightforward. For $\text{R} \oplus^+_1$ we have the rule:

\[
\frac{\Gamma \vdash t : A \quad \Delta}{\Gamma \vdash \text{inl } t : A \oplus B} \quad \text{R} \oplus^+_1
\]

By induction on the premises we have that:

\[
\Delta \vdash t : A \quad \text{(ih)}
\]

From this, we can construct the typing derivation, matching the conclusion:

\[
\frac{\Delta \vdash t : A}{\Delta \vdash \text{inl } t : A \oplus B} \quad \text{INL}
\]
Likewise, for the $R \oplus^+_2$ we have the synthesis rule:

$$
\frac{\Gamma \vdash B \Rightarrow^+ t; \Delta}{\Gamma \vdash A \oplus B \Rightarrow^+ \text{inr } t; \Delta}
$$

By induction on the premises we have that:

$$\Delta \vdash t : B$$  \hspace{1cm} (ih)

From this, we can construct the typing derivation, matching the conclusion:

$$\Delta \vdash t : B \hspace{1cm} \Delta \vdash \text{inr } t : A \oplus B$$

11. Case $L \oplus^+$

The synthesis rule for sum elimination has the form:

$$
\frac{\Gamma, x_2 : A \vdash C \Rightarrow^+ t_1; \Delta_1, x_2 : A \quad \Gamma, x_3 : B \vdash C \Rightarrow^+ t_2; \Delta_2, x_3 : B}{\Gamma, x_1 : A \oplus B \vdash \text{case } x_1 \text{ of inl } x_2 \rightarrow t_1 | \text{inr } x_3 \rightarrow t_2; (\Delta_1 \sqcup \Delta_2), x_1 : A \oplus B \Rightarrow^+}
$$

By induction on the premises we have that:

$$\Delta_1, x_2 : A \vdash t_1 : C \hspace{1cm} (ih1)$$
$$\Delta_2, x_3 : B \vdash t_2 : C \hspace{1cm} (ih2)
$$

From this, we can construct the typing derivation, matching the conclusion:

$$
\frac{x_1 : A \oplus B \vdash x_1 : A \oplus B \hspace{1cm} \Delta_1, x_2 : A \vdash t_1 : C \hspace{1cm} \Delta_2, x_3 : B \vdash t_2 : C}{(\Delta_1 \sqcup \Delta_2), x_1 : A \oplus B \vdash \text{case } x_1 \text{ of inl } x_2 \rightarrow t_1 | \text{inr } x_3 \rightarrow t_2; C \hspace{1cm} \text{CASE}}
$$

12. Case $R1^+$

The synthesis rule for unit introduction has the form:

$$\frac{\Gamma \vdash 1 \Rightarrow^+ () ; \emptyset}{\Gamma \vdash 1 \Rightarrow^+}$$

From this, we can construct the typing derivation, matching the conclusion:

$$\emptyset \vdash () : 1$$

13. Case $L1^+$

The synthesis rule for unit elimination has the form:

$$\frac{\Gamma \vdash C \Rightarrow^+ t ; \Delta}{\Gamma, x : 1 \vdash C \Rightarrow^+ \text{let } () = x \in t ; \Delta, x : 1 \Rightarrow^+}$$

38
By induction on the premises we have that:

$$\Delta \vdash t : C$$  \hspace{1cm} (ih)

From this, we can construct the typing derivation, matching the conclusion:

$$\frac{x : 1 \vdash x : 1 \text{ VAR} \quad \Delta \vdash t : C \quad \Delta, x : 1 \vdash \text{let } () = x \text{ in } t : C \text{ LET1}}{\Delta, x : 1 \vdash t : C \text{ MATCH}}$$

**Lemma 4 (Additive pruning synthesis soundness).** For all $\Gamma$ and $A$:

$$\Gamma \vdash A \Rightarrow^+ t; \Delta \quad \Rightarrow \quad \Delta \vdash t : A$$

*Appendix E* gives the proof.

**Proof.** The cases for the rules in the additive pruning synthesis calculus are equivalent to lemma (3), except for the cases of the $L' \o \Rightarrow^+$ and $R' \otimes \Rightarrow^+$ rules which we consider here:

1. Case $L' \o \Rightarrow^+$

Matching the form of the lemma, the application derivation can be constructed as:

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^+ t_1; \Delta_1, x_2 : B \quad \Gamma - \Delta_1 \vdash A \Rightarrow^+ t_2; \Delta_2}{\Gamma, x_1 : A \o B \vdash C \Rightarrow^+ (x_1 t_2)/x_2 t_1; (\Delta_1 + \Delta_2), x_1 : A \o B \text{ L' } \o \Rightarrow^+}$$

By induction on the premises we then have the following typing judgments:

$$\Delta_1, x_2 : B \vdash t_1 : C$$

$$\Delta_2 \vdash t_2 : A$$

We can thus construct the following typing derivation, making use of the admissibility of linear substitution (Lemma 1):

$$\frac{x_1 : A \o B \vdash x_1 : A \o B \text{ VAR} \quad \Delta_2 \vdash t_2 : A \text{ APP} \quad \Delta_1 \vdash t_1 : C}{\Delta_2, x_1 : A \o B \vdash x_1 t_2 : B \quad (\Delta_1 + \Delta_2), x_1 : A \o B \vdash [(x_1 t_2)/x_2] t_1 : C \text{ (L. 1)}}$$

2. Case $R' \otimes \Rightarrow^+$

The synthesis rule for the pruning alternative for pair introduction has the form:

$$\frac{\Gamma \vdash A \Rightarrow^+ t_1; \Delta_1 \quad \Gamma - \Delta_1 \vdash B \Rightarrow^+ t_2; \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^+ \langle t_1, t_2 \rangle; \Delta_1 + \Delta_2 \text{ R' } \otimes \Rightarrow^+}$$

39
By induction on the premises we have that:

\[ \Delta_1 \vdash t_1 : A \quad (ih1) \]
\[ \Delta_2 \vdash t_2 : B \quad (ih2) \]

From this, we can construct the typing derivation, matching the conclusion:

\[ \Delta_1 \vdash t_1 : A \quad \Delta_2 \vdash t_2 : B \quad PAIR \]

\[ \Delta_1 + \Delta_2 \vdash \langle t_1, t_2 \rangle : A \otimes B \]

**Lemma 9 (Soundness of focusing for subtractive synthesis).** For all contexts \( \Gamma, \Omega \) and types \( A \) then:

1. **Right Async**: \( \Gamma; \Omega \vdash A \uparrow \Rightarrow \neg t; \Delta \implies \Gamma, \Omega \vdash A \Rightarrow \neg t; \Delta \)
2. **Left Async**: \( \Gamma; \Omega \uparrow \vdash C \Rightarrow \neg t; \Delta \implies \Gamma, \Omega \vdash C \Rightarrow \neg t; \Delta \)
3. **Right Sync**: \( \Gamma; \emptyset \vdash A \downarrow \Rightarrow \neg t; \Delta \implies \Gamma \vdash A \Rightarrow \neg t; \Delta \)
4. **Left Sync**: \( \Gamma; x : A \downarrow \vdash C \Rightarrow \neg t; \Delta \implies \Gamma, x : A \vdash C \Rightarrow \neg t; \Delta \)
5. **Focus Right**: \( \Gamma; \Omega \uparrow \vdash C \Rightarrow \neg t; \Delta \implies \Gamma \vdash C \Rightarrow \neg t; \Delta \)
6. **Focus Left**: \( \Gamma; x : A; \Omega \uparrow \vdash C \Rightarrow \neg t; \Delta \implies \Gamma \vdash C \Rightarrow \neg t; \Delta \)

**Proof.** 1. Case 1. Right Async:

(a) Case \( R \rightarrow^- \)

In the case of the right asynchronous rule for abstraction introduction, the synthesis rule has the form:

\[ \frac{\Gamma, \Omega, x : A \vdash B \uparrow \Rightarrow \neg t; \Delta \quad x \notin |\Delta|}{\Gamma, \Omega \vdash A \Rightarrow \neg B \uparrow \Rightarrow \neg \lambda x.t; \Delta} \quad R \rightarrow^- \]

By induction on the first premise, we have that:

\[ (\Gamma, \Omega), x : A \vdash A \Rightarrow \neg t; \Delta \quad (ih) \]

from case 1 of the lemma. From which, we can construct the following instantiation of the \( R \rightarrow^- \) synthesis rule in the non-focusing calculus:

\[ (\Gamma, \Omega), x : A \vdash B \Rightarrow \neg t; \Delta \quad x \notin |\Delta| \]

\[ \frac{\Gamma, \Omega \vdash A \Rightarrow \neg B \uparrow \Rightarrow \neg \lambda x.t; \Delta}{\Gamma, \Omega \vdash A \rightarrow B \Rightarrow \neg \lambda x.t; \Delta} \quad R \rightarrow^- \]

(b) Case \( R \uparrow^- \)

In the case of the right asynchronous rule for transition to a left asynchronous judgement, the synthesis rule has the form:

\[ \frac{\Gamma; \Omega \uparrow \vdash C \Rightarrow \neg t; \Delta \quad C \text{ not right async}}{\Gamma; \Omega \vdash C \uparrow \Rightarrow \neg t; \Delta} \quad R \uparrow^- \]

By induction on the first premise, we have that:

\[ \Gamma, \Omega \vdash C \Rightarrow \neg t; \Delta \]

from case 2 of the lemma.
2. Case 2. Left Async:

(a) Case $L \otimes^-$
In the case of the left asynchronous rule for pair elimination, the synthesis rule has the form:

$$
\frac{
\Gamma; \Omega, x_1 : A, x_2 : B \vdash C \Rightarrow t_2; \Delta \quad \text{let } (x_1, x_2) = x_3 \text{ in } t_2; \Delta
}{\Gamma; \Omega, x_3 : A \otimes B \vdash C \Rightarrow t_2; \Delta}
$$

$L \otimes^-$

By induction on the first premise, we have that:

$$(\Gamma, \Omega), x_1 : A, x_2 : B \vdash C \Rightarrow t; \Delta$$

From which, we can construct the following instantiation of the $R \otimes^-$ synthesis rule in the non-focusing calculus:

$$(\Gamma, \Omega), x_1 : A, x_2 : B \vdash C \Rightarrow t; \Delta \quad \text{let } (x_1, x_2) = x_3 \text{ in } t; \Delta$$

$L \otimes^-$

(b) Case $L \oplus^-$
In the case of the left asynchronous rule for sum elimination, the synthesis rule has the form:

$$
\frac{
\Gamma; \Omega, x_2 : A \vdash C \Rightarrow t_1; \Delta_1 \quad \Gamma; \Omega, x_3 : B \vdash C \Rightarrow t_2; \Delta_2 \quad x_2 \not\in |\Delta_1| \quad x_3 \not\in |\Delta_2|
}{\Gamma; \Omega, x_1 : A \oplus B \vdash C \Rightarrow t_1 \text{ if } x_2 \in A \text{ or } t_2 \text{ if } x_2 \in B; \Delta_1 \cap \Delta_2}
$$

$L \oplus^-$

By induction on the first and second premises, we have that:

$$(\Gamma, \Omega), x_2 : A \vdash C \Rightarrow t_1; \Delta_1$$

$$(\Gamma, \Omega), x_3 : B \vdash C \Rightarrow t_2; \Delta_2$$

From case 2 of the lemma. From which, we can construct the following instantiation of the $L \oplus^-$ synthesis rule in the non-focusing calculus:

$$(\Gamma, \Omega), x_1 : A \oplus B \vdash C \Rightarrow \text{ case } x_1 \text{ of inl } x_2 \rightarrow t_1 | \text{ inr } x_3 \rightarrow t_2; \Delta_1 \cap \Delta_2$$

$L \oplus^-$

(c) Case $L^1^-$
In the case of the left asynchronous rule for unit elimination, the synthesis rule has the form:

$$
\frac{
\Gamma; \emptyset \vdash C \Rightarrow t; \Delta
}{\Gamma; x : 1 \vdash C \Rightarrow t; \Delta}
$$

$L^1^-$

By induction on the premise, we have that:

$$\Gamma \vdash C \Rightarrow t; \Delta$$

(ih)
from case 2 of the lemma. From which, we can construct the following instantiation of the $L_{1^-}$ synthesis rule in the non-focusing calculus matching the conclusion:

\[
\Gamma, x : 1 \vdash C \Rightarrow t; \Delta \quad L_{1^-}
\]

(d) Case $L \Box$

In the case of the left asynchronous rule for graded modality elimination, the synthesis rule has the form:

\[
\Gamma; \Omega, x_2 : [A]_r \uparrow \vdash B \Rightarrow t; \Delta, x_2 : [A]_s, 0 \subseteq s \quad L_{\Box^-}
\]

By induction on the first premise, we have that:

\[
\Gamma, \Omega, x_2 : [A]_r \vdash B \Rightarrow t; \Delta, x_2 : [A]_s \quad (ih)
\]

from case 2 of the lemma. From which, we can construct the following instantiation of the $L_{\Box^-}$ synthesis rule in the non-focusing calculus:

\[
\Gamma, \Omega, x_1 : [\Box_r A]_r \vdash B \Rightarrow t; \Delta, x_2 : [A]_s, 0 \subseteq s \\
\Gamma; \Omega, x_1 : [\Box_r A]_r \vdash B \Rightarrow \text{let } [x_2] = x_1 \text{ in } t; \Delta \quad L_{\Box^-}
\]

(e) Case $der^-$

In the case of the left asynchronous rule for dereliction, the synthesis rule has the form:

\[
\Gamma, x : [A]_s, y : A \vdash B \Rightarrow t; \Delta, x : [A]_{s'}, y \notin \Delta \quad \exists s, r \supseteq s + 1 \quad der^{-}
\]

By induction on the first premise, we have that:

\[
\Gamma, x : [A]_s, y : A \vdash B \Rightarrow t; \Delta, x : [A]_{s'} \quad (ih)
\]

from case 2 of the lemma. From which, we can construct the following instantiation of the $der^{-}$ synthesis rule in the non-focusing calculus:

\[
\Gamma, x : [A]_s, y : A \vdash B \Rightarrow t; \Delta, x : [A]_{s'}, y \notin \Delta \quad \exists s, r \supseteq s + 1 \\
\Gamma, x : [A]_s, y : A \vdash B \Rightarrow [x/y] t; \Delta, x : [A]_{s'} \quad der^{-}
\]

(f) Case $L \uparrow^-$

In the case of the left asynchronous rule for transitioning an assumption from the focusing context $\Omega$ to the non-focusing context $\Gamma$, the synthesis rule has the form:

\[
\Gamma, x : A; \Omega \uparrow \vdash C \Rightarrow t; \Delta \quad \text{A not left async} \\
\Gamma; \Omega, x : A \uparrow \vdash C \Rightarrow t; \Delta \quad L_{\uparrow^-}
\]

By induction on the first premise, we have that:

\[
\Gamma, x : A, \Omega \vdash C \Rightarrow t; \Delta \quad (ih)
\]

from case 2 of the lemma.
3. Case 3. Right Sync:

(a) Case $R \otimes^-$
In the case of the right synchronous rule for pair introduction, the synthesis rule has the form:

$$\frac{\Gamma; \emptyset \vdash A \Downarrow \Rightarrow t_1; \Delta_1 \quad \Delta_1; \emptyset \vdash B \Downarrow \Rightarrow t_2; \Delta_2}{\Gamma; \emptyset \vdash A \otimes B \Downarrow \Rightarrow \langle t_1, t_2 \rangle; \Delta_2} R \otimes^-$$

By induction on the first and second premises, we have that:

$$\Gamma \vdash A \Rightarrow t_1; \Delta_1$$  \hspace{1cm} (ih1)

$$\Delta_1 \vdash B \Rightarrow t_2; \Delta_2$$  \hspace{1cm} (ih2)

from case 3 of the lemma. From which, we can construct the following instantiation of the $R \otimes^-$ synthesis rule in the non-focusing calculus:

$$\frac{\Gamma \vdash A \Rightarrow t_1; \Delta_1 \quad \Delta_1 \vdash B \Rightarrow t_2; \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow \langle t_1, t_2 \rangle; \Delta_2} R \otimes^-$$

(b) Case $R \oplus_1^-$ and $R \oplus_2^-$
In the case of the right synchronous rules for sum introduction, the synthesis rules has the form:

$$\frac{\Gamma; \emptyset \vdash A \Downarrow \Rightarrow t; \Delta}{\Gamma; \emptyset \vdash A \oplus B \Downarrow \Rightarrow \text{inl} t; \Delta} R \oplus_1^-$$

$$\frac{\Gamma; \emptyset \vdash B \Downarrow \Rightarrow t; \Delta}{\Gamma; \emptyset \vdash A \oplus B \Downarrow \Rightarrow \text{inr} t; \Delta} R \oplus_2^-$$

By induction on the premises of these rules, we have that:

$$\Gamma \vdash A \Rightarrow t; \Delta$$  \hspace{1cm} (ih1)

$$\Gamma \vdash B \Rightarrow t; \Delta$$  \hspace{1cm} (ih2)

from case 3 of the lemma. From which, we can construct the following instatiation of the $R \oplus_1^-$ and $R \oplus_2^-$ rule in the non-focusing calculus, respectively:

$$\frac{\Gamma \vdash A \Rightarrow t; \Delta}{\Gamma \vdash A \oplus B \Rightarrow \text{inl} t; \Delta} R \oplus_1^-$$

$$\frac{\Gamma \vdash B \Rightarrow t; \Delta}{\Gamma \vdash A \oplus B \Rightarrow \text{inr} t; \Delta} R \oplus_2^-$$
(c) Case R1⁻
In the case of the right synchronous rule for unit introduction, the synthesis rule has the form:

\[ \Gamma \vdash 1 \Rightarrow \_ ; \Gamma \]

From which, we can construct the following instantiation of the R1⁻ synthesis rule in the non-focusing calculus:

\[ \Gamma, \Omega \vdash 1 \Rightarrow \_ ; \Gamma \]

(d) Case R□⁻
In the case of the right synchronous rule for graded modality introduction, the synthesis rule has the form:

\[ \Gamma; \emptyset \vdash A \uparrow \Rightarrow t ; \Delta \]
\[ \Gamma; \emptyset \vdash \Box_r A \downarrow \Rightarrow t ; \Gamma - r \ast (\Gamma - \Delta) \]

R□⁻

By induction on the premise, we have that:

\[ \Gamma \vdash A \Rightarrow t ; \Delta \]

from case 1 of the lemma. From which, we can construct the following instantiation of the R□⁻ synthesis rule in the non-focusing calculus:

\[ \Gamma \vdash A \Rightarrow t ; \Delta \]
\[ \Gamma \vdash \Box_r A \Rightarrow [t] ; \Gamma - r \ast (\Gamma - \Delta) \]

R□⁻

(e) Case R⇓⁻
In the case of the right synchronous rule for transitioning back to an asynchronous judgement, the synthesis rule has the form:

\[ \Gamma; \emptyset \vdash A \uparrow \Rightarrow t ; \Delta \]
\[ \Gamma; \emptyset \vdash A \downarrow \Rightarrow t ; \Delta \]

R⇓⁻

By induction on the premise, we have that:

\[ \Gamma \vdash A \Rightarrow t ; \Delta \]

from case 1 of the lemma.

4. Case 4. Left Sync

(a) Case L−œ⁻
In the case of the left synchronous rule for application, the synthesis rule has the form:

\[ \Gamma; x_2 : B \downarrow \vdash C \Rightarrow t_1 ; \Delta_1 \]
\[ x_2 \notin | \Delta_1 | \]
\[ \Delta_1 ; \emptyset \vdash A \downarrow \Rightarrow t_2 ; \Delta_2 \]

L−œ⁻
By induction on the first premise, we have that:

\[ \Gamma, x_2 : B \vdash C \Rightarrow t_1; \Delta_1 \]  
\(\text{(ih1)}\)

from case 4 of the lemma. By induction on the third premise, we have that:

\[ \Delta_1 \vdash A \Rightarrow t_2; \Delta_2 \]  
\(\text{(ih2)}\)

from case 3 of the lemma. From which, we can construct the following instantiation of the \(L \downarrow\) synthesis rule in the non-focusing calculus:

\[
\frac{
\Gamma, x_2 : B \vdash C \Rightarrow t_1; \Delta_1 \quad x_2 \not\in |\Delta_1| \quad \Delta_1 \vdash A \Rightarrow t_2; \Delta_2
}{
\Gamma, x_1 : A \Rightarrow (x_1 t_2)/t_2] t_1; \Delta_2
} \quad \text{L}_{\downarrow}
\]

(b) Case \(\text{LinVar}^-\)

In the case of the left synchronous rule for linear variable synthesis, the synthesis rule has the form:

\[
\frac{
\Gamma, x : A \Downarrow \vdash A \Rightarrow x; \Gamma
}{
\text{LinVar}^-}
\]

From which, we can construct the following instantiation of the \(\text{LinVar}^-\) synthesis rule in the non-focusing calculus:

\[
\frac{
\Gamma, x : A \vdash A \Rightarrow x; \Gamma
}{
\text{LinVar}^-}
\]

(c) Case \(\text{GrVar}^-\)

In the case of the left synchronous rule for graded variable synthesis, the synthesis rule has the form:

\[
\frac{
\exists s. r \geq s + 1
}{
\frac{
\Gamma, x : [A]_r \Downarrow \vdash A \Rightarrow x; \Gamma, x : [A]_s
}{
\text{GrVar}^-}
}\]

From which, we can construct the following instantiation of the \(\text{GrVar}^-\) synthesis rule in the non-focusing calculus:

\[
\exists s. r \geq s + 1
\frac{
\Gamma, x : A \vdash A \Rightarrow x; \Gamma, x : [A]_s
}{
\text{GrVar}^-}
\]

(d) Case \(L \Uparrow^-\)

In the case of the left synchronous rule for transitioning back to an asynchronous judgement, the synthesis rule has the form:

\[
\frac{
\Gamma, x : A \Uparrow \vdash C \Rightarrow t; \Delta \quad \text{A not atomic and not left sync}
}{
\Gamma, x : A \Downarrow \vdash C \Rightarrow t; \Delta
} \quad \text{L}_{\Uparrow^-}
\]

By induction on the premise, we have that:

\[ \Gamma, x : A \vdash C \Rightarrow t; \Delta \]  
\(\text{(ih)}\)

from case 2 of the lemma.
5. Case 5. Focus Right: focusR

In the case of the focusing rule for transitioning from a left asynchronous judgement to a right synchronous judgement, the synthesis rule has the form:

\[
\frac{\Gamma; \emptyset \vdash C \downarrow \Rightarrow t; \Delta \quad \text{C not atomic}}{\Gamma; \emptyset \uparrow \vdash C \Rightarrow t; \Delta \quad \text{focusR}}
\]

By induction on the first premise, we have that:

\[
\Gamma \vdash C \Rightarrow t; \Delta \quad \text{(ih)}
\]

from case 2 of the lemma.

6. Case 6. Focus Left focusL

In the case of the focusing rule for transitioning from a left asynchronous judgement to a left synchronous judgement, the synthesis rule has the form:

\[
\frac{\Gamma; x : A \downarrow \vdash C \Rightarrow t; \Delta}{\Gamma, x : A; \emptyset \uparrow \vdash C \Rightarrow t; \Delta \quad \text{focusL}}
\]

By induction on the first premise, we have that:

\[
\Gamma, x : A \vdash C \Rightarrow t; \Delta \quad \text{(ih)}
\]

from case 2 of the lemma.

**Lemma 10 (Soundness of focusing for additive synthesis).** For all contexts \( \Gamma, \Omega \) and types \( A \) then:

1. Right Async \( \Gamma; \Omega \vdash A \uparrow \Rightarrow^+ t; \Delta \quad \Longrightarrow \quad \Gamma, \Omega \vdash A \Rightarrow^+ t; \Delta \)
2. Left Async \( \Gamma; \Omega \uparrow \vdash C \Rightarrow^+ t; \Delta \quad \Longrightarrow \quad \Gamma, \Omega \vdash C \Rightarrow^+ t; \Delta \)
3. Right Sync \( \Gamma; \emptyset \vdash A \downarrow \Rightarrow^+ t; \Delta \quad \Longrightarrow \quad \Gamma \vdash A \Rightarrow^+ t; \Delta \)
4. Left Sync \( \Gamma; x : A \downarrow \vdash C \Rightarrow^+ t; \Delta \quad \Longrightarrow \quad \Gamma, x : A \vdash C \Rightarrow^+ t; \Delta \)
5. Focus Right \( \Gamma; \Omega \uparrow \vdash C \Rightarrow^+ t; \Delta \quad \Longrightarrow \quad \Gamma \vdash C \Rightarrow^+ t; \Delta \)
6. Focus Left \( \Gamma, x : A; \Omega \uparrow \vdash C \Rightarrow^+ t; \Delta \quad \Longrightarrow \quad \Gamma \vdash C \Rightarrow^+ t; \Delta \)

**Proof.** 1. Case 1. Right Async:

(a) Case \( \text{R} \Rightarrow^+ \)

In the case of the right asynchronous rule for abstraction introduction, the synthesis rule has the form:

\[
\frac{\Gamma; \Omega, x : A \vdash B \uparrow \Rightarrow^+ t; \Delta, x : A}{\Gamma; \Omega \vdash A \Rightarrow^+ \lambda x.t; \Delta \quad \text{R} \Rightarrow^+}
\]

By induction on the premise, we have that:

\[
(\Gamma, \Omega), x : A \vdash B \Rightarrow^+ t; \Delta, x : A \quad \text{(ih)}
\]

from case 1 of the lemma. From which, we can construct the following instantiation of the \( \text{R} \Rightarrow^+ \) synthesis rule in the non-focusing calculus:

\[
\frac{\Gamma, \Omega \vdash A \Rightarrow^+ \lambda x.t; \Delta}{\Gamma, \Omega \vdash A \Rightarrow^+ \lambda x.t; \Delta \quad \text{R} \Rightarrow^+}
\]
Case R↑⁺

In the case of the right asynchronous rule for transition to a left asynchronous judgement, the synthesis rule has the form:

\[
\frac{Γ; Ω \vdash C \Rightarrow t; Δ}{Γ; Ω \vdash C \Rightarrow t; Δ} \quad \text{R↑⁺}
\]

By induction on the first premise, we have that:

\[
Γ, Ω \vdash C \Rightarrow t; Δ
\]

from case 2 of the lemma.

2. Case 2. Left Async:

(a) Case L⊗⁺

In the case of the left asynchronous rule for pair elimination, the synthesis rule has the form:

\[
\frac{Γ, x_1 : A, x_2 : B \vdash C \Rightarrow t_2; Δ, x_1 : A, x_2 : B}{Γ, x_3 : A \otimes B \vdash C \Rightarrow t_2; Δ, x_3 : A \otimes B} \quad \text{L⊗⁺}
\]

By induction on the premise, we have that:

\[
(Γ, Ω), x_1 : A, x_2 : B \vdash C \Rightarrow t_2; Δ, x_1 : A, x_2 : B
\]

from case 2 of the lemma. From which, we can construct the following instantiation of the L⊗⁺ synthesis rule in the non-focusing calculus:

\[
(Γ, Ω), x_1 : A, x_2 : B \vdash C \Rightarrow t_2; Δ, x_1 : A, x_2 : B
\]

(b) Case L⊕⁺

In the case of the left asynchronous rule for sum elimination, the synthesis rule has the form:

\[
\frac{Γ; Ω \vdash C \Rightarrow t_1; Δ_1, x_2 : A}{Γ; Ω \vdash C \Rightarrow t_1; Δ_1, x_2 : A}
\]

By induction on the premises, we have that:

\[
(Γ, Ω), x_2 : A \vdash C \Rightarrow t_1; Δ_1, x_2 : A
\]

and

\[
(Γ, Ω), x_3 : B \vdash C \Rightarrow t_2; Δ_2, x_3 : B
\]

from case 2 of the lemma. From which, we can construct the following instantiation of the L⊕⁺ synthesis rule in the non-focusing calculus:

\[
(Γ, Ω), x_2 : A \vdash C \Rightarrow t_1; Δ_1, x_2 : A
\]

and

\[
(Γ, Ω), x_3 : B \vdash C \Rightarrow t_2; Δ_2, x_3 : B
\]

from case 2 of the lemma. From which, we can construct the following instantiation of the L⊕⁺ synthesis rule in the non-focusing calculus:

\[
(Γ, Ω), x_2 : A \vdash C \Rightarrow t_1; Δ_1, x_2 : A
\]

and

\[
(Γ, Ω), x_3 : B \vdash C \Rightarrow t_2; Δ_2, x_3 : B
\]

from case 2 of the lemma. From which, we can construct the following instantiation of the L⊕⁺ synthesis rule in the non-focusing calculus:

\[
(Γ, Ω), x_2 : A \vdash C \Rightarrow t_1; Δ_1, x_2 : A
\]

and

\[
(Γ, Ω), x_3 : B \vdash C \Rightarrow t_2; Δ_2, x_3 : B
\]
(c) Case \(L1^+\)
In the case of the left asynchronous rule for unit elimination, the synthesis rule has the form:

\[
\frac{\Gamma; \emptyset \vdash C \Rightarrow^+ t; \Delta}{\Gamma; x : 1 \vdash C \Rightarrow^+ \text{let } () = x \text{ in } t; \Delta, x : 1}
\]

By induction on the premise, we have that:

\[
\frac{\Gamma \vdash C \Rightarrow^+ t; \Delta}{(ih)}
\]

from case 2 of the lemma. From which, we can construct the following instantiation of the \(L1^+\) synthesis rule in the non-focusing calculus:

\[
\frac{\Gamma, x : 1 \vdash C \Rightarrow^+ \text{let } () = x \text{ in } t; \Delta, x : 1}{L1^+}
\]

(d) Case \(L\square^+\)
In the case of the left asynchronous rule for graded modality elimination, the synthesis rule has the form:

\[
\frac{\Gamma; \Omega, x_2 : [A]_r, \uparrow \vdash B \Rightarrow^+ t; \Delta \quad \text{if } x_2 : [A]_s \in \Delta \text{ then } s \sqsubseteq r \text{ else } 0 \sqsubseteq r}{\Gamma; \Omega, x_1 : \square_r A \vdash B \Rightarrow^+ \text{let } [x_2] = x_1 \text{ in } t; (\Delta \setminus x_2), x_1 : \square_r A}
\]

By induction on the first premise, we have that:

\[
\frac{(\Gamma, \Omega), x_2 : [A]_r \vdash B \Rightarrow^+ t; \Delta}{(ih)}
\]

from case 2 of the lemma. From which, we can construct the following instantiation of the \(L\square^+\) synthesis rule in the non-focusing calculus:

\[
\frac{(\Gamma, \Omega), x_1 : \square_r A \vdash B \Rightarrow^+ \text{let } [x_2] = x_1 \text{ in } t; (\Delta \setminus x_2), x_1 : \square_r A}{L\square^+}
\]

(e) Case \(\text{der}^+\)
In the case of the left asynchronous rule for dereliction, the synthesis rule has the form:

\[
\frac{\Gamma; x : [A]_s, y : A \uparrow \vdash B \Rightarrow^+ t; \Delta, y : A}{\Gamma; x : [A]_s \uparrow \vdash B \Rightarrow^+ [x/y] t; \Delta + x : [A]_1}
\]

By induction on the premise, we have that:

\[
\frac{\Gamma, x : [A]_s, y : A \vdash B \Rightarrow^+ t; \Delta, y : A}{(ih)}
\]

from case 2 of the lemma. From which, we can construct the following instantiation of the \(\text{der}^+\) synthesis rule in the non-focusing calculus:

\[
\frac{\Gamma, x : [A]_s \vdash B \Rightarrow^+ [x/y] t; \Delta + x : [A]_1}{\text{der}^+}
\]

48
(f) Case $L\hat{\downarrow}^+$
In the case of the left asynchronous rule for transitioning an assumption from the focusing context $\Omega$ to the non-focusing context $\Gamma$, the synthesis rule has the form:

$$
\frac{\Gamma, x : A ; \Omega \vdash C \Rightarrow^+ t; \Delta \quad \text{A not left async}}{\Gamma; \Omega, x : A \vdash C \Rightarrow^+ t; \Delta} \quad L\hat{\downarrow}^+
$$

By induction on the first premise, we have that:

$$
\Gamma, x : A, \Omega \vdash C \Rightarrow^+ t; \Delta \quad \text{(ih)}
$$

from case 2 of the lemma.

3. Case 3. Right Sync:

(a) Case $R\otimes^+$
In the case of the right synchronous rule for pair introduction, the synthesis rule has the form:

$$
\frac{\Gamma; \emptyset \vdash A \downarrow \Rightarrow^+ t_1; \Delta_1 \quad \Gamma; \emptyset \vdash B \downarrow \Rightarrow^+ t_2; \Delta_2}{\Gamma; \emptyset \vdash A \otimes B \downarrow \Rightarrow^+ \langle t_1, t_2 \rangle; \Delta_1 + \Delta_2} \quad R\otimes^+
$$

By induction on the premises, we have that:

$$
\Gamma \vdash A \Rightarrow^+ t_1; \Delta_1 \quad \text{(ih1)}
$$

$$
\Gamma \vdash B \Rightarrow^+ t_2; \Delta_2 \quad \text{(ih2)}
$$

from case 3 of the lemma. From which, we can construct the following instantiation of the $R\otimes^+$ synthesis rule in the non-focusing calculus:

$$
\frac{\Gamma \vdash A \Rightarrow^+ t_1; \Delta_1 \quad \Gamma \vdash B \Rightarrow^+ t_2; \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^+ \langle t_1, t_2 \rangle; \Delta_1 + \Delta_2} \quad R\otimes^+
$$

(b) Case $R\oplus^+$ and $R\oplus^+_2$
In the case of the right synchronous rules for sum introduction, the synthesis rules have the form:

$$
\frac{\Gamma; \emptyset \vdash A \downarrow \Rightarrow^+ t; \Delta}{\Gamma; \emptyset \vdash A \oplus B \downarrow \Rightarrow^+ \text{inl } t; \Delta} \quad R\oplus^+_1
$$

$$
\frac{\Gamma; \emptyset \vdash B \downarrow \Rightarrow^+ t; \Delta}{\Gamma; \emptyset \vdash A \oplus B \downarrow \Rightarrow^+ \text{inr } t; \Delta} \quad R\oplus^+_2
$$

By induction on the premises of the rules, we have that:

$$
\Gamma \vdash A \Rightarrow^+ t; \Delta \quad \text{(ih1)}
$$

$$
\Gamma \vdash B \Rightarrow^+ t; \Delta \quad \text{(ih2)}
$$

49
from case 3 of the lemma. From which, we can construct the following instantiations of the $R \oplus^+ t$ and $\text{addSumIntroRName}$ synthesis rules in the non-focusing calculus, respectively:

$$
\begin{align*}
\Gamma \vdash A \Rightarrow^+ t; \Delta & \quad \quad \quad R \oplus^+_1 \\
\Gamma \vdash A \oplus B \Rightarrow^+ \text{inl} t; \Delta & \quad \quad \quad R \oplus^+_2 \\
\end{align*}
$$

(c) Case $R1^+$

In the case of the right synchronous rule for unit introduction, the synthesis rule has the form:

$$
\Gamma; \emptyset \vdash 1 \Rightarrow^+ (); \emptyset \quad R1^+
$$

From which, we can construct the following instantiation of the $R1^+$ synthesis rule in the non-focusing calculus:

$$
\Gamma \vdash 1 \Rightarrow^+ (); \emptyset \quad R1^+
$$

(d) Case $R\Box^+$

In the case of the right synchronous rule for graded modality introduction, the synthesis rule has the form:

$$
\begin{align*}
\Gamma; \emptyset \vdash A \uparrow \Rightarrow^+ t; \Delta & \quad \quad \quad R\Box^+ \\
\Gamma; \emptyset \vdash \Box_r A \downarrow \Rightarrow^+ [t]; r \ast \Delta & \quad \quad \quad R\Box^+
\end{align*}
$$

By induction on the premise, we have that:

$$
\Gamma \vdash A \Rightarrow^+ t; \Delta \quad \text{(ih)}
$$

from case 1 of the lemma. From which, we can construct the following instantiation of the $R\Box^+$ synthesis rule in the non-focusing calculus:

$$
\begin{align*}
\Gamma \vdash A \Rightarrow^+ t; \Delta & \quad \quad \quad R\Box^+ \\
\Gamma \vdash \Box_r A \Rightarrow^+ [t]; r \ast \Delta & \quad \quad \quad R\Box^+
\end{align*}
$$

(e) Case $R\downarrow^+$

In the case of the right synchronous rule for transitioning back to an asynchronous judgement, the synthesis rule has the form:

$$
\begin{align*}
\Gamma; \emptyset \vdash A \uparrow \Rightarrow^+ t; \Delta & \quad \quad \quad R\downarrow^+ \\
\Gamma; \emptyset \vdash A \downarrow \Rightarrow^+ t; \Delta & \quad \quad \quad R\downarrow^+
\end{align*}
$$

By induction on the premise, we have that:

$$
\Gamma \vdash A \Rightarrow^+ t; \Delta \quad \text{(ih)}
$$

from case 1 of the lemma.
4. Case 4. Left Sync

(a) Case $L \rightarrow o^+$
In the case of the left synchronous rule for application, the synthesis rule has the form:

$$
\begin{array}{c}
\Gamma; x_2 : B \Downarrow C \Rightarrow t_1; \Delta_1, x_2 : B \\
\Gamma; \emptyset \vdash A \Downarrow \Rightarrow t_2; \Delta_2
\end{array}
\Rightarrow
\begin{array}{c}
\Gamma; x_1 : A \rightarrow B \Downarrow \Rightarrow C \Rightarrow t_1; \Delta_1, x_2 : B \\
\Gamma; x_1 : A \rightarrow B \Downarrow \Rightarrow C \Rightarrow t_2; \Delta_2
\end{array}
$$

By induction on the first premise, we have that:

$$
\Gamma, x_2 : B \vdash C \Rightarrow t_1; \Delta_1, x_2 : B \quad (\text{ih1})
$$

from case 4 of the lemma. By induction on the second premise, we have that:

$$
\Gamma \vdash A \Rightarrow t_2; \Delta_2 \quad (\text{ih2})
$$

from case 3 of the lemma. From which, we can construct the following instantiation of the $L \rightarrow o^+$ synthesis rule in the non-focusing calculus:

$$
\begin{array}{c}
\Gamma, x_2 : B \Downarrow C \Rightarrow t_1; \Delta_1, x_2 : B \\
\Gamma, x_2 : B \Downarrow C \Rightarrow t_2; \Delta_2
\end{array}
\Rightarrow
\begin{array}{c}
\Gamma, x_1 : A \rightarrow B \Downarrow \Rightarrow [((x_1 t_2)/x_2) t_1]; (\Delta_1 + \Delta_2), x_1 : A \rightarrow B \Rightarrow \frac{\frac{(x_1 t_2)/x_2) t_1; (\Delta_1 + \Delta_2), x_1 : A \rightarrow B}{\Gamma, x_1 : A \rightarrow B \Downarrow \Rightarrow C \Rightarrow t_1; \Delta_1, x_2 : B}}{\Gamma, x_1 : A \rightarrow B \Downarrow \Rightarrow C \Rightarrow t_2; \Delta_2}
\end{array}
$$

(b) Case $\text{LinVar}^+$
In the case of the left synchronous rule for linear variable synthesis, the synthesis rule has the form:

$$
\Gamma; x : A \vdash A \Rightarrow^+ \frac{\frac{x}{A}}{x : A} \quad \text{LinVar}^+
$$

From which, we can construct the following instantiation of the $\text{LinVar}^+$ in the non-focusing calculus:

$$
\Gamma; x : A \vdash A \Rightarrow^+ \frac{\frac{x}{A}}{x : A} \quad \text{LinVar}^+
$$

(c) Case $\text{GrVar}^+$
In the case of the left synchronous rule for graded variable synthesis, the synthesis rule has the form:

$$
\Gamma; x : [A]_r \vdash A \Rightarrow^+ \frac{x}{[A]_1} \quad \text{GrVar}^+
$$

From which, we can construct the following instantiation of the $\text{GrVar}^+$ synthesis rule in the non-focusing calculus:

$$
\Gamma; x : [A]_r \vdash A \Rightarrow^+ \frac{x}{[A]_1} \quad \text{GrVar}^+
$$
(d) Case \( L \downarrow \uparrow \)

In the case of the left synchronous rule for transitioning back to an asynchronous judgement, the synthesis rule has the form:

\[
\frac{\Gamma; x : A \uparrow \vdash C \Rightarrow^+ t; \Delta}{\Gamma; x : A \downarrow \vdash C \Rightarrow^+ t; \Delta} \text{ L}_{\downarrow \uparrow}^+
\]

By induction on the premise, we have that:

\( \Gamma, x : A \vdash C \Rightarrow^+ t; \Delta \)

from case 2 of the lemma.

5. Case 5. Focus Right: focusR\(^+\)

In the case of the focusing rule for transitioning from a left asynchronous judgement to a right synchronous judgement, the synthesis rule has the form:

\[
\frac{\Gamma; \emptyset \vdash C \Rightarrow^+ t; \Delta}{\Gamma; \emptyset \vdash C \Rightarrow^+ t; \Delta} \text{ focusR}^+
\]

By induction on the first premise, we have that:

\( \Gamma \vdash C \Rightarrow^+ t; \Delta \)

from case 2 of the lemma.

6. Case 6. Focus Left: focusL\(^+\)

In the case of the focusing rule for transitioning from a left asynchronous judgement to a left synchronous judgement, the synthesis rule has the form:

\[
\frac{\Gamma; x : A \downarrow \vdash C \Rightarrow^+ t; \Delta}{\Gamma, x : A \vdash C \Rightarrow^+ t; \Delta} \text{ focusL}^+
\]

By induction on the first premise, we have that:

\( \Gamma, x : A \vdash C \Rightarrow^+ t; \Delta \)

from case 2 of the lemma.

Lemma 11 (Soundness of focusing for additive pruning synthesis). For all contexts \( \Gamma, \Omega \) and types \( A \) then:

1. Right Async : \( \Gamma; \Omega \vdash A \uparrow \Rightarrow^+ t; \Delta \) \( \Rightarrow \) \( \Gamma, \Omega \vdash A \Rightarrow^+ t; \Delta \)
2. Left Async : \( \Gamma; \Omega \uparrow \vdash C \Rightarrow^+ t; \Delta \) \( \Rightarrow \) \( \Gamma, \Omega \vdash C \Rightarrow^+ t; \Delta \)
3. Right Sync : \( \Gamma; \emptyset \vdash A \downarrow \Rightarrow^+ t; \Delta \) \( \Rightarrow \) \( \Gamma \vdash A \Rightarrow^+ t; \Delta \)
4. Left Sync : \( \Gamma; x : A \downarrow \vdash C \Rightarrow^+ t; \Delta \) \( \Rightarrow \) \( \Gamma, x : A \vdash C \Rightarrow^+ t; \Delta \)
5. Focus Right : \( \Gamma; \Omega \uparrow \vdash C \Rightarrow^+ t; \Delta \) \( \Rightarrow \) \( \Gamma \vdash C \Rightarrow^+ t; \Delta \)
6. Focus Left : \( \Gamma, x : A; \Omega \uparrow \vdash C \Rightarrow^+ t; \Delta \) \( \Rightarrow \) \( \Gamma \vdash C \Rightarrow^+ t; \Delta \)
Proof. 1. Case: 1. Right Async: The proofs for right asynchronous rules are equivalent to those of lemma (10).

2. Case 2. Left Async: The proofs for left asynchronous rules are equivalent to those of lemma (10).

3. Case 3. Right Sync: The proofs for right synchronous rules are equivalent to those of lemma (10), except for the case of the $R' \otimes^+$ rule:

(a) Case $R' \otimes^+$

In the case of the right synchronous rule for pair introduction, the synthesis rule has the form:

$$
\frac{\Gamma; \emptyset \vdash A \Rightarrow^+ t_1; \Delta_1 \quad \Gamma - \Delta_1; \emptyset \vdash B \Rightarrow^+ t_2; \Delta_2}{\Gamma; \emptyset \vdash A \otimes B \Rightarrow^+ \langle t_1, t_2 \rangle; \Delta_1 + \Delta_2}
$$

By induction on the premises, we have that:

$$
\begin{align*}
\Gamma \vdash A \Rightarrow^+ t_1; \Delta_1 & \quad \text{(ih1)} \\
\Gamma - \Delta_1 \vdash B \Rightarrow^+ t_2; \Delta_2 & \quad \text{(ih2)}
\end{align*}
$$

from case 3 of the lemma. From which, we can construct the following instantiation of the $R' \otimes^+$ synthesis rule in the non-focusing calculus:

$$
\frac{\Gamma \vdash A \Rightarrow^+ t_1; \Delta_1 \quad \Gamma - \Delta_1 \vdash B \Rightarrow^+ t_2; \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^+ \langle t_1, t_2 \rangle; \Delta_1 + \Delta_2}
$$

4. Case 4. Left Sync: The proofs for left synchronous rules are equivalent to those of lemma (10), except for the case of the $L' \otimes^+$ rule:

(a) Case $L' \otimes^+$

In the case of the left synchronous rule for application, the synthesis rule has the form:

$$
\frac{\Gamma; x_2 : B \vdash C \Rightarrow^+ t_1; \Delta_1, x_2 : B \quad \Gamma - \Delta_1; \emptyset \vdash A \Rightarrow^+ t_2; \Delta_2}{\Gamma; x_1 : A \rightarrow B \vdash C \Rightarrow^+ [\langle x_1 t_2 \rangle t_1; (\Delta_1 + \Delta_2), x_1 : A \rightarrow B] L' \rightarrow^+}
$$

By induction on the first premise, we have that:

$$
\begin{align*}
\Gamma, x_2 : B \vdash C \Rightarrow^+ t_1; \Delta_1, x_2 : B & \quad \text{(ih1)} \\
\Gamma \vdash A \Rightarrow^+ t_2; \Delta_2 & \quad \text{(ih2)}
\end{align*}
$$

from case 4 of the lemma. By induction on the second premise, we have that:

$$
\begin{align*}
\Gamma \vdash A \Rightarrow^+ t_2; \Delta_2
\end{align*}
$$

from case 3 of the lemma. From which, we can construct the following instantiation of the $L' \rightarrow^+$ synthesis rule in the non-focusing calculus:

$$
\frac{\Gamma, x_2 : B \vdash C \Rightarrow^+ t_1; \Delta_1, x_2 : B \quad \Gamma - \Delta_1 \vdash A \Rightarrow^+ t_2; \Delta_2}{\Gamma; x_1 : A \rightarrow B \vdash C \Rightarrow^+ [\langle x_1 t_2 \rangle t_1; (\Delta_1 + \Delta_2), x_1 : A \rightarrow B] L' \rightarrow^+}
$$

53
5. Case 5. Right Focus: focusR\(^+\) - The proof for right focusing rule is equivalent to that of lemma \([10]\).

6. Case 6. Left Focus: focusL\(^+\) - The proof for left focusing rule is equivalent to that of lemma \([10]\).
Algorithm Selection for Dynamic Symbolic Execution: A Preliminary Study

Roberto Amadini¹, Graeme Gange³, Peter Schachte², Harald Søndergaard², and Peter J. Stuckey³

¹ University of Bologna, Bologna, Italy
² University of Melbourne, Victoria, Australia
³ Monash University, Melbourne, Victoria, Australia

Abstract. Given a portfolio of algorithms, the goal of Algorithm Selection (AS) is to select the best algorithm(s) for a new, unseen problem instance. Dynamic Symbolic Execution (DSE) brings together concrete and symbolic execution to maximise the program coverage. DSE uses a constraint solver to solve the path conditions and generate new inputs to explore. In this paper we join these lines of research by introducing a model that combines DSE and AS approaches. The proposed AS/DSE model is a generic and flexible framework enabling the DSE engine to solve the path conditions it collects with a portfolio of different solvers, by exploiting and extending the well-known AS techniques that have been developed over the last decade. In this way, one can increase the coverage and sometimes even outperform the aggregate coverage achievable by running simultaneously all the solvers of the portfolio.

Keywords: Software Verification, Dynamic Symbolic Execution, Algorithm Selection, Constraint solving, Portfolio solving

1 Introduction

The Algorithm selection (AS) problem was formalised by Rice in 1976 [27]. In a nutshell, given a set of algorithms \( A \) and a problem instance \( i \), AS aims to select the best algorithm in \( A \) to solve \( i \) according to a given performance metric \( m \) [33]. AS approaches are also known as portfolio approaches, where “solver” is used as a synonym of algorithm, “portfolio” indicates a subset of solvers of \( A \), and the solver selection is performed on a per-instance basis.

Algorithm selection typically uses machine learning techniques (e.g., decision trees or \( k \)-nearest neighbours) and it is not limited to the choice of a single solver: a portfolio approach can first select a number of different solvers, and then schedule their (sequential or parallel) execution [20].

The solver selection is typically performed by extracting a number of features from each problem. Features are numerical attributes characterizing a given problem instance (e.g., the number of variables or constraints).
Over the last years a large number of effective AS approaches have been proposed in different fields, including SAT solving [36, 24], constraint programming (CP) [3, 12], answer-set programming (ASP) [11], and planning [35].

Dynamic symbolic execution (DSE) [9, 22] also known as concolic execution/testing, or directed automated random testing (DART) is a software verification technique combining the concrete (or dynamic) execution of a program together with its symbolic execution [17]. DSE first collects all the constraints involving symbolic variables (the so called path conditions) encountered during the dynamic execution at each conditional statement. Then, a constraint solver (or theorem prover) is used to generate alternative execution paths by systematically negating the path conditions. This process is repeated until all the feasible paths are covered, or a resource limit (e.g., a time limit or a maximum number of iterations) is reached.

A possible bottleneck of DSE is the solver used for solving path conditions. Indeed, despite the significant progress made by constraint solvers over the last years, it is still hard for a single, arbitrarily efficient solver to properly encode and solve the great variety of path conditions arising from the DSE of different programs. This is partly due to the complexity of precisely encoding the semantics of modern programming languages. Unfortunately, depending on the shape of the input program, failing to solve even just one path condition could result in a significant loss in terms of code coverage.

In this work we propose the AS/DSE framework, a generic model that aims to get the best of the AS and DSE worlds. The goal is to mitigate the issues of single-solver DSE approaches with a portfolio of different solvers. This can be beneficial in terms of robustness (if a solver fails on a path condition, the overall DSE is not compromised), runtime minimisation and code coverage maximisation.

To our knowledge, the few approaches proposed so far [26] merely run in parallel different DSE engines — each of them running a different solver of the portfolio. Here we take a step forward by proposing a model that can be arbitrarily instantiated depending on the available solvers, the input language, the program to analyse, the execution environment, and so on. In particular, we show that a AS/DSE model may even outperform the aggregate coverage achievable by running independently all the solvers of the portfolio.

As a proof-of-concept, we implemented and evaluated two basic portfolio approaches on top of Aratha [1], a tool for JavaScript DSE enabling the use of both SMT and CP solvers for solving path conditions. Preliminary results are encouraging and show the potential of combining AS with DSE. For each analysed program the coverage achieved by portfolio approaches is never worse than the one achievable with a single solver and, in particular, one of the approaches is able to outperform the coverage reachable by running simultaneously all the solvers of the portfolio.

Paper structure. In Section 2 we give the basic notions about AS and DSE. In Section 3 we describe the AS/DSE model, while in Section 4 we show the results of the preliminary investigation we performed. In Section 5 we report the related literature before concluding in Section 6.
2 Preliminaries

We start with some background notions related to algorithm selection and dynamic symbolic execution.

2.1 Algorithm Selection

The main ingredients of an AS scenario are: (i) the algorithms to be selected, (ii) the problem instances on which algorithms are applied, (iii) the performance metric used to evaluate an algorithm on a given problem. More formally, an AS scenario is a triple \((\mathcal{I}, \mathcal{A}, m)\) where \(\mathcal{I}\) is a set of instances, \(\mathcal{A}\) is a set (or portfolio) of algorithms (or solvers) with \(|\mathcal{A}| > 1\), and \(m : \mathcal{I} \times \mathcal{A} \to \mathbb{R}\) is a performance metric that w.l.o.g. we can assume to be minimized.

An algorithm selector (or portfolio selector) aims to return the best algorithm, according to the performance metric, for a given instance. Formally, given an AS scenario \((\mathcal{I}, \mathcal{A}, m)\), a selector \(\xi\) is a total mapping \(\xi : \mathcal{I} \to \mathcal{A}\). The AS problem consists in finding a selector \(\xi\) minimizing \(\sum_{i \in \mathcal{I}} m(i, \xi(i))\).

Note that \(m\) is a partial function, i.e., we do not know a priori the value of \(m(i, A)\) for each possible \(i \in \mathcal{I}, A \in \mathcal{A}\) (otherwise the AS problem would be trivial). This means that an AS selector has to estimate \(m(i, A)\) when it is unknown, and predict the best algorithm(s) for \(i\). For each scenario we can define the virtual best solver (VBS) baseline, i.e., an “oracle selector” always choosing the algorithm \(\text{VBS}(i)\) such that \(m(i, \text{VBS}(i))\) is minimal for each \(i \in \mathcal{I}\). Hence, \(m(i, \text{VBS}(i)) \leq m(i, \xi(i))\) for each selector \(\xi\) and \(i \in \mathcal{I}, A \in \mathcal{A}\).

The above schema can be however extended by enabling the scheduling of \(k > 1\) algorithms \([(A_1, t_1), \ldots, (A_k, t_k)]\) for a given problem \(i\), where \(t_j\) is the time slot assigned to each \(A_j\) for \(j = 1, \ldots, k\). Note that, because the instances of \(\mathcal{I}\) might be too hard to solve, often a timeout \(T\) is used: \(m(i, A) \leq T\) for each \(i \in \mathcal{I}, A \in \mathcal{A}\). So, if \(A_1, \ldots, A_k\) are scheduled for time \(t_1, \ldots, t_k\) then \(\sum_{j=1}^{k} t_j = T\).

An advantage of scheduling \(k\) solvers is that algorithm \(A_j\) can use the information computed by \(A_{j'}\), with \(j' < j\), to improve its performance. For example, consider an optimization problem \(i\) where the best objective value \(v\) found by solver \(A_{j'}\) can be exploited by another solver \(A_j\) to narrow its search space (i.e., \(A_j\) solves a modified problem \(i_v\) where value \(v\) is “injected” to \(i\) [6]). It is important to note that such a collaborative approach may allow a portfolio solver to outperform the VBS, i.e., it might be \(m(i_v, A_j) < m(i, \text{VBS}(i))\).

Finally, note that AS scenarios usually characterize each instance \(i \in \mathcal{I}\) with a corresponding feature vector \(F(i) \in \mathbb{R}^n\), and the selection of the best algorithm \(A\) for \(i\) is actually performed according to \(F(i)\). For example, if \(i\) is a constraint satisfaction problem then \(F(i)\) may include the number of variables or constraints of \(i\). If \(i\) is an optimization problem, we may also want to capture numeric information about the objective function of \(i\) (e.g., its lower and upper bounds).
There is an extensive literature about selecting the presumably best set of features for a given instance. In particular, the process of refining a feature vector $F(i) \in \mathbb{R}^n$ by deriving a smaller vector $F'(i) \in \mathbb{R}^m$ with $m \leq n$ is known as feature selection. The purpose of feature selection is to simplify the prediction model, reducing the training and feature extraction costs, and improving the prediction accuracy.

### 2.2 Dynamic Symbolic Execution

DSE is a combination of concrete and symbolic execution.

Symbolic execution is a whole-program analysis technique that has its roots in the 1970s [17]. The idea is that, during execution, some variables take on symbolic values, maintained as expressions involving unknown input values. A symbolic interpreter explores the possible program paths that concrete executions could take, by reasoning about the conditions under which execution will branch this way or that. More precisely, a symbolic state $(\sigma, \Gamma)$ is maintained, consisting of a mapping $\sigma$ that associates variables with expressions, and a path constraint $\Gamma$. The latter is a conjunction of primitive constraints involving symbolic names for inputs. It effectively determines the set of input values that would take concrete execution along the current execution path; that is, a path is feasible if and only if the corresponding constraints are satisfiable.

The test for satisfiability, and the generation of a witness (a solution) in the affirmative case, is delegated to a constraint solver (or theorem prover). The symbolic state evolves according to simple rules: (1) an assignment $x = e$ updates the symbolic state $(\sigma, \Gamma)$ to $(\sigma[x \mapsto \sigma(e)], \Gamma)$, and (2) for a conditional if $(e)$ s1 else s2, two path constraints are generated, namely $\Gamma \land \sigma(e)$ and $\Gamma \land \neg \sigma(e)$; if $\Gamma \land \sigma(e)$ is satisfiable, s1 is symbolically executed under the new path constraint, and similarly for $\Gamma \land \neg \sigma(e)$ and s2.

Two significant limitations of symbolic execution are: (i) the whole program—including libraries—is often not available to the interpreter; (ii) the underlying constraint solver is often not expressive and efficient enough to handle the generated path conditions.

DSE combines symbolic execution with concrete execution, by performing the symbolic execution along with concrete execution of the given program. The motivation for this is to sidestep the two limitations mentioned above. Having the concrete runtime state allows the tool to replace symbolic variables by concrete values when faced by external function calls, and also to simplify difficult constraints. This enables progress of the symbolic execution, albeit at the sacrifice of completeness.

So DSE needs to be seeded with concrete values for symbolic variables. It can then perform a sequence of well-chosen concrete/symbolic executions (aimed at maximizing code coverage), by taking a recently generated path constraint, negating one of its conjuncts, asking a constraint solver whether the result is satisfiable, and, if so, to provide a model, which can serve as a new seed.

Consider, for example, the snippet of pseudo-code in Fig. 1. Suppose the initial input is $x \leftarrow 0$. The concrete execution of $f(x)$ will print 'bar', and
1: function f(x)
2:     if x < 0 then
3:         print(’foo’)
4:     else if x ≥ 5 then
5:         print(’fee’)
6:     else
7:         print(’bar’)

Fig. 1: Pseudo-code example. The set of inputs \{x ← −1, x ← 0, x ← 5\} covers all the lines of function f.

the DSE engine will track the corresponding path conditions: ¬(x < 0) and ¬(x ≥ 5). After that, one path condition will be negated, let us say ¬(x ≥ 5), and a constraint solver will solve ¬(x < 0) ∧ x ≥ 5. A computed solution (say, x = 5) will be the input of the next concrete execution, that will print ’fee’. This process is repeated until all the feasible paths are covered, or a pre-set resource limit (usually a time limit or a maximum number of iterations) is reached.

DSE can mitigate the aforementioned symbolic execution issues by: (i) directly invoking unavailable functions (a complete symbolic interpreter is not required); (ii) ignoring or approximating unsupported constraints. This implies that, in general, DSE cannot guarantee full coverage. In most applications, such as test data generation, this is acceptable, provided a “good enough” coverage is achieved in a reasonable time.

3 The AS/DSE Model

The purpose of this section is to map out the considerable space for algorithm selection in DSE. We list an array of opportunities for “selection” that may provide avenues to better DSE tools. Presently there is very limited support for this, as existing tools tend to be tightly coupled with specific solvers. In Section 4 we report on whatever experiments we have been able to run, based on existing technology.

The AS/DSE model is depicted in Fig. 2. The upper dashed box refers to the “classical” DSE framework. The first step is to annotate the input program P with \( n > 0 \) symbolic variables \( x_1, \ldots, x_n \) of interest, i.e., with meta-variables using symbolic values to represent input values. This can be performed manually or automatically (e.g., by means of taint analysis [29]). The result is a symbolic program \( P' \) containing both symbolic and “concrete” (i.e., non-symbolic) variables.

Given initial concrete values \( v_0 = (v_{0,1}, \ldots, v_{0,n}) \) we first execute \( P'[v_0] \), i.e., the program \( P' \) where value \( v_{0,i} \) is assigned to variable \( x_i \) for \( i = 1, 2, \ldots, n \). Concurrently, a symbolic engine collects the path conditions \( \Gamma_i = (C_1, \ldots, C_m) \) encountered during the concrete execution as explained in Section 2. At this stage, for generating the next input, the last path condition \( C_k \) not already
negated is flipped (notice that the choice of $C_k$ can be generalised to the $j$-th path condition not already negated, with $j \in \{1, \ldots, k\}$). If all the conditions have been already negated, the DSE terminates.

Let $\Gamma'_0 = (C_1, \ldots, C_{k-1}, \neg C_k)$. In the single-solver DSE a solver $S$ is now used to solve $\bigwedge_{C \in \Gamma'_0} C$. If $S$ returns a solution $\mathbf{v}_1 = (v_{1,1}, \ldots, v_{1,n})$ then we execute $P'[\mathbf{v}_1]$ to get new path conditions $\Gamma_1$; otherwise, we repeat the procedure by negating the last condition of $\Gamma'_0$ not already negated until either we find a solution $\mathbf{v}_1$ or all the conditions are negated. By iterating this process until the DSE terminates (or a given threshold is reached, e.g., a timeout or a maximum number of DSE iterations) we get a set of inputs $\{\mathbf{v}_1, \mathbf{v}_2, \ldots\}$ that ideally covers all the execution paths of $P$.

The AS/DSE model extends the single-solver model by using a portfolio of solvers $S = \{S_1, \ldots, S_p\}$. In Fig. 2, the lower dashed box shows how AS is plugged into the DSE framework. To solve a given tuple of path conditions $\Gamma$ with $S$, we define three different (yet interoperable) phases: (i) the solver selection phase, (ii) the solver execution phase, (iii) the solution selection phase.

### 3.1 Solver Selection

The first stage is selecting the solver(s) from the portfolio $S$. We are not aware of portfolio approaches for DSE that actually perform a discriminating solver selection, that is, selection on a per-instance basis. The only alternative is to run all the solvers of $S$ (possibly in parallel). This is straightforward and might work if $S$ is small but it is impractical if $S$ contains (many) more solvers than available cores (e.g., the literature presents portfolios with more than 20 solvers). Note that we can get different solvers by tuning the parameters of the same solver. Moreover, the synchronisation issues due to the simultaneous execution of too many solvers may significantly slow down the performance, especially when the running solvers have to share information.
Given the growing number of solvers based on different technologies (e.g., SAT/SMT, CP, MIP solvers), it makes sense to have an heterogeneous portfolio together with a proper solver selection heuristic $H_{sel}$ returning a non-empty subset $S' \subseteq S$ of the supposed best solver(s) of the portfolio for solving $\Gamma$. Note that the computational cost of solver selection is expected to be negligible.

The best portfolio approaches typically perform the solver prediction by extracting a set of features from the problem to solve, i.e., by computing a set of numerical attributes characterising that problem. Once features are computed, machine learning techniques can be used to determine the candidate solvers. Clearly, this process needs a proper feature extractor (e.g., [2]) and, in case of supervised learning, a dataset of known instances for which we know the performance of all the solvers of $S$. In this case, cross-validation techniques are often used to split the dataset into a training set (used to build a prediction model) and a test set (to validate that prediction model).

In our model, feature extraction can happen at different levels depending on when the solver selection is actually performed. We distinguish between three levels of solver selection: static, dynamic and hybrid.

**Static solver selection** is an “offline” AS procedure where the solvers are selected **eagerly** according to the input program $P$, or the symbolic program $P'$, regardless of the path conditions generated while analyzing $P$.

The advantages of this approach are its simplicity and efficiency: there is no need to modify the internals of the DSE engine, and the solver selection is performed only once per program: we do not need to collect the path conditions to build a training set. Static solver selection is suitable, e.g., when the size of the portfolio $S$ is much bigger than the number of available cores $c$: in this case one can select $c$ solvers from $S$ and run all of them in parallel.

The features extracted for static solver selection depends on the input program $P$ and/or the symbolic program $P'$ (depending of whether we want to take into account also the symbolic variables of $P'$). These features can be language-independent (e.g., the number of loops or symbolic variables) or bound to a specific language (e.g., the number of property accesses for JavaScript objects).

Note that, apart from the “syntactic” features extracted from the source code of the program, we can also have **probing features** derived from its actual execution. For example, we could execute $P'[\mathbf{v}]$, where $\mathbf{v}$ is a tuple of concrete values, and run the solvers for a short time on the corresponding path conditions to track their behaviour.

Unfortunately, a purely static solver selection lacks flexibility and is unable to exploit **marginal contributions** of different solvers. It risks excluding good solvers from $S$ only on the basis of the shape of $P$. However, a solver that behaves poorly on average might well be turn out to be highly effective for smaller classes of specific problems, on which other solvers struggle.

**Dynamic solver selection** refers to the “online” selection of solvers according to the path conditions to be solved. While the static solver selection occurs
just once for each program, dynamic solver selection is performed for each collected path condition. Because of its high frequency, it is essential that dynamic selection has a low computational cost.

The features extracted from each path condition $\Gamma$ are, e.g., statistics over the number and the type of the variables and constraints of $\Gamma$. One can also compute probing features by running one or more solvers of $\mathcal{S}$ on $\Gamma$. For example, one can retrieve the number of failures or the depth of the search tree after running a solver $S \in \mathcal{S}$ for a short time (e.g., 2 seconds).

Dynamic solver selection delivers flexibility and is, potentially, far more effective than static selection. What is less straightforward here is how to extract the features and train the model. The feature extraction has to be integrated into the DSE engine, while for static solver selection it can be performed “externally” without any modification to the DSE engine. Moreover, the training set can be very big because for each program a large number of (often similar) path conditions can be collected: we need some criteria to select “good representatives” among all the path conditions.

**Hybrid solver selection** combines, as the name suggests, static and dynamic selection. We can use static selection as a pre-processing step where, especially when we have a high number of solvers, we can reduce the original portfolio $\mathcal{S}$ into a smaller portfolio $\mathcal{S}'$. One may also use static selection to decide a proper parameter configuration for solvers that have a high number of parameters to be tuned. Then, dynamic selection is used on $\mathcal{S}'$ to select the presumably best solver(s) $S_r \subseteq \mathcal{S}'$ for each collected path conditions $\Gamma$. At this stage one can also reuse some of the features extracted in the static selection phase.

Finally, note that one can apply AS not only to select the best solver(s), but also to decide the best encoding for a given $\Gamma$. For example, assume $\mathcal{S} = S_{\text{sat}} \cup S_{\text{smt}} \cup S_{\text{cp}}$ where $S_{\text{sat}}$ are SAT solvers, $S_{\text{smt}}$ are SMT solvers, and $S_{\text{cp}}$ are CP solvers. Instead of directly selecting the best solver(s) in $\mathcal{S}$ for $\Gamma$, one might think to first choose the best encoding for $\Gamma$, i.e., whether it is better to convert $\Gamma$ into a SAT, a SMT or a CP problem. This actually means performing AS to first choose $S' \in \{S_{\text{sat}}, S_{\text{smt}}, S_{\text{cp}}\}$ and then choose solver(s) $S'' \subseteq S'$. An example of this hierarchical approach is provided by Hurley et al. [12] who use it to decide whether or not to encode a given CP problem to SAT before deciding the best solver for that problem.

### 3.2 Solver Execution

Let $\mathcal{S}' = \{S_1, \ldots, S_k\}$ be the selected solver(s) returned by $\mathcal{H}_{\text{sel}}$. Because we typically solve hard combinatorial problems (where a solver either solves a problem in a short time or it cannot solve it in a reasonable time) it is often desirable to select $k > 1$ solvers. In this case we have to use a proper solver execution heuristic $\mathcal{H}_{\text{exe}}$ to schedule the execution of each solver of $\mathcal{S}'$. The $\mathcal{H}_{\text{exe}}$ heuristic decides the running mode of the selected solvers, e.g.:

- how to run the solvers of $\mathcal{S}'$ (sequentially, concurrently or both)
1: if $x \geq 10$ then
2:   Stmt$_1$
3: if $C(x)$ then
4:   Stmt$_2$

Fig. 3: Example of program where $C$ is unsupported.

- how much time is allocated to each solver (typically a solving timeout is set)
- the execution order of each solver
- if and how to exchange information between them (e.g., nogoods or SMT queries)
- the configuration of their parameters
- when to stop all the solvers of $S'$.

Let us focus on the last point, which may appear counter-intuitive. Indeed, it might look more reasonable to just stop as soon as a solver finds a solution. However, waiting until a number $1 < j \leq k$ of solvers terminate can also be beneficial. In this case we sacrifice the runtime minimisation to possibly have $j > 1$ distinct solutions for $\Gamma$. Clearly, $\mathcal{H}_{exe}$ can also force the same solver to produce more than one solution.

Having different solutions $v_1, \ldots, v_j$ for the same path condition can be advantageous in terms of code coverage maximisation because often solvers are forced to over-approximate the path conditions, owing to unsupported constraints or unknown program functions. Considering $v_1, \ldots, v_j$ is somehow an “educated fuzzing” where we try distinct yet related inputs potentially leading to different program paths. In other terms, we might try to offset the “incompleteness” of solvers with the diversity of their solutions.

### 3.3 Solution Selection

Let us suppose that the solver execution phase returns $j > 1$ distinct solutions $v_1, \ldots, v_j$. In this case, we use a solution selection heuristic $\mathcal{H}_{sol}$ to decide which $v_i$ will be the input of the next DSE iteration. This phase is important because it enables us to rank $v_1, \ldots, v_j$ according to a given criteria.

For example, we may give low priority to solutions containing “default” values (e.g., 0 or the empty string) or rank solutions according to their type (note that, especially for weakly-typed languages such as JavaScript or Python, we may not have any information about the actual type of a symbolic variable).

As mentioned, it might be that the conjuncts in $\Gamma$ get relaxed, resulting in new path conditions $\Gamma'$, because the available solvers are not expressive or efficient enough to cope with the constraints of $\Gamma$. In this case, a solution of $\Gamma'$ is not necessarily a solution of $\Gamma$, so having different solutions for $\Gamma'$ may increase the probability of finding a solution for $\Gamma$ too (or for other path conditions).

For instance, consider the pseudo-code in Fig. 3 where we assume that $x$ is a symbolic variable and the condition $C$ is not supported by any solver (e.g., it
can be an unknown third-party function or a difficult mathematical function). If we start the DSE with \( x \leftarrow 0 \), then we collect path condition \( \neg(x \geq 10) \), we flip it and we solve \( x \geq 10 \). Let us suppose that we stop as soon as a solver returns a solution, say \( x = 10 \). If \( C(10) \) evaluates to false, then the next collected path conditions are \( x \geq 10 \) and \( \neg C(x) \). Once flipped \( \neg C(x) \), solvers should solve \( x \geq 10 \land C(x) \) but they cannot because they do not support \( C \). A possible way to mitigate this issue is to generate different solutions for \( x \geq 10 \) and try them to see if we can increase the code coverage (e.g., it might be that \( C(11) \) evaluates to true).

Another nice aspect is that we can use the solutions as nogoods: if at a given DSE iteration we get solution \( v = (v_1, \ldots, v_n) \), then we can add the constraint \( x_1 \neq v_1 \lor \cdots \lor x_n \neq v_n \) to the path conditions of all future DSE iterations in order to narrow the search and avoid exploring already visited paths. As we shall see in Section 4, this approach can improve the code coverage in practice.

### 3.4 Aggregate Coverage

The well-known benefits of AS are the average runtime minimisation and the maximization of the number of problems solved. However, in the context of DSE we have a further advantage. Let \( \text{COV}_S(P) \) be the set of statements (or lines) in \( P \) covered using the portfolio \( S \). Then, significantly, the coverage \( \text{COV}_{\{S_1, \ldots, S_p\}}(P) \) can be greater than the sum of its parts, that is, we may well have:

\[
\text{COV}_{\{S_1, \ldots, S_p\}}(P) \supset \text{COV}_{S_1}(P) \cup \cdots \cup \text{COV}_{S_p}(P).
\]

Note that \( \bigcup_{S \in S} \text{COV}_S(P) \) is exactly the coverage achievable by running \( p \) independent DSE analysis, each of which with a different solver \( S \in S \). In the following, we will refer to \( \bigcup_{S \in S} \text{COV}_S(P) \) as the aggregate coverage for portfolio \( S \) on program \( P \).

Consider the example in Fig. 4a where \( x \) is a symbolic variable and both conditions \( C_1(x) \) and \( C_2(x) \) are feasible. Let us suppose that \( S_1 \) can solve \( C_1(x) \) but cannot solve \( C_2(x) \), while \( S_2 \) can solve \( C_2(x) \) but not \( C_1(x) \); in this case with \( S = \{S_1, S_2\} \) we are able to cover both the statements \( \text{Stmt}_1 \) and \( \text{Stmt}_2 \), while with either \( S_1 \) or \( S_2 \) we can only cover either \( \text{Stmt}_1 \) or \( \text{Stmt}_2 \) respectively. For this example, a static approach running \( S_1 \) and \( S_2 \) independently is enough to reach the maximum coverage. Even if \( \text{COV}_S(P) \supset \text{COV}_{S_1}(P), \text{COV}_{S_2}(P) \), we cannot, in this case, improve on the aggregate coverage: \( \text{COV}_S(P) = \text{COV}_{S_1}(P) \cup \text{COV}_{S_2}(P) \) in the best case scenario.

It is important to understand the difference between aggregate coverage and the virtual best solver (VBS) of the portfolio. By definition, the VBS selects the best algorithm according to a given performance metric for every problem instance. For static solver selection we define the performance metric as: “maximize the coverage for a given program, breaking ties with minimum runtime”. So, for the example discussed above (Fig. 4a), the VBS is either \( S_1 \) or \( S_2 \) depending on which one is faster in solving \( C_1 \) or \( C_2 \) respectively. However, there is no case where the VBS is able to cover both the statements \( \text{Stmt}_1 \) and \( \text{Stmt}_2 \). In general, for static solver selection the coverage of the VBS is always less than,
1: \textbf{if } C_1(x) \textbf{ then} \quad \textbf{1: if } C_1(x) \textbf{ then}
2: \quad \textbf{Stmt}_1 \quad \textbf{2: } \textbf{Stmt}_1
3: \quad \textbf{if } C_2(x) \textbf{ then} \quad \textbf{3: if } C_2(x) \textbf{ then}
4: \quad \textbf{Stmt}_2 \quad \textbf{4: } \textbf{Stmt}_2

(a) Non-nested 'if' statements. \quad (b) Nested 'if' statements.

or equal to, the aggregate coverage. Conversely, as seen in the example above, a static scheduling (or a parallel execution) of solvers can yield better coverage than the VBS.

Fig. 4b shows a trickier example where we also assume that $C_1(x) \land C_2(x)$ can be solved by $S_2$ but not by $S_1$. As above, $S_1$ can only cover $\text{Stmt}_1$ while $S_2$ cannot solve $C_1(x)$ and thus will not cover neither $\text{Stmt}_1$ nor $\text{Stmt}_2$. However, unlike the case of Fig. 4a, independently running two different DSE engines using $S_1$ and $S_2$ respectively yields an insufficient coverage because none of them is able to reach line 4 and thus $\text{Stmt}_2$ will not be covered.

Because $\text{COV}_{S_1}(P) = \{\text{Stmt}_1\}$ and $\text{COV}_{S_2}(P) = \emptyset$, the aggregate coverage of $S$ will only cover $\text{Stmt}_1$. However, this does not mean that $\text{Stmt}_2$ cannot be covered using the solvers of $S$. Indeed, with a proper dynamic solver selection, we would be able to reach first $\text{Stmt}_1$ (thanks to $S_1$, solving $C_1(x)$) and then $\text{Stmt}_2$ (thanks to $S_2$, solving $C_1(x) \land C_2(x)$), thus outperforming the aggregate coverage.

Clearly this approach is only possible with an integrated AS/DSE implementation able to select the solvers on a path condition basis.

Following the standard definition, the VBS for dynamic solver selection is the solver that solves a given path condition in the shortest time. In this case, the coverage achieved with the VBS will never be worse than the aggregate coverage.

4 A Preliminary Evaluation

We have experimented with a portfolio approach to DSE, to the extent that existing DSE tools allow this. We have based experiments on Aratha [1], a tool for the DSE of JavaScript, because uniquely, Aratha can use both SMT solvers (viz. Z3 [25] and CVC4 [19]) and CP solvers (viz. G-Strings [5]). With this, we have implemented two static approaches, to explore whether portfolio solving turns out to improve on single-solver DSE in practice, and if so, by how much.

The first approach, which we call Aratha+, runs the DSE with G-Strings first, then with Z3 and finally with CVC4 (except that, if execution with a solver reaches 100% coverage, we do not run any subsequent solvers). Aratha+ does not perform algorithm selection itself, but uses $S = \{\text{G-Strings, Z3, CVC4}\}$ to perform three individual dynamic symbolic executions (one for each solver) and then collate the results. In practice, Aratha+ will actually compute the aggregate coverage for portfolio $S$.

The second approach, which we call Aratha++, is a variant of Aratha+ where the inputs found by a solver (i.e., the solutions of each path condition) are passed to the next solver to avoid the regeneration of the same inputs. As
Table 1: Evaluation results. Coverage is given in percentages, time in seconds.

<table>
<thead>
<tr>
<th>Solver</th>
<th>LINE</th>
<th>STMT</th>
<th>TIME</th>
<th>TOUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBSstmt</td>
<td>85.60</td>
<td>82.91</td>
<td>2.33</td>
<td>0</td>
</tr>
<tr>
<td>Z3</td>
<td>74.74</td>
<td>72.10</td>
<td>4.96</td>
<td>2</td>
</tr>
<tr>
<td>CVC4</td>
<td>79.44</td>
<td>76.56</td>
<td>4.27</td>
<td>2</td>
</tr>
<tr>
<td>G-Strings</td>
<td>81.54</td>
<td>78.74</td>
<td>6.93</td>
<td>0</td>
</tr>
<tr>
<td>Aratha++</td>
<td>85.73</td>
<td>83.04</td>
<td>10.51</td>
<td>0</td>
</tr>
<tr>
<td>Aratha++</td>
<td><strong>85.93</strong></td>
<td><strong>83.24</strong></td>
<td><strong>10.13</strong></td>
<td>0</td>
</tr>
</tbody>
</table>

Aratha+, Aratha++ does not perform algorithm selection in the strict sense because it uses all the available solvers. Aratha++ is a collaborative approach where at each DSE iteration $i$ with associated path conditions $\Gamma_i$, the execution heuristics $H_{exe}$ is this: “solve $\Gamma_i \cup \Delta$”, where $\Delta$ is a set of nogoods of the form $x_1 \neq v_1 \lor \cdots \lor x_n \neq v_n$ for each input $v = (v_1, \ldots, v_n)$ computed so far. In practice, Aratha++ tries to outperform the aggregate coverage by relying on the diversity of the generated inputs.

We compared Aratha+ and Aratha++ against the single-solver versions of Aratha using G-Strings, Z3, and CVC4. We evaluated them on the same benchmark of 197 already annotated JavaScript programs, coming from the Expose [21] test suite, used in [1]. As in [1], we set: a solving timeout of $T_{pc} = 10$ seconds for each path condition, a maximum number of $N = 1024$ DSE iterations for each program, and an overall DSE timeout of $T_{tot} = 300$ seconds (because sometimes reaching $N$ iterations can take too long). For Aratha+ and Aratha++, we set a timeout of $T_{tot}/3 = 100$ seconds for the execution of each individual solver of the portfolio. We ran all the experiments on an Ubuntu 15.10 machine with 16 GB of RAM and 2.60 GHz Intel® i7 CPU. We computed the coverage with the Istanbul tool [14].

Table 1 shows the results in terms of coverage and solving time: LINE is the average line coverage and STMT the average statement coverage (in percentage), TIME the average DSE time (in seconds) and TOUT the number of times the DSE reached timeout $T_{tot}$. To provide a baseline, we have added the performance of the “static” virtual best solver VBS$_{stmt}$, i.e., the fictitious selectors always choosing the solver achieving the maximum statement coverage for any given program. We do not include the VBS maximizing the line coverage because its performance is basically the same of VBS$_{stmt}$.

As can be seen, the DSEs using a portfolio of solvers is advantageous, both in terms of line and statement coverage. In fact, coverage achieved by Aratha+ and Aratha++ is slightly better than coverage achieved by VBS$_{stmt}$. As discussed in Section 3.4, this can happen in the absence of a “dominant” solver that can solve all the path conditions of the program.

As could be expected, the average DSE time comes out better for the single-solver approaches and VBS$_{stmt}$, because of the static, sequential approach used by Aratha+ and Aratha++. A proper parallel implementation and/or $H_{sel}$
heuristic would likely reduce this gap. Note, however, that TIME is not the most critical metric here. At least for the purpose of test data generation, the aim is to find the right balance among three competing objectives: maximizing coverage while minimizing test suite size, all in the shortest possible time.

The interesting thing in Table 1 is that Aratha++ is greater than the sum of its parts, because it can slightly improve the coverage computed by Aratha+. We can see why by looking at the cross-comparisons of Table 2.

Table 2 reports the number of times the solver on that row achieves a better statement coverage than the solver on that column (again, we do not report the table for line coverage, which is essentially the same) for each program of the benchmarks. Here we do not include VBSstmt as it makes little sense to make cross-comparisons between real and fictitious solvers.

No single-solver approach improves on Aratha+ or Aratha++ (this holds for line coverage too), even if the timeout of Aratha+ and Aratha++ is one third of their timeout. This confirms that the coverage of Aratha+ is the aggregate coverage for $S = \{ \text{G-Strings}, \text{Z3}, \text{CVC4} \}$. Note that this might be no longer true if the best coverage $\text{COV}_S(\mathcal{P})$ for a program $\mathcal{P}$ was computed by a solver $S$ in more than $T_{\text{tot}}/3$ seconds, because each solver of the portfolio is run for at most $T_{\text{tot}}/3$ seconds.

Aratha+ and Aratha++ are able to outperform the coverage achievable with the best single solver of $S$ for a given program (i.e., the one reaching the maximum coverage) for 4 and 5 programs respectively thanks to the combination of different solvers (as discussed in Section 3.4). Interestingly, in one case Aratha++ also outperforms the coverage of Aratha+. This happens because Z3 exploits the inputs computed by G-Strings and generates a new input that allows Aratha++ to achieve the full coverage.

A snippet of that program is shown in Fig. 5, where $x$ is its only symbolic variable. The DSE with G-Strings generates inputs {'','null','hello'}, the one with Z3 produces {false, {'length':39}, 'hello'} while the one with CVC4 computes {'','length':true,'hello'}. None of these will cause line 4 to be reached, because $x$.replace('h...o', '') === '' only succeeds when $x$ is equal to 'h...o'.4 Aratha expects the function replace to be

---

4 In JavaScript, $z$.replace($x,y$) returns a new string where $x$ is replaced by $y$ in $z$.

Note that $x$ may be a regular expression, but for simplicity Aratha only considers string values for $x$. In this case, the first occurrence of $x$ in $y$ is replaced.

---

Table 2: Coverage cross-comparisons.

<table>
<thead>
<tr>
<th>Solver</th>
<th>Z3</th>
<th>CVC4</th>
<th>G-Strings</th>
<th>Aratha+</th>
<th>Aratha++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z3</td>
<td>0</td>
<td>19</td>
<td>19</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CVC4</td>
<td>47</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>G-Strings</td>
<td>65</td>
<td>31</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Aratha+</td>
<td>67</td>
<td>43</td>
<td>29</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Aratha++</td>
<td>68</td>
<td>44</td>
<td>30</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Fig. 5: Snippet JavaScript program where Aratha++ outperforms Aratha+.

applied to strings only, and so an invocation `x.replace('h...o', '')` causes Aratha to abort DSE for the current execution trace for objects such as `[null]`, `{‘length’:true}`, nor `{‘length’:39}`. Using Aratha does not bring any benefit because the input it yields is just the union of the inputs produced by G-Strings, CVC4, and Z3.

If we instead use Aratha++, then Z3 can take advantage of the inputs computed by G-Strings and use the “nogoods” `x ≠ ‘’`, `x ≠ [null]`, and `x ≠ ‘hello’` each time it solves a new path condition. This enables Z3 to produce new inputs ‘\u0000’ and ‘h...o’. The latter in particular allows Aratha++ to reach line 4. This witnesses that generating multiple solutions and exchanging information between solvers is an aspect that deserves to be deepened.

5 Related Work

Algorithm selection started to attract the attention of the SAT community about a decade ago. SATzilla [37, 36] was one of the first SAT portfolio solvers. Its first version [37] used a ridge regression method to predict the performance of a SAT solver, while a subsequent version [36] improved the previous one with a weighted random forest approach provided with a cost-sensitive loss function for punishing mis-classifications in direct proportion to their performance impact. Another well-known AS approach for SAT problems is 3S [15] which first executes a static schedule of solvers computed offline and then, at run time, selects via k-Nearest Neighbour a solver to be executed for the remaining time. CSHC [24] is a clustering-based approach that combines 3S’s static scheduling with an algorithm selector based on cost sensitive hierarchical clustering. SATzilla, 3S, and CHSC won several gold medals in different editions of the SAT competition.

AS was successfully applied in other fields such as constraint programming [3, 12], answer-set programming [11], and planning [35]. For more comprehensive surveys on AS, we refer the reader to [13, 18, 33, 4, 16].

The ideas behind dynamic symbolic execution go back to Godefroid, Klarlund and Sen’s DART project [9]. Since then, advances in solver technology saw DSE tools improve rapidly, in some cases finding large-scale use. For example, Microsoft’s SAGE [10] DSE tool reportedly detected up to one third of all bugs discovered during the development of Windows 7−−bugs that were missed by other testing methods. Other popular DSE tools nowadays are for example [32, 30, 7, 23, 8, 34].
DSE seems particularly suitable for dynamic languages such as *JavaScript*. The first DSE application to *JavaScript* programs was the *Kudzu* project [28]. More recently, *EXPOSE* [21] was proposed to reason about *JavaScript* string matching via (extended) regular expressions, although in a limited fashion. *Aratha* [1] is the first *JavaScript* DSE tool capable of solving path conditions with different constraint solvers. It was built on top of *Jalangi* 2 [31], a framework for implementing dynamic analyses for *JavaScript*.

The only previous work we are aware of, combining both AS and DSE is that of Palikareva and Cadar [26], where different solvers are run in parallel, without any actual solver selection or information exchange between solvers. Note that this approach can never outperform the virtual best solver: its best possible implementation would only improve the DSE time of the *Aratha* and *Aratha*++ approaches defined in Section 4, but it will never achieve the coverage of *Aratha*++.

6 Conclusions

We have explored the scope and use of Algorithm Selection (AS) in Dynamic Symbolic Execution (DSE), proposing a generic AS/DSE framework. The framework is independent of the target language to analyse, as well as of the underlying solvers. The idea is to improve the DSE engine by using a portfolio of different solvers. The work is constrained by the fact that current DSE tools have not been built with portfolio solving in mind, but we have been able to conduct preliminary experiments in the context of *JavaScript* DSE. The results encourage further research in this direction, and we hope to spur sufficient interest to open a bridge between the AS, the DSE, and the constraint solving communities.

There are numerous directions in which this work should be extended. It would be interesting to extend the pool of solvers, benchmarks, and target languages. It would also be worthwhile exploring the use of other (including more sophisticated) coverage metrics.

A main goal is to develop an integrated DSE tool that is able to select a number of solvers from an arbitrarily large portfolio, and to run them in a concurrent and cooperative way (i.e., by enabling the information exchange between solvers). A useful step in this direction would be the definition of a constraint language able to encode the path conditions of a given programming language regardless of the target solver(s) used to solve the path conditions.

Acknowledgments

This work is supported by the Australian Research Council (ARC) through Linkage Project Grant LP140100437 and Discovery Early Career Researcher Award DE160100568.
References


Translation of Interactive Datalog Programs for Microcontrollers to Finite State Machines

Mario Wenzel and Stefan Brass

Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik, Von-Seckendorff-Platz 1, D-06099 Halle (Saale), Germany
brass@informatik.uni-halle.de, mario.wenzel@informatik.uni-halle.de

Abstract. “Smart” devices have become ubiquitous in modern households and industry. Especially in home-automation, robotics, and sensing tasks, rule-based approaches seem ideal to describe the behavior of interactive systems. But managing input data and state is hard. With little choice of programming language, most code targeted at microcontrollers is written in imperative C or assembler. Microlog is a deductive database language with a strong logic foundation based on Datalog extended with a representation of time and calls to external functions that may be used to control sensors and actors. In this paper we describe a method to precalculate sets of Datalog facts that may be derivable for a point in time. Values that will be known only at runtime are represented as parameters of those “states”. During “state transition”, a small number of conditions on parameters and input values must be checked. By representing a possibly quite large number of facts as a single state number and a few parameter values, memory and computing time are saved. If no parameters are needed, the result of this compilation is basically a finite state machine.

Keywords: Deductive Database · Datalog · Microcontroller · Arduino · Compiler

1 Introduction

With the introduction of cheap programmable microcontrollers, powerful reprogrammable “smart” devices have found their way into our homes and offices. “Smart everything” is not an understatement as we find smart vacuum cleaners, smart light-bulbs, smart coffee machines, smart radios and TVs, smart ovens and microwaves, smart sex-toys, smart toothbrushes, smart washing machines and the like. Alternative open source firmwares are available for quite a few of the mentioned device types. With expanded lifespan and utility from what the vendor originally intended to sell, the aftermarket for flashable devices flourishes. So do the open source communities around those devices.

Naturally those cheap programmable devices find their way into hobbyists’ workshops as well as school and university courses and curricula [112]. However, the programming languages and paradigms that are supported by microcontroller systems such as the Arduino are limited.
We believe that declarative programming can be an interesting option even for such small devices. Declarative languages have many advantages:

- Declarative programs are usually shorter than an equivalent program in a procedural language. This enhances the productivity of the programmers.
- There can be no problems with uninitialized variables or memory leaks.
- The language has a mathematically precise semantics based on logic, which makes programs easier to verify and programming arguably easier to teach.

Our declarative language Microlog is a language that allows us to model both program state and side effects in a declarative manner [14]. Based on the Datalog variant Dedalus [2], Microlog has strong foundations in logic. Dedalus captures a notion of state, similar to the Statelog language [10], but is embedded in Datalog. Our model of side-effects is similar to action atoms and external atoms from Answer Set Programming [5,7] with deterministic semantics. Rules are usually quite small, have a simple structure, and a well-defined and explicit interface to the rest of the program. This allows us to reason deductively about program and world states. Our goal is to declaratively program microcontrollers and give static safety guarantees for data-driven interactive programs on microcontrollers. The safe use of our limited resources is quite important. The GPIO pins, for example, should only be used in a consistent and deterministic manner. Our microcontroller, even though it stores and derives some data, should also not run out of memory.

In this paper, we present a novel compilation technique for Microlog-programs. It precomputes possible “states”, which are sets of Datalog-facts that are true at a point in time. This does not work for all Microlog-programs, because one can write programs for which the number of facts keeps growing over time. However, if we cannot prove that the number of facts for a time point is bounded, the program is anyway problematic, especially for small microcontroller systems. If the approach is applicable for a program, the result is often similar to a finite state machine where the IO is done during state transitions.

Our approach targets AVR-based microcontrollers, like the Arduino, with an 8-bit CPU and a memory that ranges from 0.5 to 8 KB. While our generated C-Code can be used for different hardware platforms, we have chosen microcontrollers with the ATmega168 8-bit processor as the minimal viable target.

- There is only 1 KB of SRAM available that is used for both heap and stack data. This means we are limited in operational memory for storing database facts and in algorithm design with regards to function call depth.
- 16 KB of Flash memory can be used to store the program. This might seem a lot in comparison, but this is also used to store additional libraries for peripheral access that are wanted by the user.
- It has an operational speed of 20 MHz, which is a lot compared to the amount of data we have to operate on.

These limitations, along with the difficulty of dynamic memory management, preclude complex approaches where a possible runtime must model non-deterministic choice or have backtracking semantics. From a Microlog program we
generate simple C-code and we generically interface with the rest of the system by calling external C-functions (e.g., from libraries). This approach works for other embedded systems and processors as well.

To test the utility of our language we also build programs for LEGO EV3 robots. EV3 units have a “proper” ARM CPU and run Linux. With 64 MB of SRAM, resource management is less of a concern there, and more complex approaches for modelling and planning are also available.

In Section 2 we will recapitulate syntax and semantics of the Microlog language. In contrast to our earlier paper, we have simplified the IO: All calls of external procedures are now done in the head. In Section 3, we do the transition from the standard computation of the minimal model to the computation by states. Section 4 explains the precomputation of states at compile time, which is the main technical contribution of this paper.

2 Language, State Management, Example Application

Our query language is modeled after the Dedalus$_0$ language. Dedalus$_0$ and our language “Microlog” are based on Datalog. A Datalog program is a finite set of rules of the form $A \leftarrow B_1 \land \cdots \land B_n$, where the head literal $A$ and the body literals $B_i$ are atomic formulas $p(t_1, \ldots, t_m)$ with a predicate $p$ and terms $t_1, \ldots, t_m$.

Terms are constants or variables. We also allow arithmetic comparisons in the rule body (i.e., $=, \neq, <, \leq, >, \geq$). Furthermore, we need the successor predicate for natural numbers $\text{succ}$, but this is only allowed as last literal in so-called “inductive rules” (see below). Let $I_B$ be the standard interpretation for the built-in predicates (comparisons and $\text{succ}$), e.g., $I_B[\text{succ}] = \{(i, i+1) \mid i \in \mathbb{N}_0\}$.

Rules must be range-restricted, i.e., all variables that appear anywhere in the rule must appear also in a body literal with a predicate that is not a comparison (the use of $\text{succ}$ is specially restricted, see below). This ensures that all variables are bound to a value when the rule is applied.

In order to explicitly model time, Microlog programs have the syntactic restriction that every predicate must have a first argument from the domain of the natural numbers which we refer to as the timestamp. Some fact $p(\ldots)$ is true in timestamp $n$ iff $p(n, \ldots)$ is in the minimal model of our program. We refer to the selection of all facts with a certain timestamp as a state.

All body literals in a rule body must have as their timestamp the same special variable $T$, as rules may only rely on facts from a single timestamp (the variable $T$ always refers to “now”). The rule head either shares the same $T$ as its timestamp (as in Dedalus, this is called a deductive rule) or it has the timestamp $T'$ and the literal $\text{succ}(T, T')$ is the last literal of the rule body (as in Dedalus, this is called an inductive rule).

Deductive rules allow for normal Datalog deduction steps and inductive rules govern how data from facts of one timestamp is reproduced into facts of the following timestamp.

Rules without body are only allowed as initial facts if their timestamp is 0. Syntactically, no other configurations of the timestamp arguments are allowed.
```c
#define HIGH 0x1
#define LOW 0x0

void digitalWrite(uint8_t pin, uint8_t val);
int digitalRead(uint8_t pin);
```

Fig. 1. Extract from Arduino.h Header File

The special variables \( T \) and \( T' \) are not allowed to be used elsewhere in any other part of a rule or bound to other variables, neither is the succ-predicate.

This model can be used to update relations in a stateful fashion. If a fact is not transported from one timestamp to the next, we have a notion of deletion. The notion of state captured by the timestamp is similar to the Statelog language \[10\]. The facts with some timestamp \( n \) can be seen as “happening earlier” than the facts with timestamp \( m \) with \( n < m \). This is useful to model interactions with the environment. The minimal model for such a logic program may extend into infinity. The evaluating program does not terminate and as an interactive system we do not want it to.

Of course, a Datalog program for a Microcontroller must interface with the libraries for querying input devices and performing actions on output devices. One approach would be to have a fixed set of built-in predicates, but there are quite a lot of library functions and new libraries are being developed, e.g., for new types of input and output devices, or controller boards. A few examples of interface functions are shown in Fig. 1.

Our modelling of side effects in logic programs can be likened to HEX programs with action atoms \[7\]. The observation of the actions is similar to an external source of computation \[5\]. External sources of computation in logic programs have been modeled as functional oracles which, in our case, describe the environment during a particular run of a Microlog program.

For each function \( f \) that can be called, there is a special predicate \texttt{call\_f} with a reserved prefix “\texttt{call\_}”. The predicate has arguments of the function to be called, arguments for the return values, and of course the standard time argument. E.g., derived facts about the predicate \texttt{call\_digitalWrite}(\texttt{T'}, \texttt{Pin}, \texttt{Val}) lead to the corresponding calls of the interface function \texttt{digitalWrite} in the following state \( T' \). The set-semantics ensures that duplicate calls are eliminated, i.e., even if there are different ways to deduce the fact, only one call is done. The sequence of calls within the same timestamp is undefined. If a specific sequence is required, the calls must be spread out over multiple timestamps.

For each interface function \( f \) there is a second predicate \texttt{ret\_f} that contains all parameters of the call and a parameter for the return value. For instance, for the function \texttt{digitalRead}, there are two predicates:

\[1\] We actually allow users to define a whole statement block as a special predicate. This allows combining interface functions that always need to be called together, defining multiple return arguments, or just doing arithmetic. To avoid confusion, we only refer to function calls from here on out.
– call_digitalRead(\( T' \), Pin, ?), and
– ret_digitalRead(\( T \), Pin, Val).

For the output positions that are not assigned a value in the “call” predicate but are in the “ret” we use the special marker ? to achieve a consistent argument list. One could view this as an existentially quantified anonymous variable with the promise that in the corresponding ret-predicate, there will be some return value.

A call is only ever done in the next state, so that the result value is also only available in the next state. This ensures, e.g., that the occurrence of a call cannot depend on its own result.

Since calls of interface functions usually have side effects and cannot be taken back, it is important to clearly define which calls are actually done. In contrast, the evaluation sequence of literals in a rule body can be chosen by the optimizer. Therefore the special call_f predicate can be used only in rule heads. Correspondingly, the ret_f predicate can only be used in the rule body and is defined by the derived call_f-facts and the environment.

A form of condition-action rules can be seen (amongst other systems for event theory) in Event Calculus [11] and Logic Production Systems [9] with the main difference that Microlog does not allow for negation or disjunction in the rule heads. Also, all actions that are are possible are taken, not just a single one. This removes backtracking and nondeterministic choice from our system. We claim that the embedding into Datalog is powerful enough (e.g., to implement some decision procedure for action prioritization, or planning procedures) and allows us to readily apply well-researched methods and techniques.

We add some syntactic sugar to make it easier to work with the syntactic restrictions. From the rule structure and syntactic restrictions it should be clear when we refer to the sugared version of Microlog:

<table>
<thead>
<tr>
<th>Unsugared Version</th>
<th>Sugared Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deductive Rules: the time argument is left out in the rule head and every subgoal.</td>
<td></td>
</tr>
<tr>
<td>( p(T, X) \leftarrow q(T, X, Y) \land p(T, Y) )</td>
<td>( p(X) \leftarrow q(X, Y) \land p(Y) )</td>
</tr>
<tr>
<td>Inductive Rules: the suffix “@next” is added to the rule head and the time argument is left out in the rule head and every subgoal and we leave out the succ predicate.</td>
<td></td>
</tr>
<tr>
<td>( \text{call}_f(T', X, ?) \leftarrow p(T, X) \land \text{succ}(T', T') )</td>
<td>( \text{#f}(X, ?)@\text{next} \leftarrow p(X) )</td>
</tr>
<tr>
<td>Initial Facts: replacing the time argument 0, the suffix “@0” is added.</td>
<td></td>
</tr>
<tr>
<td>( p(0, 5) )</td>
<td>( p(5)@0 ).</td>
</tr>
<tr>
<td>Static Facts: We leave the body empty. time is a reserved predicate defined by time(0) and time(( T' )) \leftarrow time(T) \land \text{succ}(T, T') ). These two rules are added to every program. They ensure that time will be true for all states.</td>
<td></td>
</tr>
<tr>
<td>( p(T, 5) \leftarrow \text{time}(T) ).</td>
<td>( p(5) ).</td>
</tr>
<tr>
<td>IO: We replace the call_-,prefix, which can only appear in rule heads, with #. As the ret_-prefix can only appear in rule bodies, we replace that with # as well.</td>
<td></td>
</tr>
<tr>
<td>( \text{call}_f(T', X, ?) \leftarrow p(T, X) \land \text{succ}(T, T') ).</td>
<td>( \text{#f}(X, ?)@\text{next} \leftarrow p(X) ).</td>
</tr>
<tr>
<td>( p(T, X) \leftarrow \text{ret}_f(T, 5, X) ).</td>
<td>( p(X) \leftarrow #f(5, X) ).</td>
</tr>
</tbody>
</table>

Finally, we need also constants from the interface definition. If our Datalog program contains e.g. #HIGH, this corresponds to the constant HIGH in the generated C-code.
% static example configuration
hasWindow(2).
adjacent(1, 2).

% gathering world state
#readWindow(R, ?)@next :- hasWindow(R).
#readDoor(A, B, ?)@next :- adjacent(A, B).

% deduce model using transitive closure
windowOpen(R) :- #readWindow(R, #open).
doorOpen(A, B) :- #readDoor(A, B, #open).
connected(A, B) :- doorOpen(A, B).
connected(B, A) :- doorOpen(A, B).
connected(A, C) :- connected(A, B), connected(B, C), A != C.

% effects
#heatingOff(R)@next :- windowOpen(R).
#heatingOff(O)@next :- windowOpen(R), connected(R, O).

Fig. 2. Heating control program in Microlog (actual concrete syntax)

With the fixpoint semantics of Datalog and external function calls, we can
describe reactive data-driven programs with more complex behavior than, for
example, typical home automation rules.

Example 1. We can easily describe a system where the heating not only shuts
off in a room with an open window (a common use case for home automation),
but also in all (other) rooms connected via open doors (see Figure 2). The
function calls are from a fictitious library that wraps communication to the
sensors and actors, #open is a C constant from that library. It is usually the
case that control to hardware is provided by some library as protocols like
I²C need to be observed or read analog voltage levels need to be translated to
values in expected units. The static rules of hasWindow and adjacent are the
configuration of our program to our specific example home with two connected
rooms where the second room has window (also used in Section 3). A user, if this
program was provided to them, would only need to add those facts to configure
it for their home.

3 Computing a Sequence of States

After replacing the abbreviations, a Microlog-program is a set \( P_M \) of Datalog-
rules (and facts). However, for the deduction, not only these rules are used, but
also input facts from the external environment that contain the return values
of function calls. Let \( \mathcal{E} \) be the set of all such facts for the \( \text{ret_f} \)-predicates. The
semantics of the Microlog program \( P_M \) is the mapping from input facts \( \mathcal{E} \) to the

\(^2\) At the time of writing, PlatformIO.org hosts over 7,000 libraries for embedded devices. More than half are available for Arduino.
minimal model $\mathcal{I}_{\min}$ of $P := P_M \cup \mathcal{E}$, i.e., the set of all derivable facts. We can use the standard $T_P$ operator for deriving facts that are immediate consequences of the rules and already known facts:

$$T_P(\mathcal{I}) := \{ A\theta \mid A \leftarrow B_1 \land \cdots \land B_m \in P, \theta \text{ is a ground substitution for this rule such that}$$

$$\text{for all } i = 1, \ldots, m: \ B_i\theta \in \mathcal{I} \text{ or}$$

$$B_i \text{ has a built-in predicate and } \mathcal{I}_B \models B_i \theta \}. $$

The least fixed point of this operator, $\mathcal{I}_{\min}$, is the minimal Herbrand model of $P$. It can be obtained by iterating the operator, i.e., one starts with $\mathcal{I}_0 := \emptyset$, and then has a series of Herbrand interpretations (sets of facts) $\mathcal{I}_{i+1} := T_P(\mathcal{I}_i)$.

The “limit” of this construction yields $\mathcal{I}_{\min} = \bigcup_{i=0}^{\infty} \mathcal{I}_i$.

We are actually not interested in arbitrary sets $\mathcal{E}$, but only sets satisfying the causality requirement that the $\text{call}_f$-facts in $\mathcal{E}$ correspond to derived $\text{call}_f$-facts:

**Definition 1 (Causal Set of Input Facts).** Let a Microlog-Program $P_M$ be given. A set $\mathcal{E}$ of facts is causal (for $P_M$) iff

- it contains only facts with return predicates ($\text{ret}_f$), and
- for each fact $\text{ret}_f(i, c_1, \ldots, c_n) \in \mathcal{E}$ there is $\text{call}_f(i, c'_{1}, \ldots, c'_{n}) \in \mathcal{E}$ with the same timestamp $i$ and such that for each $j$, $1 \leq j \leq n$, $c'_j = c_j$ or $c'_j = ?$ (the special marker for return arguments),
- and, vice versa, for each fact $\text{call}_f(i, c'_1, \ldots, c'_n) \in \mathcal{E}$ there is exactly one fact $\text{ret}_f(i, c_1, \ldots, c_n) \in \mathcal{E}$ that is matching in the above sense.

It might be a philosophical problem that the input facts seem to be there before the computation starts that can produce the calls that cause them. But in this way, the formal definition is simpler and in the spirit of standard Datalog.

By considering only sets $\mathcal{E}$ that satisfy the causality property, we eliminate the unreasonable cases. Because calls are derived in Microlog always for the next point in time, it is not possible that a call depends on its own return value.

However, we will also give the definitions for a computation in the sequence of timestamps, which is a more realistic model of what should happen in practice. Fortunately, both views lead to the same model.

We want to compute a sequence $S_0, S_1, \ldots$ of sets of facts, where $S_i$ contains facts derived at time $i$. Because the time information is contained in the position $i$ of the set $S_i$ in the sequence, we can get rid of the time argument in the facts. This is quite similar to getting back from the full Datalog version of a Microlog program to a version that uses the syntactic sugar for hiding the time argument. However, the result is now pure Datalog with special predicates. Since at time $i$, also facts for the next state are derived, we need the following special predicates:

- For all normal predicates $p$, we introduce a new predicate $\text{next}_p$, and
- for the $\text{call}_f$ predicates, we introduce a predicate $\text{ncall}_f$ (calls are always derived for the next state).

Now for literals $A$, we write $\hat{A}$ for a version without time argument. More precisely, if $A$ is $p(t_0, t_1, \ldots, t_n)$, $\hat{A}$ is
This removal of the time argument can also be applied to a set of rules $\mathcal{R}$. Let

\[
\overline{\mathcal{R}} := \{ \overline{A} \leftarrow \overline{B}_1 \land \cdots \land \overline{B}_{n-1} \mid A \leftarrow B_1 \land \cdots \land B_n \in \mathcal{R} \\
\text{and } B_n \text{ is } \text{succ}(T, T') \text{ or } \text{time}(T) \} \cup \\
\{ \overline{A} \leftarrow \overline{B}_1 \land \cdots \land \overline{B}_n \mid A \leftarrow B_1 \land \cdots \land B_n \in \mathcal{R} \\
\text{and } B_n \text{ is neither } \text{succ}(T, T') \text{ nor } \text{time}(T) \}.
\]

We also need the converse operation, i.e. adding a time argument, for the obtained facts. Let $p(c_1, \ldots, c_n)[i]$ be:

- $\text{call}_f(i + 1, c_1, \ldots, c_n)$ if $p$ is of the form $\text{call}_f$,
- $q(i + 1, c_1, \ldots, c_n)$ if $p$ is of the form $\text{next}_q$,
- $\overline{p}(i, c_1, \ldots, c_n)$ otherwise.

This definition is extended to sets of facts: $\mathcal{F}[i] := \{ F[i] \mid F \in \mathcal{F} \}$.

Since we need to apply the rules piecewise we need to partition the program $P_M$ into

- $P_{\text{init}}$, all facts with time argument 0,
- $P_{\text{always}}$, consisting of the rules and static facts, which are applicable for any point in time: $P_{\text{always}} := P_M - P_{\text{init}}$.

Now we can compute the facts in the minimal model in the chronological order, timestamp by timestamp. We do so by using the predicates without time argument, i.e., $\overline{P}_{\text{init}}$ and $\overline{P}_{\text{always}}$.

- $\mathcal{N}_0 := \overline{P}_{\text{init}}$

The set $\mathcal{N}_i$ contains seed facts for the next state $i$. Since at time 0, there is no previous state, the initial facts take the role of the seed facts here.

- $\mathcal{E}_0 := \emptyset$

At time 0, there are no calls, and therefore no external input facts.

- $\mathcal{S}_i := \text{fp}(T_{P_{\text{always}} \cup \mathcal{N}_i \cup \mathcal{E}_i})$

The “state $i$”, i.e., the facts for time $i$ are obtained iteratively by applying the rules in $\overline{P}_{\text{always}}$, to the seed facts $\mathcal{N}_i$, the static facts (also part of $\overline{P}_{\text{always}}$) and the input facts $\mathcal{E}_i$ until a fixpoint is reached. The derived facts include also calls and $\text{next}_p$-facts that refer to the next state.

- $\mathcal{N}_i := \{ p(c_1, \ldots, c_n) \mid \text{next}_p(c_1, \ldots, c_n) \in \mathcal{S}_{i-1} \}$

This extracts and transforms the seed facts for next point in time $i$, $i \geq 1$.

- $\mathcal{C}_i := \{ \text{call}_p(c_1, \ldots, c_n) \mid \overline{\text{call}}_p(c_1, \ldots, c_n) \in \mathcal{S}_{i-1} \}$

These are the call facts for next point in time $i$, $i \geq 1$. One can see these calls as “output” of some kind of state machine (not yet a finite state machine).

- $\mathcal{E}_i := \{ \text{ret}_f(c_1, \ldots, c_n) \mid \text{ret}_f(i, c_1, \ldots, c_n) \in \mathcal{E} \}$

These are the input facts from $\mathcal{E}$ for time $i \geq 1$ (with the time argument removed). They are given by the external environment, but note that causality requires that each fact in $\mathcal{E}_i$ is the result of a call in $\mathcal{C}_i$. Since the calls were computed in the previous state, we can actually execute them to compute $\mathcal{E}_i$. 


Theorem 1. This iterative computation by timepoints yields exactly the minimal model of the given program if we add the time argument: $\mathcal{T}_{\text{min}} = \bigcup_{i=0}^{\infty} S_i[i]$.

Theorem 2. If $P_M$ is finite and $E$ is causal, each state $S_i$, $i \in \mathbb{N}_0$, is finite.

This ensures that we can effectively compute each state. Please remember that $\text{succ}$ can only be used for switching to the next point in time. It cannot be used for computations within a state.

4 Precomputation of States

Our goal is to precompute the possible states, i.e., sets of derivable facts for a timestamp. Of course, such a precomputation is not always possible, because for some programs the set of facts at a point in time can possibly grow without limits over time. However, for the given small hardware, such programs would be problematic anyway. When microcontrollers are embedded in some hardware application, one would expect that the program provably works, and does never stop with an “out of memory” error. Often the hardware does not even permit to communicate such a message.

Of course, there are a number of values that are only known at runtime (input values). We use special variables to model them:

Definition 2 (Parameter Variable). Let $V_1, V_2, \ldots$ be a sequence of pairwise distinct variables that do not occur in the given Datalog program (they are reserved). We call these variables “parameter variables”.

The parameter variables correspond to memory locations that are used for storing return values of the function calls (unknown at “compile time”).

Definition 3 (Parameterized Fact). A parameterized fact is a formula of the form $p(t_1, \ldots, t_m)$ where each $t_i$, $1 \leq i \leq m$, is a constant or a parameter variable.

Definition 4 (Parameterized State). A parameterized state is a finite set of parameterized facts.

Parameters have a global meaning in the state: If two parameterized facts in a state both contain $V_1$, they will have the same value. This is a difference to normal variables in rules, which have only local scope (limited to a rule).

The initial state is $S_0$ as in Section 3. It does not contain parameters because it does not depend on input.

Definition 5 (Initial State). The initial state is $S_0 = \text{lp}(\bar{T}_\text{always} \cup \bar{T}_\text{init})$.

For instance, the example (see Fig. 2) contains the following rule (after elimination of syntactic sugar and the removal of the time argument as in Section 3):

\[ \text{ncall} \cdot \text{readWindow}(R, ?) \leftarrow \text{hasWindow}(R). \]
This rule can be applied with the static configuration fact hasWindow(2). All rules depending on input are not yet applicable in the initial state. Therefore, the complete initial state $S_0$ in the example is:

```
hasWindow(2)  ncall_readWindow(2, ?).
adjacent(1, 2) ncall_readDoor(1, 2, ?).
```

Now let any parameterized state be given (for instance, the initial one). Our goal is to compute the possible successor states. The given state contains a number of (parameterized) ncall_f-facts. For each such ncall_f-fact, there will be a ret_f-fact in the next. Since during the translation, we do not know the actual input values, we use symbolic facts with parameters as defined above. If there are several symbolic ret_f-facts, the parameters of these must be disjoint. The parameters must also be disjoint from parameters in next_p-facts (these will also be transferred as p-facts into the next state). In the example, we have two facts corresponding to the returns of the called functions:

```
ret_readWindow(2, V1).  Generated Code: V1 = readWindow(1);
ret_readDoor(1, 2, V2).   V2 = readDoor(1, 2);
```

In the example, there are no next_p-facts. In general, the computation of the next state starts with the following facts:

**Definition 6 (Seed Facts).** Let $S$ be a parameterized state, and let

- $next_p(t_{i,1}, \ldots, t_{i,k})$ for $i = 1, \ldots, m$ be all (parameterized) next_p-facts in $S$, and
- $ncall_f(u_{i,1}, \ldots, u_{i,l})$ for $i = 1, \ldots, n$ be all (parameterized) ncall_f-facts in $S$ (in some fixed order).

Then the seed facts $seed(S)$ for the next state are:

- $p_i(t_{i,1}, \ldots, t_{i,k})$ for $i = 1, \ldots, m$, and
- $ret_f(\hat{u}_{i,1}, \ldots, \hat{u}_{i,l})$ for $i = 1, \ldots, n$, where $\hat{u}_{i,j}$ is $u_{i,j}$ unless $u_{i,j}$ is $?$, in which case $\hat{u}_{i,j}$ is the first unused parameter variable.

With these “seed” facts (and the static facts), we want to apply again the rules to compute the next state. Now the problem is that for some rules, the values of the parameters do matter. E.g., consider the rule:

```
windowOpen(R) ← ret_readWindow(R, #open).
```

So the question is whether the parameter $V_1$ is equal to the constant #open or not. Now, when we want to apply the rule, we do a unification between the rule body and existing parameterized facts. In the example, this will bind the parameter $V_1$ to the constant #open. Since at compile time, we do not know the value of $V_1$, the result will be a “conditional fact”:

```
windowOpen(2) ← V_1 = #open.
```
“Conditional facts” were used by Brass and Dix for characterizing and computing negation semantics [3]. There, the conditions were delayed negative literals.

Definition 7 (Conditional Fact). A conditional fact is a formula of the form
\[ p(t_1, \ldots, t_m) \leftarrow \varphi \]
where each \( t_i, i = 1, \ldots, m \), is a constant or a parameter variable, and \( \varphi \) is a consistent conjunction of atomic formulas \( u \gamma u' \) with \( \gamma \in \{=, \neq, <, \leq, \geq, >\} \) and \( u \) and \( u' \) are parameters or constants, with at least one parameter.

For the unification, we would need only conditions of the form \( V = c \) and \( V = V' \). However, we permit comparisons as built-in predicates in the rules, and if body literals with such predicates cannot be evaluated at compile time, they also become part of the condition.

Note that “consistent” means here that there is a variable assignment for the parameters such that the formula is true in the standard interpretation of the built-in predicates \( I_B \) with these values of the variables. E.g., \( V_1 = 5 \land V_1 < 3 \) is inconsistent. In the same way, \( \varphi_1 \) and \( \varphi_2 \) are called equivalent, if they have identical truth values in \( I_B \) for all variable assignments (ground substitutions).

Conditional facts with inconsistent conditions would not be useful. The consistency of conjunctions of the above form can be easily checked [8,4]. If one wants additional built-in predicates, one might need a more powerful constraint solver. However, forbidding inconsistent conditions is only an optimization: Without this, states might be obtained that are actually not reachable.

Definition 8 (Rule Application to Conditional Facts). Let
\[ A \leftarrow B_1 \land \cdots \land B_m \land C_1 \land \cdots \land C_n \]
be a rule, where the \( B_i, i = 1, \ldots, m \), are normal literals, and the \( C_i, i = 1, \ldots, n \), are literals with a built-in predicate. Let \( B'_i \leftarrow \varphi_i, i = 1, \ldots, m \), be conditional facts and \( \theta \) be a most general unifier for \( (B_1, \ldots, B_m) \) and \( (B'_1, \ldots, B'_m) \) that does not map parameters to variables of the rule (since the direction of variable-to-variable bindings is arbitrary, this is always possible). Let
\[ \Phi := \{\varphi_i \mid i = 1, \ldots, m\} \cup \{C_i \theta \mid i = 1, \ldots, n\} \cup \{V_i = V, \theta \mid V \text{ is a parameter variable occurring in some } B'_i, 1 \leq i \leq m\}. \]

If \( \Phi \) is consistent, then the rule application yields \( A\theta \leftarrow \varphi \), where \( \varphi \) is equivalent to a conjunction of all formulas in \( \Phi \). Else, the rule application is not possible.

We permit any formula equivalent to \( \Phi \), because we of course want to eliminate duplicate conditions, and trivial conditions such as \( V = V \), which can occur when \( \theta \) is the identity mapping for \( V \). Of course, we also want to eliminate \( C_i \theta \) if it does not contain parameter variables (because of the required consistency, this must be true in the standard interpretation of the built-in predicates). The implementation is free to add or remove implied conditions. This might help to find duplicates. Actually, we want to eliminate not only duplicates, but conditional facts that are “weaker” than another conditional fact:
**Definition 9 (Subsumed Conditional Fact).** A conditional fact $A_1 \leftarrow \varphi_1$ is subsumed by a conditional fact $A_2 \leftarrow \varphi_2$ iff for every ground substitution $\theta$ (for the parameters that occur in at least one of them) whenever $\mathcal{I}_B \models \varphi_1\theta$, also $A_1\theta = A_2\theta$ and $\mathcal{I}_B \models \varphi_2\theta$ hold.

Two conditional facts are called equivalent iff they subsume each other.

In other words: $p(t_1,\ldots,t_n) \leftarrow \varphi_1$ is subsumed by $p(u_1,\ldots,u_n) \leftarrow \varphi_2$ iff $\varphi_1$ implies $\varphi_2 \land (t_1 = u_1) \land \cdots \land (t_n = u_n)$.

Obviously, subsumed conditional facts can be deleted in the fixpoint iteration, because the subsuming conditional fact is more general (in the case of equivalent facts, all except one can be deleted). An example for a quite complex case is:

$p(V_1) \leftarrow V_1 \leq V_2 \land V_1 \geq V_2$ is subsumed by $p(V_2)$. The first conditional fact is only applicable if $V_1 = V_2$, and then it produces the same fact as the second.

One possible algorithm is to “normalize” derived conditional facts in the following way. First, expand the condition in the rule body by all easily derived consequences, especially equations. Now, if the condition contains $V_i = c$ with a constant $c$, eliminate $V_i$ from the head and other conditions by replacing it there by $c$ (note that $V_i = c$ must be kept in the condition, because the conditional fact is applicable only under this condition). In the same way, if the condition contains $V_i = V_j$ with $i < j$, replace $V_j$ everywhere else by $V_i$. Finally, order the remaining non-trivial conditions in some standard order. Then delete conditional facts with the same head and a superset of the conditions in the body.

**Definition 10 (Successor State).** We write $\bar{\Phi}_p$ for the immediate consequence operator for conditional facts (possibly with elimination of subsumed conditional facts). Let a parameterized state $\mathcal{S}$ be given. The conditional successor state is $S' := \bar{\Phi}_p(\bar{\Phi}_{\text{seed}}(\mathcal{S}) \cup P_{\text{always}})$.

From the conditional successor state, we get one successor state for each consistent valuation $\nu$ of the atomic formulas appearing in the conditions. A valuation is consistent if the conjunction of the atomic formulas it assigns true and the conjunction of the negations of the formulas it assigns false is consistent. Then $S'|\nu := \{ A | A \leftarrow \varphi \in S', \nu \models \varphi \}$ is the successor state for $\nu$.

In the example, the conditional successor state is:

```
ret_readWindow(2, V1).  windowOpen(2) ← V1 = #open.
ret_readDoor(1, 2, V2).  doorOpen(1, 2) ← V2 = #open.
hasWindow(2).             connected(1, 2) ← V2 = #open.
adjacent(1, 2).           connected(2, 1) ← V2 = #open.
ncall_readWindow(2, ?).   ncall_heatingOff(2) ← V1 = #open.
ncall_readDoor(1, 2, ?).  ncall_heatingOff(1) ← V1 = #open ∧ V2 = #open.
```

Thus, depending on the values of $V_1$ and $V_2$, there are four possible states (leaving out the unconditional part, i.e. the left side above).
However, two states differ in their outside behavior only if they have different ncall\_f or next\_p-facts. These facts determine the calls that are done (the “output” of the machine), and the information moved into the next state (the state transition). Other facts are only needed during the computation. Thus, we can merge two states \( S \) and \( S' \) if \( \text{seed}(S) = \text{seed}(S') \). In the example, \( S_3 \) and \( S_4 \) have the same behavior as \( S_0 \). We do not have to compute their successor states.

In contrast, \( S_1 \) and \( S_2 \) behave differently, because of the calls to \( \text{heatingOff} \). However, since \( \text{ret}\_\text{heatingOff} \) does not appear in the rule bodies, this actually does not influence the successor states. So in this simple example, we are already done and have only three states: \( S_0, S_1, \) and \( S_2 \).

The function calls in these states are (with parameter variables for the result):

<table>
<thead>
<tr>
<th>( S_0 )</th>
<th>( S_1 )</th>
<th>( S_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>readWindow(2, ( V_1 ))</td>
<td>readWindow(2, ( V_1 ))</td>
<td>readWindow(2, ( V_1 ))</td>
</tr>
<tr>
<td>readDoor(1, 2, ( V_2 ))</td>
<td>readDoor(1, 2, ( V_2 ))</td>
<td>readDoor(1, 2, ( V_2 ))</td>
</tr>
<tr>
<td>heatingOff(2)</td>
<td>heatingOff(2)</td>
<td>heatingOff(2)</td>
</tr>
</tbody>
</table>

In the example, the transition function is independent of the current state:

<table>
<thead>
<tr>
<th>( V_1 = #\text{open} )</th>
<th>( V_1 \neq #\text{open} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V_2 = #\text{open} )</td>
<td>( V_2 \neq #\text{open} )</td>
</tr>
</tbody>
</table>

However, if \( V_1 \) (the sensor of the window) is not \#open, the value of \( V_2 \) (the sensor at the door) does not matter. If we look at the full set of conditional facts, it does influence the connected-facts, but these are important for the observable outside behavior only if the window is open. The resulting finite state machine is shown in Fig. 3 (gray box means \( \text{heatingOff} \) in that room).

In general, states can be computed as shown in Fig. 4. For space reasons, we left out code generation, but e.g. the two recursive calls to \( \text{gen} \) would be preceded by \( \text{if}(\varphi_i) \) and \( \text{else} \), and when the end of the recursion is reached, the variable for the current state is assigned the state number of the successor state \( S_{\text{true}} \). The choice of the condition \( \varphi_i \) for the case analysis is important for the size of the generated code. E.g., if in the example, one would first choose \( V_2 = \#\text{open} \), one would have to duplicate the case analysis for \( V_1 \). As a heuristics, we propose to minimize in each step the number of split conditions, i.e. remaining conditions in conditional facts that appear in both cases.

13
Of course, there is the question of termination. If no input values are copied to the next state, i.e., no next-p-fact and no ncall-f-fact contains a parameter variable, termination is guaranteed, and we get a classic finite state machine as in the example: First, the number of calls is bounded, because they can contain only constants explicitly occurring in the program. Thus, also the number of parameter variables is bounded (for new parameter variables, the one with the smallest index is chosen that is not currently used). Finally, the number of facts in a state is bounded because there is only a fixed set of predicates and arguments. However, the method works more generally even if states are generated that do contain parameters (then the result is not really a finite state machine). In this case, termination of the state generation is not guaranteed. A simple solution is to set a limit for the number of states, or a limit for the number of parameter variables in a state, and stop if that limit is exceeded.

In future work, we will consider additional mechanisms to improve termination. For instance, we do not actually need parameter variables with types that permit only two values (e.g., open and closed), but could introduce both variants with a bit as a condition. Functional dependencies for derived predicates [6] can be used to prove that only one of several parameter variables will be moved to the next state.

5 Conclusion

Our Datalog-based language Microlog can be used to manage state and state transitions through IO in a declarative fashion. By compilation into C-Code, we generate sources that are compatible with a vast amount of libraries available in the Arduino ecosystem. However, the approach is not limited to the Arduino microcontroller system.

Some programs can be translated to finite state machines where the state transitions of our logic program are precalculated, as we can deduce the behavior for every environment for every point in time. This is particularly important because we then have a limit for the required memory. In systems with e.g., 1 KB
(1) STATES = \{seed(S_0)\};
(2) DONE = \emptyset;
(3) while(STATES − DONE ≠ \emptyset)
(4)      choose S ∈ STATES − DONE;
(5)      DONE = DONE ∪ \{S\};
(6)      S' := lfp(\overline{T}_S ∪ \overline{P}_{always});
(7)      gen(S', \emptyset);
(8)      (9) procedure gen(S', \Phi):
(10)     S_{\text{poss}} := \{A \mid A ← \varphi \in S' and \Phi \cup \{\varphi\} \text{ is consistent}\};
(11)     S_{\text{true}} := \{A \mid A ← \varphi \in S' and \Phi \vdash \varphi\};
(12)     if(S_{\text{true}} = S_{\text{poss}})
(13)     then STATES = STATES ∪ \{seed(S_{\text{true}})\};
(14)     else
(15)      Choose conditional fact A ← \varphi_1 ∧ \cdots ∧ \varphi_n \in S'
(16)      and i ∈ \{1, \ldots, n\}
(17)      such that \Phi \cup \{\varphi_1, \ldots, \varphi_n\} is consistent and
(18)      \Phi \not\vdash \varphi_i;
(19)      gen(S', \Phi \cup \{\varphi_i\});
(20)      gen(S', \Phi \cup \{\neg \varphi_i\});

Fig. 4. Computation of States

RAM, it is obvious that the correctness of a program depends also on its ability
to run with these very limited resources. The result is a Finite State Machine
when the behaviors depend only on values from the environment that do not lie
arbitrarily long in the past.

The presented approach for compilation can also handle the more general
case, where behaviors do depend on values from environment states long past,
but only on a bounded number of those. Then the generated states have data
values as parameters. Not all Microlog-programs can be translated in this way.
In some cases states are being generated with more and more parameters.

It is possible to allow also time-stratified negation as in [2] (i.e. negation is
stratified when only deductive rules are considered). For the standard Datalog
semantics, we would then use the well-founded model instead of the minimal
model. For the conditional facts, we would delay also negative body literals as
in [3], which can later be evaluated by positive and negative reduction as the
other conditions are eliminated by the case analysis.

Our compiler already supports program transformation for some classes of
Microlog programs, as well as the naive evaluation, and we have developed ex-
ample programs for Arduino and the LEGO EV3 robotics platform showing the
viability of our approach. The compiler and example programs are available at
https://dbs.informatik.uni-halle.de/microlog/.
References


Pointer Data Structure Synthesis from Answer Set Programming Specifications

Sarat Chandra Varanasi, Neeraj Mittal, and Gopal Gupta

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

Abstract. We develop an inductive proof-technique to generate imperative programs for pointer data structures from behavioural specifications expressed in the Answer Set Programming (ASP) formalism. ASP is a non-monotonic logic based formalism that employs negation-as-failure which helps emulate the human thought process, allowing domain experts to model desired system behaviour elegantly and succinctly. We argue in this paper that ASP’s reliance on negation-as-failure makes it a better formalism than those based on first-order logic for writing formal specifications. We assume the domain expert provides the representation of inductively defined data structures along with a specification of its operations in ASP. Our procedures combined with our novel proof-technique reason over the specifications and automatically generate an imperative program. Our proof-technique leverages the idea of partial deduction to simplify logical specifications. By algebraically simplifying logical specifications written in ASP, we arrive at a residual specification which can be interpreted as an appropriate imperative program. This work is in the realm of constructing programs that are correct according to a given specification.

1 Introduction

Declarative specifications allow a domain expert to model properties and behaviours of a system precisely. When specifications are executable, the expert can often test the specification quickly and refine them when necessary. Executable specifications are often slow computationally due to the non-deterministic search involved. Our work extracts deterministic imperative programs from executable specifications written in Answer Set Programming (ASP) by means of a novel proof-technique. Our method applies to tree-based pointer data structures with key-ordering implementing common set abstractions such as insert and delete operations. We are interested in these operations in order to eventually automatically generate concurrent data structures. Deriving concurrent data structures requires an understanding of how the sequential imperative program is constructed. We achieve synthesis of the sequential imperative programs for insert, delete of linked lists and external BSTs from their declarative ASP encoding. Answer Set Programming [6] is a logical formalism based on non-monotonic
logic and allows for elegant and succinct specifications of complex problems. The ASP formalism can specify a wide range of computational problems such as constrained search, planning and so on. Its primary use is in artificial intelligence to perform commonsense reasoning but can also express complex ideas involving dynamic properties of a system [4, 14, 2]. Our work is to be contrasted with program verification and other approaches for program synthesis that are based on constraint solving [8, 22]. Program verification using constraint solving involves reducing an imperative program to a set of constraints and proving that the constraints are satisfiable. This often involves additional logical formalisms to be encoded in the constraint solving system. For example, to verify a linked list program in practice, separation logic [21, 19] has to be used which is a formalism extending Hoare-logic [9]. On the other hand, current state of the art program synthesis techniques generates syntactic programs and verify them given an Input-Output specification. Our work varies from both these approaches. We assume that a domain expert provides a declarative behavioural specification of a pointer data structure written in the convenient notation of ASP from which an (efficient) imperative program is synthesized. The behavioural specification contains axioms for representation, properties and primitive operations of the data structure. Representation encodes the domains involved in describing the data structure and its associated well-formedness conditions. Properties encode abstractions used in the respective data structures. Common properties for pointer data structures include reachability of nodes, relationships between nodes present in the data structure and so on. Primitive operations are the permitted read and write operations that can perform operations on only a part of the data structure. The program generated is correct-by-construction as in the case of program synthesis and is performed without the need for additional logics. The resultant program is nothing but a composition of the allowed primitive operations. Our approach presents an inductive proof-technique based on partial deduction [11] of logic programs, to generate imperative programs.

2 Background

Answer Set Programming: Answer Set Programming is very similar to first-order logic except that the deductions performed may be non-monotonic. Conclusions that were inferred earlier may be retracted in light of new information. Answer Set Programming has both model-theoretic [7] and proof-theoretic semantics [1]. A typical Answer Set Program is a collection of rules of the form:

1. \( p \leftarrow \)
2. \( p \leftarrow q_1, q_2, \ldots, q_m, \text{not } r_1, \text{not } r_2, \ldots, \text{not } r_n \{m \geq 0, n \geq 0\} \)
3. \( \leftarrow p', q', \ldots, r' \).

The first rule form depicts raw facts that are unconditionally true. The second rule is the implication used in Prolog i.e. \( p \) is true if the literals in the body of \( p \) are true. For the second rule form, the literals \( q_1, q_2, \ldots, q_m \) constitute the positive literals in the body of \( p \) and likewise, the literals \( r_1, r_2, \ldots, r_n \) denote the negative literals. Negated literals take a special place in ASP. Negated literals prefixed with \( \text{not} \), stand for negation-as-failure (NAF) as opposed to classical
negation in First Order Logic (FOL). A negative literal has a truth value true if the literal has no proof from the rules specified in a program (or theory). The third rule form represents constraints on the truth of literals. That is, the conjunction of literals in the constraint, as in rule form (3) cannot be simultaneously true. An answer set or a model $M$ of a normal logic program $\Pi$ is a set of satisfiable literals in the program under the negation-as-failure semantics. An answer set program $\Pi$ may have more than one model which gives rise to the non-monotonicity of its inferences. The set of models $M_\Pi$ of $\Pi$ are commonly referred to as Stable models and the corresponding semantics of ASP is called the Stable Model Semantics. Details of ASP and ASP solvers can be found elsewhere [6, 5, 1].

**Partial Deduction of Logic Programs:** Logic programs represent both a first order theory and computational artifacts where a computation begins with an issued query [11]. For a rule $p \leftarrow q$, proposition $p$ is inferred if $q$ is true. This corresponds to standard implication in Logic. Computationally, a query $? - p$ triggers a recursive top-down search of the rules defining $p$ and succeeds if the search terminates at a fact. This is widely known as the SLD-tree expansion of a query (goal) in Logic Programming [15]. The SLD-computation and the logical deduction of inferring $p$ from $q$ are equivalent. As an application in partial evaluation, the top-down search of SLD-trees can be directed to explore only part of the search space while selectively not evaluating another part of the search space. For example, for the rule $\{ p \leftarrow q, r \}$ and the query $\{ ? - p \}$, $q$ may have a proof but $r$ may not. Assuming the definition of $r$ may be incomplete, the rule for $p$ may be re-written as $p \leftarrow r$. The re-written rule is a simplification of the original rule and is referred to as “residual code”. One can view partial evaluation as performing deduction up to known information. Hence this process of partial evaluation is also termed partial deduction.

**Key Contributions:** Our main contribution of the paper is to perform a proof by mathematical (structural) induction to verify algebraic operations for inductively defined data structures such as linked lists and binary search trees. By having a domain expert model the algebraic operations as a planning problem [13] in ASP, we are able to verify that the operations are satisfiable for a countably infinite instances of the data structure. This proof, in turn gives rise to stable models that can be exhibit isomorphic computational steps involving some assumed primitive steps (provided by the domain expert). We give a general overview of our technique in the next section.

### 3 General Proof Technique towards Synthesis

Our general technique consists of a proof by mathematical induction on some computation on an inductively defined tree-based pointer data structure. We assume there exists an inductive definition, $\Delta$, defining the data structure. An example recursive definition of a linked-list is given below:
consider a linked-list that is fully unfolded versus a list that is partially unfolded. Note that $\delta'_{b+1} \in S_{\Delta}$ but is consistent with the definition of $\Delta$.

The bottom list is the partially unfolded list. Partial list is achieved by suspending the unfolding of the list predicate. We suspend the unfolding of list just before generating a list instance $\delta_{b+1} \in S_{\Delta}$. The nodes involved in the suspended definition constitute the set of suspended nodes in $\Pi_{\sigma}$. Partial Instances enable Partial Deduction: Consider the partial instance $\delta'_{b+1}$ for some $\Delta$. We show that if $\text{goal}$ is satisfied for the partial instance $\delta'_{b+1}$ then $\text{goal}$ is satisfiable for all instances $\delta_{b+i} \in S_{\Delta}, i = 1, 2, \ldots$. The satisfiability check performed for the instance $\delta'_{b+1}$ represents partial deduction of $\Pi_{\sigma}$ with respect to $\delta'_{b+1}$. This partial deduction step relies on certain assumptions about the algebraic operation $\sigma$ itself: the algebraic operation can be performed by manipulating only part of the data structure instance. Consider linked list insert operation, only two nodes where the target key needs to be inserted are ever modified. This is true for every instance of a linked list. We claim that this is the same as $\text{goal}$ succeeding for the partial instance. This is in turn equivalent to performing the inductive step in an inductive proof. We also show that the partial deduction performed is equivalent to complete deduction. That is, the satisfiability check of $\text{goal}$ for instance $\delta'_{b+1}$ preserves satisfiability for $\delta_{b+1}$.

Inductive Verification yields Imperative Code: Consider the sorted linked-list insert imperative program from an undergraduate classroom exercise. For a node to be inserted into the list with target key $k$ and nodes $x,y$ reachable in the list such that $k_x < k < k_y$, the imperative code is just two pointer linkages. Assume that $x.next = y$ denotes the edge from $x$ to $y$. Then, the two pointer linkages are just the steps: (1) $x.next := \text{target}$ and (2) $\text{target}.next := y$.

The state-of-the-art approaches to synthesize destructive update code for pointer programs uses Separation Logic, a popular formalism for pointer data structures [20]. Our approach, can synthesize the destructive update as a side-effect of the inductive verification step. While checking satisfiability of $\Pi_{\sigma,\Delta}$ for instances $\delta_{b}$ and $\delta'_{b+1}$, the corresponding logic programs generate models. The literals in the models represent the necessary destructive update pointer linkages required in order to satisfy $\sigma$. Let $\Pi_{\delta_{b}}$ and $\Pi_{\delta'_{b+1}}$ denote the answer set programs for $\delta_{b}$ and $\delta'_{b+1}$ respectively. Let the respective models be $\mathcal{M}_{b}$ and $\mathcal{M}'_{b+1}$. If there is a unification of literals between $\mathcal{M}_{b}$ and $\mathcal{M}'_{b+1}$, then there is a unifying property that holds for a countably infinite instances of $\Delta$. Because the unifying property contains the literal $\text{goal}$, the unification captures the same computation from the initial state to final state. Therefore, the literals represent the set of destructive pointer linkages for a countably infinite instances of $\Delta$. In other words, the imperative program to satisfy $\text{goal}$ using pointer linkages is synthesized. For illustration, consider linked-list definition $\text{list}$ modelling the insert operation. Then, $\delta_{b}$ is the
consider a linked-list that is fully unfolded versus a list that is partially unfolded. Note that $\delta_{b+1}^{t} \notin S_{\Delta}$ but is consistent with the definition of $\Delta$.

\[
\begin{array}{c}
\text{h} & \text{a} & \text{t} \\
\text{k}_h & \bullet & \text{k}_a & \bullet & \text{k}_t \\
\end{array}
\]

\[
\begin{array}{c}
\text{h} & \text{a} & \text{b} \\
\text{k}_h & \bullet & \text{k}_a & \bullet & \text{k}_b & \bullet \\
\end{array}
\]

Fig. 1: Linked List Fully Unfolded (top), Partially unfolded (bottom)

The bottom list is the partially unfolded list. Partial list is achieved by suspending the unfolding of the list predicate. We suspend the unfolding of list just before generating a list instance $\delta_{b+1} \in S_{\Delta}$. The nodes involved in the suspended definition constitute the set of suspended nodes in $\Pi$.

**Partial Instances enable Partial Deduction:** Consider the partial instance $\delta_{b+1}^{t}$ for some $\Delta$. We show that if goal is satisfied for the partial instance $\delta_{b+1}^{t}$ then goal is satisfiable for all instances $\delta_{b+i} \in S_{\Delta}$, $i = 1, 2, \ldots$. The satisfiability check performed for the instance $\delta_{b+1}^{t}$ represents partial deduction of $\Pi_{\sigma}$ with respect to $\delta_{b+1}^{t}$. This partial deduction step relies on certain assumptions about the algebraic operation $\sigma$ itself: the algebraic operation can be performed by manipulating only part of the data structure instance. Consider linked list insert operation, only two nodes where the target key needs to be insert are ever modified. This is true for every instance of a linked list. We claim that this is the same as goal succeeding for the partial instance. This is in turn equivalent to performing the inductive step in an inductive proof. We also show that the partial deduction performed is equivalent to complete deduction. That is, the satisfiability check of goal for instance $\delta_{b+1}^{t}$ preserves satisfiability for $\delta_{b+1}^{t}$.

**Inductive Verification yields Imperative Code:** Consider the sorted linked-list insert imperative program from an undergraduate classroom exercise. For a node to be inserted into the list with target key $k$ and nodes $x, y$ reachable in the list such that $kx < k < ky$, the imperative code is just two pointer linkages. Assume that $x.next = y$ denotes the edge from $x$ to $y$. Then, the two pointer linkages are just the steps: (1) $x.next := \text{target}$ and (2) $\text{target.next} := y$.

The state-of-the-art approaches to synthesize destructive update code for pointer programs uses Separation Logic, a popular formalism for pointer data structures [20]. Our approach, can synthesize the destructive update as a side-effect of the inductive verification step. While checking satisfiability of $\Pi_{\sigma}$, for instances $\delta_{b}$ and $\delta_{b+1}^{t}$, the corresponding logic programs generate models. The literals in the models represent the necessary destructive update pointer linkages required in order to satisfy $\sigma$. Let $\Pi_{\delta_{b}}$ and $\Pi_{\delta_{b+1}^{t}}$ denote the answer set programs for $\delta_{b}$ and $\delta_{b+1}^{t}$ respectively. Let the respective models be $M_{b}$ and $M_{b+1}^{t}$. If there is a unification of literals between $M_{b}$ and $M_{b+1}^{t}$, then there is a unifying property that holds for a countably infinite instances of $\Delta$. Because the unifying property contains the literal goal, the unification captures the same computation from the initial state to final state. Therefore, the literals represent the set of destructive pointer linkages for a countably infinite instances of $\Delta$. In other words, the imperative program to satisfy $\sigma$ using pointer linkages is synthesized. For illustration, consider linked-list definition list modelling the insert operation. Then, $\delta_{b}$ is the
single edge \( \{ \text{edge}(h, t) \} \). \( \delta_{b+1}' \) is the set of edges for the partial instance consisting of \( \{ \text{edge}(h, x), \text{edge}(x, y) \} \). Consider the relation \( \text{link}(x, y) \) that represents the logical equivalent of \( \text{x.next} := y \). If a target node \( \tau \) to be inserted satisfies the key constraints, then the models of \( \Pi_{\delta_b} \) and \( \Pi_{\delta_{b+1}}' \) will respectively contain \( M_b = \{ \text{link}(h, \tau), \text{link}(\tau, t) \} \) and \( M_{b+1} = \{ \text{link}(x, \tau), \text{link}(\tau, y) \} \). The link operations are precisely the equivalent pointer linkages in an imperative program. We illustrate the linked-list world further in the next section. The technique can be generalized to arbitrary pointer data structures where the algebraic operations modify only a fragment of the data structure instances.

4 Example for Linked List Insert, Delete Operation

Notation: Linked-lists are specified using the definitions \( \text{Base}, \text{Suffix} \) as before. Logic program variables \( X, Y, Z, \ldots \) are used to represent nodes, \( K, K_1, K_2, \ldots \) are used to represent keys, \( T \) is invariably used to describe the transition from initial state to final state. \( T = 0 \) represents initial state of the world, \( T = 1 \) represents the final state.

Initial State: The initial state captures the input data for the answer set program \( \Pi_{\tau} \). To distinguish between initial and final state we use \( T = 0, 1 \) respectively. The predicates of \( \text{list} \), and \( \text{edge}(X, Y) \) are extended with a \( T \) argument. Similarly, other definitions which are fluents are extended into the time domain. An example initial state is given below:

| Edges: \{ \text{edge}(h, x), \text{edge}(x, t) \} | Keys: \{ \text{key}(h, k_h), \text{key}(x, k_x), \text{key}(t, k_t) \} |
| Key order: \{ l(t(k_h, k_x), l(k_x, k_t) \} | Target node, key: \text{key}(\tau, k_\tau) |

Definitions: At this point, we need to encode basic definitions involved in the Linked-List Theory. These are standard definitions for reachability of nodes and the definitions of a key being present in the list.

\[
\begin{align*}
\text{reach}(h, T) & \leftarrow \quad \text{(Reachable-Head)} \\
\text{reach}(X, T) & \leftarrow \text{edge}(Y, X, T), \text{reach}(Y, T) \quad \text{(Reachable-Recursive)} \\
\text{present}(K, T) & \leftarrow \text{key}(X, K), \text{reach}(X, T) \quad \text{(Key-Present)}
\end{align*}
\]

Final State(s): To model insert operation \( \Pi_{\text{insert}} \) would contain \( \text{goal_insert} \) predicate. Similarly to model delete operation, \( \Pi_{\text{delete}} \) would contain \( \text{goal_delete} \) predicate.

\[
\begin{align*}
\text{goal_insert} & \leftarrow \neg \text{present}(k_\tau, 0), \text{present}(k_\tau, 1) \quad \text{(Insert)} \\
\text{goal_delete} & \leftarrow \text{present}(k_\tau, 0), \neg \text{present}(k_\tau, 1) \quad \text{(Delete)}
\end{align*}
\]

Pointer Linkage Operation and its Effects: The only destructive update primitive operation involved in linked-list insert (delete) operation is that of pointer linkage. As mentioned in the previous section, we let \( \text{link}(X, Y) \) denote the imperative equivalent of \( \text{x.next} := y \). The \( \text{link} \) operation is abduced (guessed) and then checked against constraints imposed by the insert operation. The effect of \( \text{link}(X, Y) \) is to create the edge from \( X \) to \( Y \). The rules \text{Modified} and \text{Inertia} together state what does not change in the computation when a node is left untouched.
Proof. This is shown below.

Therefore, from the isomorphism, we have both the imperative program and the model. We denote this unification as the generic one.

Finding the correspondence involves starting with the nodes that are modified in the base case model and finding an isomorphism between the modified nodes such that the literals associated with modified nodes using $\Pi$ (generic) can be translated into literals in the inductive case. The minimization involves constructing a proof-tree for every literal. For example, to query the system and extract the nodes in the proof-tree that are not children of other literals in the proof-tree. The construction of the proof-tree is provided by the Proof-Theoretic s(CASP) system [1].

The minimization involves constructing a proof-tree for every literal to extract the nodes in the proof-tree that are not children of other literals. The construction of the proof-tree is provided by the Proof-Theoretic s(CASP) system [1].

State Constraints and Invariants: We restrict the effects of $\text{link}(X,Y)$ abducible to reject self-loop edges, and spurious edges that may be created in the computation. Also, the list has to be well-formed at all times. Further, insert operation should not lose any keys that are already present in the list. Similarly, delete operation should not lose any keys other than the target key. Finally, suspended nodes are left untouched in the computation. The role of suspended nodes is made clear in the correctness proof.

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\leftarrow \text{edge}(X,Y,T), \text{edge}(Y,X,T)$</td>
<td>(No-Self-Loop)</td>
</tr>
<tr>
<td>$\leftarrow \text{edge}(X,h,T)$</td>
<td>(Not-Before-Head)</td>
</tr>
<tr>
<td>$\leftarrow \text{edge}(t,X,T)$</td>
<td>(Not-Beyond-Tail)</td>
</tr>
<tr>
<td>$\leftarrow \text{not list}(T)$</td>
<td>(List-Invariant)</td>
</tr>
<tr>
<td>$\leftarrow \text{goal-insert}, \text{present}(K,0), \text{not present}(K,1)$</td>
<td>(No-Key-Loss)</td>
</tr>
<tr>
<td>$\leftarrow \text{goal-delete}, K \neq \tau, \text{present}(K,0), \text{not present}(K,1)$</td>
<td>(No-Key-Loss-Delete)</td>
</tr>
<tr>
<td>$\leftarrow \text{suspended}(X), \text{modified}(X)$</td>
<td>(Suspended-Unmod)</td>
</tr>
</tbody>
</table>

5 Synthesizing Code for Insert, Delete Operation

We have described the general theory of linked-lists ($\Delta$) with rules to specify computation of insert and delete operations. Let the corresponding ASP programs be $\Pi_{\text{insert}}$ and $\Pi_{\text{delete}}$. Now we need to search for the smallest instance in $S_\Delta$ for satisfiability. We systematically generate instances $\delta_0, \delta_1, \ldots$ until $\text{goal-insert}$ ($\text{goal-delete}$) is satisfied. Let this instance be $\delta_b$. Once we have $\delta_b$, we perform the unfolding and suspension step. Let this instance be denoted $\delta'_{b+1}$. We again check the satisfiability of $\Pi_{\text{insert}}$ ($\Pi_{\text{delete}}$). If the program is satisfiable, then we look for unifications in the models for $\delta_b$ and $\delta'_{b+1}$. The input data and the models for the base case, inductive case for insert, delete operation are shown in Table 1. By obtaining models for both base case and inductive case, we have proved the satisfiability of $\text{goal-insert}$ and $\text{goal-delete}$ for every instance in $S_\Delta$ from the base case instance $\delta_0$. We prove the correctness of this claim in the formalization section.

Model Correspondence between Base Case and Inductive Case: Because the link actions for both base case and inductive case are isomorphic, we can find a correspondence between the nodes in the base case and the inductive case. The entire model for base case and inductive case in case of linked list insert operation is shown in Table 2. It is clear that there is a correspondence between literals involving nodes that were modified in the computation.
Pointer Data Structure Synthesis from ASP Specifications

for Linked List operations in this paper. An example for External BST insert operation synthesis can be found elsewhere [23].

6 Synthesis of Data-Structure Traversal Code

Traversal code for an arbitrary data structure allows for reading a set of nodes into memory that satisfy a given condition. Synthesizing efficient traversal code for data-structures is non-trivial. We outline two approaches. From the destructive update synthesis, we derived the precondition necessary to perform the algebraic operation. In case of list insert operation, the precondition

\[
\text{pre}(x,y) \equiv \{ \text{reach}(x), \text{suffix}(y) \}. \]

The synthesized traversal code should visit the least number of nodes while finding the nodes that satisfy \text{pre}.

Fully Automatic Method:
The precondition \text{pre}(x,y) can be viewed as a datalog query. There are several tools that synthesize imperative code from datalog queries. Notable framework among them is Souffl{é} [10]. The program \(\Pi_{\text{insert}}\) without the pointer linkage operations, constraints and goal predicate is a datalog program without negation. This datalog program can be used as a background theory to generate the imperative code for the query \text{pre}(x,y). The translated code would be inefficient in the number of nodes visited as it performs the search viewing the data-structure relations as a database.

Semi-Automatic Method:
We assume that the domain expert provides a recursive traversal relation to reach the target key for insert (delete) operation. One such traverse relation looks as follows:

\[
\text{traverse}(X) \leftarrow \text{pre}(X,Y). \quad \text{(Trv-Base)}
\]

\[
\text{traverse}(X) \leftarrow \text{edge}(X,Y), \text{reach}(X), \text{suffix}(Y), \neg \text{pre}(X,Y), \text{traverse}(Y). \quad \text{(Trv-Ind)}
\]

\[
\text{start} \leftarrow \text{traverse}(h). \quad \text{(Start-Traversal)}
\]

Given the explicit information from the domain expert about where to begin the traversal from and how to visit nodes in succession makes the synthesis task trivial. All we need to do is verify that for an arbitrary target key \(k_\tau\), the predicate \text{start} succeeds for \(\delta_b\) and \(\delta'_b + 1\). Once the verification step is done, we need to translate the recursive definition of \text{traverse} into imperative code. A straightforward translation looks as follows:

\[
\text{function traverse }(x) \{
\text{if} ( \text{reach}(x) \land \text{suffix}(x.\text{next}) \land x.\text{key} < \text{target}.\text{key} \land \text{target}.\text{key} < x.\text{next}.\text{key} ) \{
\text{return } (x,x.\text{next})
\}
\text{if}( \text{reach}(x) \land \text{suffix}(x.\text{next})) \{
\text{// assumed that pre}(x,x.\text{next}) \text{ is not satisfied}
\text{return traverse}(x.\text{next})
\}
\}
\]

However, the above code is still inefficient. We have not shown the definition of \text{reach}(x), \text{suffix}(x.\text{next}) in an imperative setting. We could again use a

\[
\delta_b : \text{time}(0) \text{ time}(1) \text{ node}(h) \text{ node}(t) \text{ node}(\tau) \text{ key}(h,k_h) \text{ key}(t,k_t) \text{ key}(\tau,k_\tau) \text{ lt}(k_h,k_t) \text{ lt}(k_t,k_\tau) \text{ reach}(h,0) \text{ reach}(t,0) \text{ reach}(\tau,1) \text{ reach}(t,1) \text{ present}(k_t,0) \text{ present}(k_\tau,0) \text{ present}(k_\tau,1) \text{ present}(k_\tau,1) \text{ edge}(h,t) \text{ edge}(t,h) \text{ edge}(\tau,1) \text{ edge}(\tau,1) \text{ list}(0) \text{ list}(1) \text{ suffix}(t,0) \text{ suffix}(t,1) \text{ suffix}(\tau,1) \text{ modified}(h) \text{ modified}(\tau) \text{ link}(\tau,t) \text{ link}(h,\tau) \text{ goal } \text{insert}
\]

\[
\delta_{b+1} : \text{time}(0) \text{ time}(1) \text{ node}(h) \text{ node}(x) \text{ node}(y) \text{ node}(\tau) \text{ key}(h,k_h) \text{ key}(x,k_x) \text{ key}(y,k_y) \text{ key}(\tau,k_\tau) \text{ lt}(k_h,k_t) \text{ lt}(k_t,k_x) \text{ lt}(k_x,k_\tau) \text{ reach}(h,0) \text{ reach}(x,0) \text{ reach}(y,0) \text{ reach}(h,1) \text{ reach}(x,1) \text{ reach}(y,1) \text{ reach}(\tau,1) \text{ reach}(\tau,1) \text{ present}(k_t,0) \text{ present}(k_\tau,0) \text{ present}(k_\tau,1) \text{ present}(k_\tau,1) \text{ present}(k_\tau,1) \text{ edge}(h,x) \text{ edge}(x,y) \text{ edge}(h,x,0) \text{ edge}(y,0) \text{ edge}(h,x,1) \text{ edge}(\tau,1) \text{ edge}(\tau,1) \text{ list}(0) \text{ list}(1) \text{ suffix}(x,0) \text{ suffix}(x,1) \text{ suffix}(y,0) \text{ suffix}(y,1) \text{ suffix}(\tau,1) \text{ modified}(x) \text{ modified}(\tau) \text{ link}(\tau,y) \text{ link}(x,\tau) \text{ goal } \text{insert} \text{ suspended}(y)
\]

Finding the correspondence involves starting with the nodes that are modified and finding an isomorphism between the modified nodes such that the literals in the base case model are interchangeable with the literals in the inductive case model. We denote this unification as the \textit{generic} model of \(\Pi_{\text{insert}}\) \(\Pi_{\text{delete}}\). Therefore, from the isomorphism, we have both the imperative program and proof. This is shown below\footnote{comma represents conjunction}:

\[
\{ \text{reach}(x), \text{suffix}(y), \text{edge}(x,y), \text{lt}(k_x,k_y), \text{lt}(k_\tau,k_y) \}
\]

\[
\text{link}(x,\tau)
\]

\[
\text{link}(\tau,y)
\]

\[
\{ \text{goal } \text{insert} \}
\]

Computing the above minimal precondition literals from the generic model requires some work. We need to minimize the literals in the generic model given \(\Pi_{\text{insert}}\) \(\Pi_{\text{delete}}\). The minimization involves constructing a proof-tree for every literal associated with modified nodes using \(\Pi_{\text{insert}}\) and picking only the literals that are not derived from other literals in the proof-tree. The construction of proof-tree is provided by the Proof-Theoretic s(CASP) system [1]. We simply query the system and extract the nodes in the proof-tree that are not children of other precondition literals. For example, \{reach(y)\} is not in the precondition as it is derived from \{reach(x), edge(x,y)\}. We have shown the synthesis

\footnote{comma represents conjunction}
for Linked List operations in this paper. An example for External BST insert operation synthesis can be found elsewhere [23].

6 Synthesis of Data-Structure Traversal Code

Traversal code for an arbitrary data structure allows for reading a set of nodes into memory that satisfy a given condition. Synthesizing efficient traversal code for data-structures is non-trivial. We outline two approaches. From the destructive update synthesis, we derived the precondition necessary to perform the algebraic operation. In case of list insert operation, the precondition \(pre(x, y) \equiv \{reach(x), suffix(y), edge(x, y), lt(kx, ky), lt(kx, ky)\}\). The synthesized traversal code should visit the least number of nodes while finding the nodes that satisfy \(pre\).

**Fully Automatic Method:** The precondition \(pre(x, y)\) can be viewed as a datalog query. There are several tools that synthesize imperative code from datalog queries. Notable framework among them is Soufflé [10]. The program \(\Pi_{\text{insert}}\) without the pointer linkage operations, constraints and \(goal\) predicate is a datalog program without negation. This datalog program can be used as a background theory to generate the imperative code for the query \(pre(x, y)\). The translated code would be inefficient in the number of nodes visited as it performs the search viewing the data-structure relations as a database.

**Semi-Automatic Method:** We assume that the domain expert provides a recursive traversal relation to reach the target key for insert (delete) operation. One such traverse relation looks as follows:

\[
\text{traverse}(X) \leftarrow pre(X, Y). \quad \text{(Trv-Base)}
\]

\[
\text{traverse}(X) \leftarrow edge(X, Y), reach(X), suffix(Y), \text{not pre}(X, Y), traverse(Y). \quad \text{(Trv-Ind)}
\]

\[
\text{start} \leftarrow \text{traverse}(h). \quad \text{(Start-Traversal)}
\]

Given the explicit information from the domain expert about where to begin the traversal from and how to visit nodes in succession makes the synthesis task trivial. All we need to do is verify that for an arbitrary target key \(k_t\), the predicate \(start\) succeeds for \(d_0\) and \(d_{0+1}\). Once the verification step is done, we need to translate the recursive definition of \(traverse\) into imperative code. A straightforward translation looks as follows:

```plaintext
function traverse(x) {
    if (reach(x) && suffix(x.next) && x.key < target.key && target.
        key < x.next.key) {
        return (x, x.next)
    }

    if(reach(x) && suffix(x.next)){
        // assumed that pre(x,x.next) is not satisfied
        return traverse(x.next)
    }
}
```

However, the above code is still inefficient. We have not shown the definition of \(reach(x), suffix(x.next)\) in an imperative setting. We could again use a
datalog synthesis as mentioned before. But would still amount to lots of redundant computation. To remove this inefficiency, we “compile-away” the predicates reach(x), suffix(x.next). This can be done by exploring the proof-tree from the goal-directed s(CASP) system. Consider the proof-tree as shown by the s(CASP) system for the query ?- start for the inductive case of list (shown below). It is clear from the proof tree that the conjunction \{reach(x), edge(x, y), suffix(y)\} is an invariant in the traversal computation. This invariant also holds in the base case (not shown). Therefore, the predicates \{reach(x), suffix(y)\} can be compiled away. However, we need to keep the primitive \{edge(x, y)\} as it is used further in the computation of traverse(y).

**Predicting the Invariant:** As discussed above, it is possible that part of the conjuncts in pre(x, y) could be an invariant. We search for the maximal conjunct in pre(x, y) that is true in the recursive definition of traverse(x). To do this, we can define multiple traversal predicates each varying in the number of conjuncts in pre(x, y). If traverse is succeeds for both base case and inductive case, then the invariant can be compiled away in the residual code.

```prolog
start :-
  traverse(h) :-
    reach(h),
    edge(h, x),
    suffix(x),
    not_eq_node(x, y),
    key(x, x),
    key(y, x),
    key(x, y),
    lt(kx, ky),
    suffix(y),
    not pre(h, x),
  traverse(h) :-
    pre(x, y),
    proved(edge(x, y)),
    reach(x),
    proved(reach(h)),
    proved(suffix(x)),
    key(x, x),
    key(y, x),
    lt(kx, ktarget),
    lt(ktarget, ky).
```

Therefore, the simplified imperative code now looks like:

```prolog
function traverse(x){
  if (x.key < target.key && target.key < x.next.key){
    return (x, x.next)
  }
  return traverse(x.next)
}
```

7 Formalizing the synthesis procedure and Assumptions

7.1 Class of Data Structures Assumed

**Definition** Heap H is a set of nodes, L is a universe of labels. E is a set of directed edges between nodes, labelled by elements from L. More precisely, the set of edges E is a relation over H × L × H. No node can have more than one edge with the same label to another node. That is,
∀ n₁, n₂, n₁′, n₂′, ∀ ℓ : (n₁, ℓ, n₂) ∈ E ∧ (n₁′, ℓ, n₂′) ∈ E ⇒ n₂ = n₂′

**Definition** Let root denote a distinguished node in the Heap. A (tree) pointer data structure is a two-place relation D ⊆ {root} × ℙ(E). For our work, the relation should describe a tree-structure inductively. If the relation is list, then D² represents a linked list.

**Definition** A primitive step s : H × H × L × ℙ(E) → ℙ(E) is a function that links two nodes using a label. Its definition is as follows: s(x, y, ℓ, e) = (e \ ((x, ℓ, y') ∈ e) \ { (x, ℓ, y)}

**Definition** An algebraic operation σ_D : H × {root} × ℙ(E) → ℙ(E)_⊥ of a data structure D is a mapping such that

σ_D(x, root, e) = \begin{cases} e' & \text{if Input}(x) \land D(root, e) \land D(root, e') \land Constraints(root, e, e') \\ \bot & \text{otherwise} \end{cases}

where Constraints imposes the input-output relationship between e and e' and Input checks whether the input is well-formed.

**Definition** We say that σ_D is computable in n primitive steps if

∀ x ∈ σ_D(x, root, e) = e' \Rightarrow

∃ x₁, ..., x_n, y₁, ..., y_n, ℓ₁, ..., ℓ_n, s(x_n, y_n, ℓ_n, s(x_{n-1}, y_{n-1}, ℓ_{n-1}, s(., s(x₁, y₁, ℓ₁, e.), .))) = e'

We denote this relation as C_n = \{ σ_D : σ_D is computable in n primitive steps \}

We consider algebraic operations from the class C_n in this paper. For example, insertList ∈ C_2 and deletebst \footnote{list is synonymous with Linked Lists, cbst with External BSTs} ∈ C_1.

### 7.2 Computation Correspondence

We rely on the notion of having an isomorphism between the base case model and the inductive case model to extract an imperative program. The following definitions capture this precisely. Let M be an answer set of a normal logic program Π. Then, we define A_M to be the set of all actions present in M. Literals of a model M are all propositions that are true in M. A model M can be treated as a set.

**Definition** Let M₁, M₂ be answer sets of some program Π and A_M₁, A_M₂, their destructive update steps. We say that M₁ embeds (⇝) into M₂ if there is a structure preserving map of nodes in M₁ to nodes in M₂. Let N_M₁ and N_M₂ denote the nodes of models M₁ and M₂ respectively.

**Definition** An embedding f : N_M₁ ⇝ N_M₂ is a mapping such that

∀ lit : lit(x₁, x₂, ..., x_n) ∈ M₁ ⇒ lit(f(x₁), f(x₂), ..., f(x_n)) ∈ M₂

M₁ ⇝ M₂ \iff (∃ f : N_M₁ ⇝ N_M₂) \land (∀ p ∈ null_M₁ ⇒ p ∈ M₂)

where null_M₁ is the set of nullary predicates of Π that are true in M₁.

**Definition** We say that M₁ and M₂ have a computation correspondence (⇝) iff either M₁ embeds into M₂ or vice-versa, and both A_M₁, A_M₂ have the same number of actions (steps).

\footnote{D is independent of whether the definition Δ is recursive or not. They will be equivalent without affecting the formalization, D is used to explicitly model heaps}
The left tree represents the base case. We know that the deletion of node with two non-nil children involves fetching the inorder successor of \( l \) (in this case) and replacing \( l \) with the inorder successor. However, consider the suspended tree (inductive case) on the right. To successfully delete node \( l \) we need to access the inorder successor of \( l \) which is only found in the left-subtree of \( \text{suspended} \). But the sub-tree of \( \text{suspended} \) is inaccessible due to the way our technique handles suspended nodes. Therefore, we cannot use the inorder successor lemma to find the model correspondence between the base case and the inductive case. However, the tree on the right-hand side can still complete the deletion of node \( l \): by letting node \( l' \) take the position of \( l \). Thus, we can verify the inductive step but not synthesize an imperative program.

![Fig. 2: Internal BST: Without suspended nodes (left), With suspended node (right)](image)

### Data Structure Operation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Linked List Insert</th>
<th>Linked List Delete</th>
<th>External BST Insert</th>
<th>External BST Delete</th>
<th>Internal BST Insert</th>
<th>Internal BST Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data structure Traversal (semi-automatic)</th>
<th>Yes</th>
</tr>
</thead>
</table>

| Table 3: Data Structure Operations Verified and Synthesized |

---

**Synthesis Procedure**

**Destructive Update:** The procedure \( \text{Enumerate-And-Check-Base-Case} \) takes the data structure theory with algebraic operation \( \sigma \) and the recursive definition \( \Delta \) and enumerates \( S_\Delta \) starting from \( \delta_0, \delta_1, \ldots \) and so on. If it finds the base case \( \delta_b \) it returns it to its calling procedure along with the program \( \Pi_{\delta_b} \). \( \Pi_{\delta_b} \) simply contains the literals associated with \( \delta_b \) as facts. Then, the unfolding and suspension step is performed. The procedure, \( \text{Unfold-And-Suspend} \) takes in \( \delta_b \) and unfolds the recursive definition partially to generate the partial instance \( \delta_b^{+1} \). Along with \( \delta_b^{+1} \), the program \( \Pi_{\delta_b^{+1}} \) which contains the instance \( \delta_b^{+1} \) and the rewritten rules to account for suspended nodes is returned. We then check the satisfiability of \( \Pi_{\delta_b^{+1}} \). If a correspondence is found as described in Section 7, we transform the correspondence (generic model) into an appropriate imperative notation. At the same time, the precondition is also obtained by inspecting the literals associated with modified nodes in the computation.

**Lemma 1.** Trace correspondence is an equivalence relation.

**Lemma 2.** If \( M_b \sim M_i \), then there exists an imperative program satisfying \( \Pi_\sigma \). **Discussion** The imperative program constructed is the straight line program consisting of the actions satisfying the necessary goal, that can execute in any order. Complexity of computing the embedding between \( M_b \) and \( M_i \) is \( O(|M_{\bar{\sigma}}| \times |M_\sigma| \times L^2) \) where \( |M_{\bar{\sigma}}|, |M_\sigma| \) represent cardinality of sets \( M_{\bar{\sigma}}, M_\sigma \) and \( L^2 \) is the number of mappings between nodes from \( A_{M_\bar{\sigma}} \) and \( A_{M_\sigma} \) respectively belonging to models in \( M_{\bar{\sigma}}, M_\sigma \). \( L = |A_{M_\bar{\sigma}}| \).

**Lemma 3.** Partial deduction is equivalent to complete deduction for tree-based inductive data structures with suspended nodes.

**Incompleteness of Partial Deduction** Our partial deduction procedure is still incomplete for \( \mathcal{C}_n \). Consider the task of last node \( (\text{delete}_\text{last} \text{list}) \) in a linked list. We know that \( \text{delete}_\text{last} \text{list} \in \mathcal{C}_1 \). But suspending \( \text{suffix}(X) \) cannot be used to establish that the last element can be deleted. We require an alternative way to fold/unfold the recursive definition to expose the correct partial instance in order to delete the node. We leave this for future work. Our partial deduction procedure can currently decide the subset of \( \mathcal{C}_n \) with inductive definition suspensions where the algebraic operation can be performed on the visible fragment unfolded in the inductive step.

**Incompleteness of Synthesis** We can generate straight line programs for the subset decidable by partial deduction: Linked Lists, External BSTs. This is due to two reasons. It has to do with the two conjuncts in computation correspondence relation. The first conjunct requires an entire model to embed into another model. This is not the case with Internal BST delete operation. A model of delete operation for a smaller Internal BST does not necessarily embed into the model of delete operation for a larger Internal BST. The second requirement that \( |A_{M_1}| = |A_{M_2}| \) is also strong. This is not the case for reversal of nodes in a linked list where the number of primitive steps is equal to the size of the list. Work is in progress towards relaxing the requirements and generating the mentioned data structure operations. Table 3 shows the the data structure operations that we can verify or synthesize.

**Incompleteness of Synthesis for Internal BST Delete** Consider the deletion of a node in internal bst (Figure 2) where the node to be deleted has two children that are not nil. The node to be deleted is labelled \( l \).
The left tree represents the base case. We know that the deletion of node with two non nil children involves fetching the inorder successor of l (in this case) and replacing l with the inorder successor. However, consider the suspended tree (inductive case) on the right. To successfully delete node l we need to access the inorder successor of l which is only found in the left-subtree of suspended. But the sub-tree of suspended is inaccessible due to the way our technique handles suspended nodes. Therefore, we cannot use the inorder successor lemma to find the model correspondence between the base case and the inductive case. However, the tree on the right-hand side can still complete the deletion of node l: by letting node ll take the position of l. Thus, we can verify the inductive step but not synthesize an imperative program.

![Diagram of trees](image)

**Fig. 2:** Internal BST: Without suspended nodes (left), With suspended node (right)

<table>
<thead>
<tr>
<th>Data Structure Operation</th>
<th>Verify</th>
<th>Synthesize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked List Insert</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Linked List Delete</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>External BST Insert</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>External BST Delete</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Internal BST Insert</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Internal BST Delete</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Data structure Traversal (semi-automatic)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

| Table 3: Data Structure Operations Verified and Synthesized |

8 Synthesis Procedure

**Destructive Update:** The procedure **Enumerate-And-Check-Base-Case** takes the data structure theory with algebraic operation $\sigma$ and the recursive definition $\Delta$ and enumerates $S_\Delta$ starting from $\delta_0, \delta_1, \ldots$ and so on. If it finds the base case $\delta_b$ it returns it to its calling procedure along with the program $\Pi_\delta_b$. $\Pi_\delta_b$ simply contains the literals associated with $\delta_b$ as facts. Then, the unfolding and suspension step is performed. The procedure **Unfold-And-Suspend** takes in $\delta_b$ and unfolds the recursive definition partially to generate the partial instance $\delta'_{b+1}$. Along with $\delta'_{b+1}$, the program $\Pi_{\delta'_{b+1}}$ which contains the instance $\delta'_{b+1}$ and the rewritten rules to account for suspended nodes is returned. We then check the satisfiability of $\Pi_{\delta'_{b+1}}$. If a correspondence is found as described in Section 7, we transform the correspondence (generic model) into an appropriate imperative notation. At the same time, the precondition is also obtained by inspecting the literals associated with modified nodes in the computation. The procedure **Get-**
PRECONDITION also minimizes the literals in the synthesized precondition using a proof-tree analysis as described in Section 5. **Traversals:** We show the semi-automatic method. A predefined traversal relation is assumed to be provided by the domain expert. We guess the invariant by adding partial conjuncts from the precondition to the recursive case of the traversal relation. If the traversal succeeds in both base case and recursive case, then the guessed invariant is correct. Otherwise, we weaken the invariant by dropping more literals and repeat the procedure. In the worst case, there would be no invariant guessed and the synthesized traversal code would use the fully-automatic approach. The function Check-Sat checks satisfiability of a program using standard ASP solvers while Models returns all stable models.

```
1: procedure Synthesize-Destructive-Update(Δ, Π₀)
2:    Input: Δ, inductive definition that can enumerate S_Δ
3:    Input: ASP Encoding of algebraic operation σ_Δ as Π₀
4:    Output: Synthesized Imperative Program
5:    (δ₀, Π₀) ← Enumerate-And-Check-Base-Case(Δ, Π₀)
6:    (δ₀, Π(track)) ← Unfold-And-Suspend(δ₀, Π₀)
7:    if Check-Sat(Π(track)) then
8:        M(track) ← Models(Π(track))
9:        M(track) ← Models(Π(track))
10:       if Exists-Computation-Correspondence(M(track), M(track)) then
11:          M(track) ← Get-Correspondence(M(track), M(track))
12:          pre ← Get-Predicate(M(track))
13:          Code ← Translate-To-Imperative-Primitives(M(track))
14:          return (Code, pre, δ₀)
15:   end if
16: end if
17: end procedure
18: procedure Synthesize-Traversal-Code(Δ, Π, pre, trv, δ₀)
19:    Input: Δ, δ₀ as before, δ₀ is from the destructive update
20:    Π contains definitions associated with Δ in ASP
21:    pre ∈ Π is the synthesized precondition from above procedure
22:    trv ∈ Π is the traversal recursive relation provided by domain expert
23:    Π(track) ← Check-Base-Case(Δ, Π)
24:    (δ₀(track), Π(track)) ← Unfold-And-Suspend(δ₀, Π)
25:    if Check-Sat(Π(track)) then
26:        M(track) ← Models(Π(track))
27:        M(track) ← Models(Π(track))
28:        inv ← Predict-Invariant(M(track), M(track), trv)
29:        Code ← Generate-Recursive-Code(trv, inv)
30:        return Code
31: end if
32: end procedure
```

9 Conclusion and Future Work

We have verified and synthesized a small class of inductive pointer data structure operations in this paper from their ASP specifications. The rules in ASP
encoding of linked lists are written naturally, in almost a commonsensical fashion. Setting aside the superficial burden of not having to write quantifiers, the rules represent succinct definitions of a subject theory. Negative concepts such as unreachability are naturally qualified with NAF. Without using NAF, providing a definition of unreachable is not straightforward, and has to be defined procedurally. Therefore, not relying on hard evidence for negative information makes ASP a better formalism than First-Order Logic, to write specifications. Note that this comes at a price of causality. That is, there has to be a strong causal relation between a rule definition and its consequent. For example, rule \{p ← q\} should mean \{p ↔ q\}. Otherwise, soundness of the judgement through negation-as-failure would be compromised.

**Related Work:** Our work relates with several ideas existing in Program Analyses, Logical Formalisms and Transformation. Program Analyses of Heap manipulating programs have been studied well. Most popular among them is separation logic [19], an extension of First-Order Logic. The state-of-the-art for sequential pointer data-structure synthesis is Synthetic Separation Logic [20]. Other decidable logics include the STRAND [16] logic. STRAND verifies recursive heap manipulating formulas by translating them into a decidable fragment of quantifier free theories and uses a theorem prover. Our approach solely relies on the predicates and lemmas provided by the domain expert. Our eventual aim is to generalize our partial deduction procedure. Another significant framework is the DRYAD [17] logic. The framework works similar to STRAND by generating quantifier free verification conditions for recursive heap-manipulating programs. Their framework relies on unfolding recursively the memory footprint operated by the verified program’s basic blocks. This is quite similar to the visible finite fragment of the data structure necessary to satisfy the algebraic operation for the unfolded and suspended (inductive) case in our partial deduction scheme. However, our approach yet has incompleteness with algebraic operations and recursive definitions. Nonetheless, we believe that a semantics-guided approach to program transformation should be sufficient for the program synthesis problem.

In spirit, our work is closely related to the semantics preserving transformation due to Darlington [3] and extraction of program and proofs from axiomatic descriptions of data structures due to Manna [18]. When compared to Abstract Interpretation, our procedure retains precision when performing partial evaluation (deduction). The idea that Partial Deduction performs an inductive proof is also detailed elsewhere [12]. We have applied it independently in ASP to pointer data structures.

**Future Work:** Our procedure currently generates straight line programs for destructive update and if-else branches only for recursive definitions. Because both the destructive update and traversal part are verified for all instances of the data structure (inductively), termination of the generated program is guaranteed for well-formed inputs. Much work still remains, to generalize to more classes of pointer-data structures such as Internal BSTs and more algebraic operations such as Linked List reversal. Eventually, our goal is to automatically transform the synthesized sequential data structures into concurrent data structures.
References
[16] Parthasarathy Madhusudan, Gennaro Parlato, and Xiaokang Qiu. “Decidable logics combining heap structures and data”. In: Proceedings of the
Generating Functions for Probabilistic Programs

Lutz Klinkenberg, Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Joshua Moerman, and Tobias Winkler

1 RWTH Aachen University, 52062 Aachen, Germany
2 University College London, United Kingdom

Abstract. This paper investigates the usage of generating functions (GFs) encoding measures over the program variables for reasoning about discrete probabilistic programs. To that end, we define a denotational GF-transformer semantics for probabilistic while-programs, and show that it instantiates Kozen’s seminal distribution transformer semantics. We then study the effective usage of GFs for program analysis. We show that finitely expressible GFs enable checking super-invariants by means of computer algebra tools, and that they can be used to determine termination probabilities. The paper concludes by characterizing a class of — possibly infinite-state — programs whose semantics is a rational GF encoding a discrete phase-type distribution.

Keywords: probabilistic programs · quantitative verification · semantics · formal power series.

1 Introduction

Probabilistic programs are sequential programs for which coin flipping is a first-class citizen. They are used e.g. to represent randomized algorithms, probabilistic graphical models such as Bayesian networks, cognitive models, or security protocols. Although probabilistic programs are typically rather small, their analysis is intricate. For instance, approximating expected values of program variables at program termination is as hard as the universal halting problem [19]. Determining higher moments such as variances is even harder. Deductive program verification techniques based on a quantitative version of weakest preconditions [22,17] enable to reason about the outcomes of probabilistic programs, such as what is the probability that a program variable equals a certain value. Dedicated analysis techniques have been developed to e.g., determine tail bounds [6], decide almost-sure termination [23,8], or to compare programs [1].

⋆ This research was funded by the ERC AdG project FRAPPANT (787914) and the DFG RTG 2236 UnRAVeL.

Generating Functions for Probabilistic Programs*

Lutz Klinkenberg\textsuperscript{1}[0000--0002--3812--0572], Kevin Batz\textsuperscript{1}[0000--0001--8705--2564],
Benjamin Lucien Kaminski\textsuperscript{1,2}[0000--0001--5185--2324], Joost-Pieter
Katoen\textsuperscript{1}[0000--0002--6143--1926], Joshua Moerman\textsuperscript{1}[0000--0001--9819--8374], and
Tobias Winkler\textsuperscript{1}[0000--0003--1084--6408]

\textsuperscript{1} RWTH Aachen University, 52062 Aachen, Germany
\textsuperscript{2} University College London, United Kingdom
{lutz.klinkenberg, kevin.batz, benjamin.kaminski, katoen, joshua, tobias.winkler}@cs.rwth-aachen.de

Abstract. This paper investigates the usage of generating functions
(GFs) encoding measures over the program variables for reasoning about
discrete probabilistic programs. To that end, we define a denotational
GF-transformer semantics for probabilistic while-programs, and show
that it instantiates Kozen’s seminal distribution transformer semantics.
We then study the effective usage of GFs for program analysis. We show
that finitely expressible GFs enable checking super-invariants by means
of computer algebra tools, and that they can be used to determine ter-
mination probabilities. The paper concludes by characterizing a class of
— possibly infinite-state — programs whose semantics is a rational GF
encoding a discrete phase-type distribution.

Keywords: probabilistic programs · quantitative verification · seman-
tics · formal power series.

1 Introduction

Probabilistic programs are sequential programs for which coin flipping is a first-
class citizen. They are used e.g. to represent randomized algorithms, probabilistic
graphical models such as Bayesian networks, cognitive models, or security pro-

cotocols. Although probabilistic programs are typically rather small, their analysis
is intricate. For instance, approximating expected values of program variables at
program termination is as hard as the universal halting problem [19]. Determin-
ing higher moments such as variances is even harder. Deductive program verifica-
tion techniques based on a quantitative version of weakest preconditions [22,17]

 enable to reason about the outcomes of probabilistic programs, such as what is
the probability that a program variable equals a certain value. Dedicated anal-
ysis techniques have been developed to e.g., determine tail bounds [6], decide
almost-sure termination [23,8], or to compare programs [1].

* This research was funded by the ERC AdG project FRAPPANT (787914) and the
DFG RTG 2236 UnRAVeL.
This paper aims at exploiting the well-tried potential of probability generating functions (PGFs) [16] for analyzing probabilistic programs. In our setting, PGFs are power series representations encoding discrete probability mass functions of joint distributions over program variables. PGF representations — in particular if finite — enable a simple extraction of important information from the encoded distributions such as expected values, higher moments, termination probabilities or stochastic independence of program variables.

To enable the usage of PGFs for program analysis, we define a denotational semantics of a simple probabilistic while-language akin to probabilistic GCL [22]. Our semantics is defined in a forward manner: given an input distribution over program variables as a PGF, it yields a PGF representing the resulting subdistribution. The “missing” probability mass represents the probability of non-termination. More accurately, our denotational semantics transforms formal power series (FPS). Those form a richer class than PGFs, which allows for overapproximations of probability distributions. The meaning of While-loops are given as least fixed points of FPS transformers. It is shown that our semantics is in fact an instantiation of Kozen’s seminal distribution-transformer semantics [20].

The semantics provides a sound basis for program analysis using PGFs. Using Park’s Lemma, we obtain a simple technique to prove whether a given FPS over-approximates a program’s semantics i.e., whether an FPS is a so-called super-invariant. Such upper bounds can be quite useful: for almost-surely terminating programs, such bounds can provide exact program semantics, whereas, if the mass of an overapproximation is strictly less than one, the program is provably non-almost-surely terminating. This result is illustrated on a non-trivial random walk and on examples illustrating that checking whether an FPS is a super-invariant can be automated using computer algebra tools.

In addition, we characterize a class of — possibly infinite-state — programs whose PGF semantics is a rational function. These homogeneous bounded programs (HB programs) are characterized by loops in which each unbounded variable has no effect on the loop guard and is in each loop iteration incremented by a quantity independent of its own value. Operationally speaking, HB programs can be considered as finite-state Markov chains with rewards in which rewards can grow unboundedly large. It is shown that the rational PGF of any program that is equivalent to an almost-surely terminating HB program represents a multi-variate discrete phase-type distribution [26]. We illustrate this result by obtaining a closed-form characterization for the well-studied infinite-state dueling cowboys example [22].

Related work. Semantics of probabilistic programs is a well-studied topic. This includes the seminal works by Kozen [20] and McIver and Morgan [22]. Other related semantics of discrete probabilistic while-programs are e.g., given in several other articles like [22,29,11,27,4]. PGFs have recent scant attention in the analysis of probabilistic programs. A notable exception is [5] in which generating functions of finite Markov chains are obtained by Padé approximation. Computer
Generating Functions for Probabilistic Programs

algebra systems have been used to transform probabilistic programs [7], and more recently in the automated generation of moment-based loop invariants [2].

Organization of this paper. After recapping FPSs and PGFs in Sections 2–3, we define our FPS transformer semantics in Section 4, discuss some elementary properties and show it instantiates Kozen’s distribution transformer semantics [20]. Section 5 presents our approach for verifying upper bounds to loop invariants and illustrates this by various non-trivial examples. In addition, it characterizes programs that are representable as finite-state Markov chains equipped with rewards and presents the relation to discrete phase-type distributions. Section 6 concludes the paper. All proofs can be found in the appendix.

2 Formal Power Series

Our goal is to make the potential of probability generating functions available to the formal verification of probabilistic programs. The programs we consider will, without loss of generality, operate on a fixed set of \( k \) program variables. The valuations of those variables range over \( \mathbb{N} \). A *program state* \( \sigma \) is hence a vector in \( \mathbb{N}^k \). We denote the state \((0,\ldots,0)\) by \( \mathbf{0} \).

A prerequisite for understanding probability generating functions are (multivariate) *formal power series* — a special way of representing a potentially infinite \( k \)-dimensional array. For \( k=1 \), this amounts to representing a sequence.

**Definition 1 (Formal Power Series).** Let \( X = X_1, \ldots, X_k \) be a fixed sequence of \( k \) distinct formal indeterminates. For a state \( \sigma = (\sigma_1, \ldots, \sigma_k) \in \mathbb{N}^k \), let \( X^\sigma \) abbreviate the formal multiplication \( X_1^{\sigma_1} \cdots X_k^{\sigma_k} \). The latter object is called a monomial and we denote the set of all monomials over \( X \) by \( \text{Mon}(X) \).

A (multivariate) formal power series (FPS) is a formal sum

\[
F = \sum_{\sigma \in \mathbb{N}^k} [\sigma]_F \cdot X^\sigma,
\]

where \( \mathbb{R}_{\geq 0}^\infty \) denotes the extended positive real line. We denote the set of all FPSs by \( \text{FPS} \). Let \( F, G \in \text{FPS} \). If \( [\sigma]_F < \infty \) for all \( \sigma \in \mathbb{N}^k \), we denote this fact by \( F \ll \infty \). The addition \( F + G \) and scaling \( r \cdot F \) by a scalar \( r \in \mathbb{R}_{\geq 0}^\infty \) is defined coefficient-wise by

\[
F + G = \sum_{\sigma \in \mathbb{N}^k} ([\sigma]_F + [\sigma]_G) \cdot X^\sigma \quad \text{and} \quad r \cdot F = \sum_{\sigma \in \mathbb{N}^k} r \cdot [\sigma]_F \cdot X^\sigma.
\]

For states \( \sigma = (\sigma_1, \ldots, \sigma_k) \) and \( \tau = (\tau_1, \ldots, \tau_k) \), we define \( \sigma + \tau = (\sigma_1 + \tau_1, \ldots, \sigma_k + \tau_k) \). The multiplication \( F \cdot G \) is given as their Cauchy product (or discrete convolution)

\[
F \cdot G = \sum_{\sigma, \tau \in \mathbb{N}^k} [\sigma]_F \cdot [\tau]_G \cdot X^{\sigma + \tau}.
\]

Drawing coefficients from the extended reals enables us to define a *complete lattice* on FPSs in Section 4. Our analyses in Section 5 will, however, only consider FPSs with \( F \ll \infty \).
3 Generating Functions

A generating function is a device somewhat similar to a bag. Instead of carrying many little objects detachedly, which could be embarrassing, we put them all in a bag, and then we have only one object to carry, the bag.

— George Pólya [30]

Formal power series pose merely a particular way of encoding an infinite $k$-dimensional array as yet another infinitary object, but we still carry all objects forming the array (the coefficients of the FPS) detachedly and there seems to be no advantage in this particular encoding. It even seems more bulky. We will now, however, see that this bulky encoding can be turned into a one-object bag carrying all our objects: the generating function.

**Definition 2 (Generating Functions).** The generating function of a formal power series $F = \sum_{\sigma \in \mathbb{N}^k} [\sigma]_F \cdot \mathbf{X}^\sigma \in \text{FPS}$ with $F \ll \infty$ is the partial function

$$f : [0, 1]^k \to \mathbb{R}_{\geq 0}, \quad (x_1, \ldots, x_k) \mapsto \sum_{\sigma = (\sigma_1, \ldots, \sigma_k) \in \mathbb{N}^k} [\sigma]_F \cdot x_1^{\sigma_1} \cdots x_k^{\sigma_k}.$$ 

In other words: in order to turn an FPS into its generating function, we merely treat every *formal* indeterminate $X_i$ as an "actual" indeterminate $x_i$, and the formal multiplications and the formal sum also as "actual" ones. The generating function $f$ of $F$ is *uniquely determined* by $F$ as we require all coefficients of $F$ to be non-negative, and so the ordering of the summands is irrelevant: For a given point $\mathbf{x} \in [0, 1]^k$, the sum defining $f(\mathbf{x})$ either converges absolutely to some positive real or diverges absolutely to $\infty$. In the latter case, $f$ is undefined at $\mathbf{x}$ and hence $f$ may indeed be partial.

Since generating functions stem from formal power series, they are infinitely often differentiable at $0 = (0, \ldots, 0)$. Because of that, we can recover $F$ from $f$ as the (multivariate) Taylor expansion of $f$ at $0$.

**Definition 3 (Multivariate Derivatives and Taylor Expansions).** For $\sigma = (\sigma_1, \ldots, \sigma_k) \in \mathbb{N}^k$, we write $f^{(\sigma)}$ for the function $f$ differentiated $\sigma_1$ times in $x_1$, $\sigma_2$ times in $x_2$, and so on. If $f$ is infinitely often differentiable at $0$, then the Taylor expansion of $f$ at $0$ is given by

$$\sum_{\sigma \in \mathbb{N}^k} \frac{f^{(\sigma)}(0)}{\sigma_1! \cdots \sigma_k!} \cdot x_1^{\sigma_1} \cdots x_k^{\sigma_k}.$$ 

If we replace every indeterminate $x_i$ by the *formal* indeterminate $X_i$ in the Taylor expansion of generating function $f$ of $F$, then we obtain the formal power series $F$. It is in precisely that sense, that $f$ *generates* $F$.

**Example 1 (Formal Power Series and Generating Functions).** Consider the infinite (1-dimensional) sequence $\frac{1}{1}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \ldots$. Its (univariate) FPS — the entity carrying all coefficients detachedly — is given as

$$\frac{1}{2} + \frac{1}{4} \cdot X + \frac{1}{8} \cdot X^2 + \frac{1}{16} \cdot X^3 + \frac{1}{32} \cdot X^4 + \frac{1}{64} \cdot X^5 + \frac{1}{128} \cdot X^6 + \frac{1}{256} \cdot X^7 + \ldots.$$ 

(†)
On the other hand, its generating function — the bag — is given concisely by

\[
\frac{1}{2 - x}.
\]

Figuratively speaking, (†) is itself the infinite sequence \(a_n := \frac{1}{2^n}\), whereas (§) is a bag with the label “infinite sequence \(a_n := \frac{1}{2^n}\). The fact that (†) generates (§),
follows from the Taylor expansion of \(\frac{1}{2-x}\) at 0 being \(\frac{1}{2} + \frac{1}{4}x + \frac{1}{8}x^2 + \ldots\). △

The potential of generating functions is that manipulations to the functions —
i.e. to the concise representations — are in a one-to-one correspondence to the
associated manipulations to FPSs [10]. For instance, if \(f(x)\) is the generating
function of \(F\) encoding the sequence \(a_1, a_2, a_3,\ldots\), then the function \(f(x) \cdot x\) is
the generating function of \(F \cdot X\) which encodes the sequence \(0, a_1, a_2, a_3,\ldots\).

As another example for correspondence between operations on FPSs and
generating functions, if \(f(x)\) and \(g(x)\) are the generating functions of \(F\) and \(G\),
respectively, then \(f(x) + g(x)\) is the generating function of \(F + G\).

**Example 2 (Manipulating Generating Functions).** Revisiting Example 1, if we
multiply \(\frac{1}{2-x}\) by \(x\), we change the label on our bag from “infinite sequence
\(a_n := \frac{1}{2^n}\)” to “a 0 followed by an infinite sequence \(a_{n+1} := \frac{1}{2^n}\)” and — just by
changing the label — the bag will now contain what it says on its label. Indeed,
the Taylor expansion of \(\frac{1}{2-x}\) at 0 is \(0 + \frac{1}{2}x + \frac{1}{4}x^2 + \frac{1}{8}x^3 + \frac{1}{16}x^4 + \ldots\) encoding
the sequence \(0, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \ldots\). △

Due to the close correspondence of FPSs and generating functions [10], we use
both concepts interchangeably, as is common in most mathematical literature.
We mostly use FPSs for definitions and semantics, and generating functions in
calculations and examples.

**Probability Generating Functions.** We now use formal power series to rep-resent probability distributions.

**Definition 4 (Probability Subdistribution).** A probability subdistribution
(or simply subdistribution) over \(\mathbb{N}^k\) is a function

\[
\mu: \mathbb{N}^k \to [0, 1], \quad \text{such that} \quad |\mu| = \sum_{\sigma \in \mathbb{N}^k} \mu(\sigma) \leq 1.
\]

We call \(|\mu|\) the mass of \(\mu\). We say that \(\mu\) is a (full) distribution if
\(|\mu| = 1\), and a proper subdistribution if \(|\mu| < 1\). The set of all subdistributions on \(\mathbb{N}^k\) is
denoted by \(\mathcal{D}_{\leq}(\mathbb{N}^k)\) and the set of all full distributions by \(\mathcal{D}(\mathbb{N}^k)\).

We need subdistributions for capturing non-termination. The “missing” proba-bility mass \(1 - |\mu|\) precisely models the probability of non-termination.

The generating function of a (sub-)distribution is called a **probability generating function.** Many properties of a distribution \(\mu\) can be read off from its
generating function \(G_\mu\) in a simple way. We demonstrate how to extract a few
common properties in the following example.
Generating Functions for Probabilistic Programs

The FPS semantics of \(pGCL\) will be defined in a forward denotational style, where the program variables \(x_1, \ldots, x_k\) correspond to the formal indeterminates \(X_1, \ldots, X_k\) of FPSs.

For handling assignments, if-conditions and while-loops, we need some auxiliary functions on FPSs: For an arithmetic expression \(E\) over program variables, we denote by \(\text{eval}^\sigma(E)\) the evaluation of \(E\) in program state \(\sigma\). For a predicate \(B \subseteq \mathbb{N}^k\) and FPS \(F\), we define the restriction of \(F\) to \(B\) by

\[
\langle F \rangle_B := \sum_{\sigma \in B} \left[\sigma\right] F \cdot X_{\sigma_1} \cdots X_{\sigma_k},
\]

i.e. \(\langle F \rangle_B\) is the FPS obtained from \(F\) by setting all coefficients \(\left[\sigma\right] F\) where \(\sigma \not\in B\) to 0. Using these prerequisites, our FPS transformer semantics is given as follows:

**Definition 7 (FPS Semantics of \(pGCL\)).** The semantics \(J_P^K : \text{FPS} \rightarrow \text{FPS}\) of a loop-free \(pGCL\) program \(P\) is given according to the upper part of Table 1.

The unfolding operator \(\Phi_B^P\) for the loop \(\text{while}(B)\{P\}\) is defined by

\[
\Phi_B^P : (\text{FPS} \rightarrow \text{FPS}) \rightarrow (\text{FPS} \rightarrow \text{FPS}), \quad \psi \mapsto \lambda F . \langle F \rangle_{\neg B} + \psi(J_P^K(\langle F \rangle_B)).
\]

The partial order \((\text{FPS}, \preceq)\) extends to a partial order \((\text{FPS} \rightarrow \text{FPS}, \sqsubseteq)\) on FPS transformers by a point-wise lifting of \(\preceq\). The least element of this partial order is the transformer \(0 = \lambda F.0\) mapping any FPS \(F\) to the zero series. The semantics of \(\text{while}(B)\{P\}\) is then given by the least fixed point (with respect to \(\sqsubseteq\)) of its unfolding operator, i.e.

\[
J_{\text{while}}(B)\{P\} = \ellfp \Phi_B^P.
\]

### Table 1.

<table>
<thead>
<tr>
<th>Program</th>
<th>FPS Transformer Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{skip})</td>
<td>(F)</td>
</tr>
<tr>
<td>(x_i := E)</td>
<td>(F)</td>
</tr>
<tr>
<td>(P; P)</td>
<td>(F)</td>
</tr>
<tr>
<td>({P} [p] {P})</td>
<td>(F)</td>
</tr>
<tr>
<td>(\text{if}(B) {P} \text{ else } {P})</td>
<td>(F)</td>
</tr>
<tr>
<td>(\text{while}(B) {P})</td>
<td>(F)</td>
</tr>
</tbody>
</table>

4 FPS Semantics for \(pGCL\)

In this section, we give denotational semantics to probabilistic programs in terms of FPS transformers and establish some elementary properties useful for program analysis. We begin by endowing FPSs and PGFs with an order structure:

**Definition 5 (Order on FPS).** For all \(F, G \in \text{FPS}\), let

\[
F \preceq G \quad \text{iff} \quad \forall \sigma \in \mathbb{N}^k : \left[\sigma\right]_G \leq \left[\sigma\right]_F.
\]

**Lemma 1 (Completeness of \(\preceq\) on FPS).** \((\text{FPS}, \preceq)\) is a complete lattice.

4.1 FPS Transformer Semantics

Recall that we assume programs to range over exactly \(k\) variables with valuations in \(\mathbb{N}^k\). Our program syntax is similar to Kozen [20] and McIver & Morgan [22].

**Definition 6 (Syntax of \(pGCL\) [20,22]).** A program \(P\) in probabilistic Guarded Command Language (pGCL) adheres to the grammar

\[
P ::= \text{skip} \mid x_i := E \mid P; P \mid \{P\} [p] \{P\}\]
\[
\mid \text{if}(B) \{P\} \text{ else } \{P\} \mid \text{while}(B) \{P\},
\]

where \(x_i \in \{x_1, \ldots, x_k\}\) is a program variable, \(E\) is an arithmetic expression over program variables, \(p \in [0, 1]\) is a probability, and \(B\) is a predicate (called guard) over program variables.

Example 3 (Geometric Distribution PGF). Recall Example 1. The presented formal power series encodes a geometric distribution \(\mu_{\text{geo}}\) with parameter \(1/2\) of a single variable \(X\). The fact that \(\mu_{\text{geo}}\) is a proper probability distribution, for instance, can be easily verified computing \(G_{\text{geo}}(1) = \frac{1}{2-1} = 1\). The expected value of \(X\) is given by \(G'_{\text{geo}}(1) = \frac{1}{(2-1)^2} = 1\). △

Extracting Common Properties. Important information about probability distributions is, for instance, the first and higher moments. In general, the \(k\)th factorial moment of variable \(X_i\) can be extracted from a PGF by computing \(\frac{\partial^k G}{\partial X_i^k}(1, \ldots, 1)\). This includes the mass \(|G|\) as the 0th moment. The marginal distribution of variable \(X_i\) can simply be extracted from \(G\) by \(G(1, \ldots, X_i, \ldots, 1)\).

We also note that PGFs can treat stochastic independence. For instance, for a bivariate PGF \(H\) we can check for stochastic independence of the variables \(X\) and \(Y\) by checking whether \(H(X, Y) = H(X, 1) \cdot H(1, Y)\).

In general, one must take the limit \(X_i \to 1\) from below.
Example 4. Consider the program $P = \{x := 0\} \{1/2\} \{x := 1\} \#c := c + 1$. Using the input PGF $G = 1$, which denotes a point mass on state $\sigma = 0$. Using the annotation style shown in the left margin, denoting that $J_{P, K}(G) = G'$, we calculate $J_{P, K}(G)$ as follows:

\[
(F)_B := \sum_{\sigma \in B} [\sigma]_F \cdot X^\sigma,
\]

i.e. $(F)_B$ is the FPS obtained from $F$ by setting all coefficients $[\sigma]_F$ where $\sigma \notin B$ to 0. Using these prerequisites, our FPS transformer semantics is given as follows:

**Definition 7 (FPS Semantics of \textit{pGCL}).** The semantics $[P] : \text{FPS} \rightarrow \text{FPS}$ of a loop-free \textit{pGCL} program $P$ is given according to the upper part of Table 1.

The unfolding operator $\Phi_{B, P}$ for the loop $\texttt{while}(B)\{P\}$ is defined by

\[
\Phi_{B, P} : (\text{FPS} \rightarrow \text{FPS}) \rightarrow (\text{FPS} \rightarrow \text{FPS}), \quad \psi \mapsto \lambda F. (F)_{-B} + \psi ([P] ((F)_B)).
\]

The partial order $(\text{FPS}, \preceq)$ extends to a partial order $(\text{FPS} \rightarrow \text{FPS}, \sqsubseteq)$ on FPS transformers by a point-wise lifting of $\preceq$. The least element of this partial order is the transformer $0 = \lambda F. 0$ mapping any FPS $F$ to the zero series. The semantics of $\texttt{while}(B)\{P\}$ is then given by the least fixed point (with respect to $\sqsubseteq$) of its unfolding operator, i.e.

\[
[\texttt{while}(B)\{P\}] = \text{lfp} \Phi_{B, P}.
\]

### Table 1. FPS transformer semantics of \textit{pGCL} programs.

<table>
<thead>
<tr>
<th>$P$</th>
<th>$<a href="F">P</a>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>$F$</td>
</tr>
<tr>
<td>$x_i := E$</td>
<td>$\sum_{\sigma \in \mathbb{N}^k} \mu_{\sigma} X_{i1}^{\sigma_1} \cdots X_{ik}^{\sigma_k}$</td>
</tr>
<tr>
<td>${P_1} [p] {P_2}$</td>
<td>$p \cdot <a href="F">P_1</a> + (1 - p) \cdot <a href="F">P_2</a>$</td>
</tr>
<tr>
<td>$\texttt{if}(B){P_1} \texttt{else}{P_2}$</td>
<td>$[P_1](\langle F \rangle_B) + [P_2](\langle F \rangle_{-B})$</td>
</tr>
<tr>
<td>$P_1; P_2$</td>
<td>$<a href="%5BP_2%5D(F)">P_1</a>$</td>
</tr>
<tr>
<td>$\texttt{while}(B){P}$</td>
<td>$(\text{lfp} \Phi_{B, P})(F)$, for $\Phi_{B, P}(\psi) = \lambda F. (F)_{-B} + \psi ([P] ((F)_B))$</td>
</tr>
</tbody>
</table>
Example 4. Consider the program \( P = \{ x := 0 \} \{ \text{if } x = 1 \{ c := c + 1 \} \} \), and the input PGF \( G = 1 \), which denotes a point mass on state \( \sigma = 0 \). Using the annotation style shown in the left margin, denoting that \( \|P\|(G) = G' \), we calculate \( \|P\|(G) \) as follows:

\[ \| P \| \left( \begin{array}{l} 1 \\
\{ x := 0 \} \{ \text{if } x = 1 \{ c := c + 1 \} \} \\
\{ 1 \} + \frac{X}{2} \\
c := c + 1 \\
\{ \frac{C}{2} + \frac{CX}{2} \} \end{array} \right) \]

As for the semantics of \( c := c + 1 \), see Table 2. \( \triangle \)

Before we study how our FPS transformers behave on PGFs in particular, we now first argue that our FPS semantics is well-defined. While evident for loop-free programs, we appeal to the Kleene Fixed Point Theorem for loops [21], which requires \( \omega \)-continuous functions.

**Theorem 1 (\( \omega \)-continuity of pGCL Semantics).** The semantic functional \( \| \cdot \| \) is \( \omega \)-continuous, i.e. for all programs \( P \in \text{pGCL} \) and all increasing \( \omega \)-chains \( F_1 \leq F_2 \leq \ldots \) in FPS,

\[ \| P \| \left( \sup_{n \in \mathbb{N}} F_n \right) = \sup_{n \in \mathbb{N}} \| P \| (F_n) . \]

**Theorem 2 (Well-definedness of FPS Semantics).** The semantics functional \( \| \cdot \| \) is well-defined, i.e. the semantics of any loop \( \text{while } (B) \{ P \} \) exists uniquely and can be written as

\[ \| \text{while } (B) \{ P \} \| = \text{ifp } \Phi_{B,P} = \sup_{n \in \mathbb{N}} \Phi_{B,P}^n (0) . \]

### 4.2 Healthiness Conditions of FPS Transformers

In this section we show basic, yet important, properties which follow from [20]. For instance, for any input FPS \( F \), the semantics of a program cannot yield as output an FPS with a mass larger than \( |F| \), i.e. programs cannot create mass.

<table>
<thead>
<tr>
<th>( P )</th>
<th>( | P | (F) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := x + k )</td>
<td>( X^k \cdot F(X,Y) )</td>
</tr>
<tr>
<td>( x := k \cdot x )</td>
<td>( F(X^k,Y) )</td>
</tr>
<tr>
<td>( x := x + y )</td>
<td>( F(X,XY) )</td>
</tr>
</tbody>
</table>

Table 2. Common assignments and their effects on the input PGF \( F(X,Y) \).
Theorem 5. The FPS semantics of pGCL is an instance of Kozen’s semantics, i.e. for all pGCL programs \(P\), we have
\[
\tau \circ J^P_K = T(P) \circ \tau.
\]
Equivalently, the following diagram commutes:

For more details about the connection between FPSs and measures, as well as more information about the actual translation, see Appendix A.3.

5 Analysis of Probabilistic Programs

Our PGF semantics enables the representation of the effect of a pGCL program on a given PGF. As a next step, we investigate to what extent a program analysis can exploit such PGF representations. To that end, we consider the overapproximation with loop invariants (Section 5.1) and provide examples showing that checking whether an FPS transformer overapproximates a loop can be checked with computer algebra tools. In addition, we determine a subclass of pGCL programs whose effect on an arbitrary input state is ensured to be a rational PGF encoding a phase-type distribution (Section 5.2).

5.1 Invariant-style Overapproximation of Loops

In this section, we seek to overapproximate loop semantics, i.e. for a given loop \(W = \text{while } (B) \{P\}\) we want to find a (preferably simple) FPS transformer \(\psi\), such that
\[
J^W_K \sqsubseteq \psi,
\]
meaning that for any input \(G\), we have
\[
J^W_K(G) \preceq \psi(G).
\]
Notably, even if \(G\) is a PGF, we do not require \(\psi(G)\) to be one. Instead, \(\psi(G)\) can have a mass larger than one. This is fine, because it still overapproximates the actual semantics coefficient-wise. Such overapproximations immediately carry over to reading off expected values (cf. Section 3), for instance
\[
\partial \partial X J^W_K(G) (1) \leq \partial \partial X \psi(G) (1).
\]
We use invariant-style reasoning for verifying that a given \(\psi\) overapproximates the semantics of \(J^W_K\). For that, we introduce the notion of a superinvariant and employ Park’s Lemma [28] — well-known in fixed point theory — to obtain a conceptually simple proof rule for verifying overapproximations of while loops.

Theorem 6 (Superinvariants and Loop Overapproximations). Let \(\Phi_B,P\) be the unfolding operator of \(\text{while } (B) \{P\}\) (cf. Def. 7) and \(\psi:\text{FPS} \to \text{FPS}\). Then
\[
\Phi_B,P(\psi) \sqsubseteq \psi
\]
implies
\[
J_{\text{while } (B) \{P\}} K \sqsubseteq \psi.
\]
Theorem 5. The FPS semantics of pGCL is an instance of Kozen’s semantics, i.e. for all pGCL programs $P$, we have
$$\tau \circ \llbracket P \rrbracket = \Xi(P) \circ \tau.$$  
Equivalently, the following diagram commutes:

\[
\begin{array}{ccc}
\text{FPS} & \xrightarrow{\tau} & \mathcal{M} \\
\downarrow[\llbracket P \rrbracket] & & \downarrow[\Xi(P)] \\
\text{FPS} & \xrightarrow{\tau} & \mathcal{M}
\end{array}
\]

For more details about the connection between FPSs and measures, as well as more information about the actual translation, see Appendix A.3.

5 Analysis of Probabilistic Programs

Our PGF semantics enables the representation of the effect of a pGCL program on a given PGF. As a next step, we investigate to what extent a program analysis can exploit such PGF representations. To that end, we consider the overapproximation with loop invariants (Section 5.1) and provide examples showing that checking whether an FPS transformer overapproximates a loop can be checked with computer algebra tools. In addition, we determine a subclass of pGCL programs whose effect on an arbitrary input state is ensured to be a rational PGF encoding a phase-type distribution (Section 5.2).

5.1 Invariant-style Overapproximation of Loops

In this section, we seek to overapproximate loop semantics, i.e. for a given loop $W = \text{while}(B)\{P\}$, we want to find a (preferably simple) FPS transformer $\psi$, such that $\llbracket W \rrbracket \subseteq \psi$, meaning that for any input $G$, we have $\llbracket W \rrbracket(G) \leq \psi(G)$ (cf. Definition 7). Notably, even if $G$ is a PGF, we do not require $\psi(G)$ to be one. Instead, $\psi(G)$ can have a mass larger than one. This is fine, because it still overapproximates the actual semantics coefficient-wise. Such overapproximations immediately carry over to reading off expected values (cf. Section 3), for instance

$$\frac{\partial}{\partial X} \llbracket W \rrbracket(G)(1) \leq \frac{\partial}{\partial X} \psi(G)(1).$$

We use invariant-style reasoning for verifying that a given $\psi$ overapproximates the semantics of $\llbracket W \rrbracket$. For that, we introduce the notion of a superinvariant and employ Park’s Lemma [28] — well-known in fixed point theory — to obtain a conceptually simple proof rule for verifying overapproximations of while loops.

Theorem 6 (Superinvariants and Loop Overapproximations). Let $\Phi_{B,P}$ be the unfolding operator of $\text{while}(B)\{P\}$ (cf. Def. 7) and $\psi : \text{FPS} \to \text{FPS}$. Then
$$\Phi_{B,P}(\psi) \subseteq \psi \implies \llbracket \text{while}(B)\{P\} \rrbracket \subseteq \psi.$$
while \((x = 1)\) {
\{ \\
\{ x \\assign 0 \} \quad [1/2] \quad \{ x \\assign 1 \}; \\
c \\assign c + 1 \\
\}
}

**Program 1.1.** Geometric distribution generator.

We call a \(\psi\) satisfying \(\Phi_{B,P}(\psi) \subseteq \psi\) a superinvariant. We are interested in linear superinvariants, as our semantics is also linear (cf. Theorem 4). Furthermore, linearity allows to define \(\psi\) solely in terms of its effect on monomials, which makes reasoning considerably simpler:

**Corollary 2.** Given \(f\): Mon \((X) \rightarrow \text{FPS}\), let the linear extension \(\hat{f}\) of \(f\) be

\[
\hat{f} : \text{FPS} \rightarrow \text{FPS}, \quad F \mapsto \sum_{\sigma \in \mathbb{N}^k} [\sigma]_{P} f(X^\sigma).
\]

Let \(\Phi_{B,P}\) be the unfolding operator of \(\text{while}(B)\{P\}\). Then

\[
\forall \sigma \in \mathbb{N}^k : \Phi_{B,P}(\hat{f})(X^\sigma) \subseteq \hat{f}(X^\sigma) \quad \text{implies} \quad \llbracket \text{while}(B)\{P\} \rrbracket \subseteq \hat{f}.
\]

We call an \(f\) satisfying the premise of the above corollary a superinvariantlet. Notice that superinvariantlet and their extensions agree on monomials, i.e. \(f(X^\sigma) = \hat{f}(X^\sigma)\). Let us examine a few examples for superinvariantlet-reasoning.

**Example 5 (Verifying Precise Semantics).** In Program 1.1, in each iteration, a fair coin flip determines the value of \(x\). Subsequently, \(c\) is incremented by 1.

Consider the following superinvariantlet:

\[
f(X^iC^j) = C^j \cdot \begin{cases} \\
\frac{c}{2-c}, & \text{if } i = 1; \\
X^i, & \text{if } i \neq 1.
\end{cases}
\]

To verify that \(f\) is indeed a superinvariantlet, we have to show that

\[
\Phi_{B,P}(\hat{f})(X^iC^j) = \langle X^iC^j \rangle_{x \neq 1} + \hat{f}(\llbracket P \rrbracket (\langle X^iC^j \rangle_{x = 1}))
\]

\[
\leq \hat{f}(X^iC^j).
\]

For \(i \neq 1\), we get

\[
\Phi_{B,P}(\hat{f})(X^iC^j) = \langle X^iC^j \rangle_{x \neq 1} + \hat{f}(\llbracket P \rrbracket (0)) = X^iC^j = f(X^iC^j) = \hat{f}(X^iC^j).
\]

For \(i = 1\), we get

\[
\Phi_{B,P}(\hat{f})(X^iC^j) = \hat{f}(\frac{1}{2}X^0C^{j+1} + \frac{1}{2}X^1C^{j+1}) = \frac{1}{2}f(X^0C^{j+1}) + \frac{1}{2}f(X^1C^{j+1}) \quad \text{(by linearity of } \hat{f})
\]

\[
= \frac{c^{j+1}}{2-c} = f(X^1C^j) = \hat{f}(X^1C^j). \quad \text{(by definition of } f)
\]

Hence, Corollary 2 yields

\[
J_\mathcal{W}(X) \subseteq f(X) = C^2 - C.
\]

For this example, we can state even more. As the program is almost surely terminating, and \(\|f(X^iC^j)\|_{\mathbb{N}^k} = 1\) for all \((i,j) \in \mathbb{N}^2\), we conclude that \(\hat{f}\) is exactly the semantics of \(\mathcal{W}\), i.e. \(\hat{f} = J_\mathcal{W}\).

\(\triangle\)

**Example 6 (Verifying Proper Overapproximations).** Program 1.2 models a one-dimensional, left-bounded random walk. Given an input \((i,j) \in \mathbb{N}^2\), this program can only terminate in an even (if \(i\) is even) or odd (if \(i\) is odd) number of steps.

This insight can be encoded into the following superinvariantlet:

\[
f(X^0C^j) = C^j \quad \text{and} \quad f(X^iC^j) = C^j \cdot \left\{ \begin{array}{ll}
C^1 - C^2, & \text{if } i \text{ is odd;} \\
1 - C^2, & \text{if } i \text{ is even.}
\end{array} \right.
\]

It is straightforward to verify that \(f\) is a proper superinvariantlet (proper because \(C^1 - C^2 = C + C^3 + C^5 + \ldots\) is not a PGF) and hence \(f\) properly overapproximates the loop semantics. Another superinvariantlet for Program 1.2 is given by

\[
h(X^iC^j) = C^j \cdot \begin{cases} \\
\frac{1}{1 - \sqrt{1 - C^2}} X^i, & \text{if } i \geq 1; \\
1, & \text{if } i = 0.
\end{cases}
\]

Given that the program terminates almost-surely [13] and that \(h\) is a superinvariantlet yielding only PGFs, it follows that the extension of \(h\) is exactly the semantics of Program 1.2. An alternative derivation of this formula for the case \(h(X)\) can be found, e.g., in [14].

For both \(f\) and \(h\), we were able to prove that they are indeed superinvariantlets semi-automatically, using the computer algebra library SymPy [24]. The code is included in Appendix B (Program 1.5).

\(\triangle\)

**Example 7 (Proving Non-almost-sure Termination).** In Program 1.3, the branching probability of the choice statement depends on the value of a program variable. This notation is just syntactic sugar, as this behavior can be mimicked by loop constructs together with coin flips [3, pp. 115f].

To prove that Program 1.3 does not terminate almost-surely, we consider the following superinvariantlet:

\[
f(X^iC^j) = 1 - e^{-i/2} - 2 \sum_{n \geq 0} \frac{1}{n!},
\]

where \(e = 7.18281828\ldots\) is Euler’s number.
while (x > 0) {
    {x := x + 1} \{1 / 2\} {x := x - 1};
    c := c + 1
}

Program 1.2. Left-bounded 1-dimensional random walk.

Hence, Corollary 2 yields $\langle W \rangle (X) \subseteq f(X) = \frac{C}{2-C}$.

For this example, we can state even more. As the program is almost surely terminating, and $|f(X^iC^j)| = 1$ for all $(i, j) \in \mathbb{N}^2$, we conclude that $f$ is exactly the semantics of $W$, i.e. $\hat{f} = \langle W \rangle$.

\[ \triangle \]

Example 6 (Verifying Proper Overapproximations). Program 1.2 models a one-dimensional, left-bounded random walk. Given an input $(i, j) \in \mathbb{N}^2$, this program can only terminate in an even (if $i$ is even) or odd (if $i$ is odd) number of steps. This insight can be encoded into the following superinvariantlet:

\[
\begin{align*}
  f(X^iC^j) &= C^j \\
  f(X^{i+1}C^j) &= C^j \cdot \begin{cases} 
    \frac{C}{1-C^2}, & \text{if } i \text{ is odd;} \\ 
    \frac{1}{1-C^2}, & \text{if } i \text{ is even.}
  \end{cases}
\end{align*}
\]

It is straightforward to verify that $f$ is a proper superinvariantlet (proper because $\frac{C}{1-C^2} = C + C^3 + C^5 + \ldots$ is not a PGF) and hence $f$ properly overapproximates the loop semantics. Another superinvariantlet for Program 1.2 is given by

\[
h(X^iC^j) = C^j \cdot \begin{cases} 
  \left( \frac{1-\sqrt{1-C^2}}{C} \right)^i, & \text{if } i \geq 1; \\ 
  1, & \text{if } i = 0.
\end{cases}
\]

Given that the program terminates almost-surely [13] and that $h$ is a superinvariantlet yielding only PGFs, it follows that the extension of $h$ is exactly the semantics of Program 1.2. An alternative derivation of this formula for the case $h(X)$ can be found, e.g., in [14].

For both $f$ and $h$, we were able to prove that they are indeed superinvariantlets semi-automatically, using the computer algebra library SymPy [24]. The code is included in Appendix B (Program 1.5).

\[ \triangle \]

Example 7 (Proving Non-almost-sure Termination). In Program 1.3, the branching probability of the choice statement depends on the value of a program variable. This notation is just syntactic sugar, as this behavior can be mimicked by loop constructs together with coin flips [3, pp. 115f].

To prove that Program 1.3 does not terminate almost-surely, we consider the following superinvariantlet:

\[
f(X^i) = 1 - \frac{1}{e} \sum_{n=0}^{i-2} \frac{1}{n!}, \quad \text{where } e = 2.71828\ldots \text{ is Euler's number.}
\]
$$J \cdot W(K(H)) = C \cdot J(W(K(H)))$$

for all PGFs $H$. By symmetry, the same applies to variable $d$. Unfolding the loop once on input 1, yields

$$J(W(K(1)) = (1 - a) C \cdot J(W(K(T))) + a CX.$$ 

A similar equation for $J(W(K(T))$ involving $J(W(K(1)$ on its right-hand side holds. This way we obtain a system of two linear equations, although the program itself is infinite-state. The linear equation system has a unique solution $J(W(1)$ in the field of rational functions over the variables $C,D,T,X$ which is the PGF


From $G$ we can easily read off the following: The probability that the first cowboy wins ($x = 1$ and $t = 0$) equals $1 - a - (1 - a)(1 - b)$, and the expected total number of shots of the first cowboy is

$$\frac{1}{a} + \frac{b}{a} - \frac{ab}{a}.$$ 

Notice that this quantity equals $\infty$ if $a$ and $b$ are both zero, i.e. if both cowboys have zero hit chance.

If we write $G_V$ for the PGF obtained by substituting all but the variables in $V$ with 1, then we moreover see that $C \cdot G \neq G_{C,D}$. This means that $C$ and $D$ (as random variables) are stochastically dependent.

The distribution encoded in the PGF $J(W(K)$ is a discrete phase-type distribution. Such distributions are defined as follows: A Markov reward chain is a Markov chain where each state is augmented with a reward vector in $\mathbb{N}^k$. By definition, a (discrete) distribution on $\mathbb{N}^k$ is of phase-type iff it is the distribution of the total accumulated reward vector until absorption in a Markov reward chain with a single absorbing state and a finite number of transient states. In fact, Program 1.4 can be described as a Markov reward chain with two states ($X_0 T_0$ and $X_0 T_1$) and 2-dimensional reward vectors corresponding to the "counters" $(c, d)$: the reward in state $X_0 T_0$ is $(1, 0)$ and $(0, 1)$ in the other state.

Each $pGCL$ program describes a Markov reward chain [11]. It is not clear which (non-trivial) syntactical restrictions to impose to guarantee for such chains to be finite. In the remainder of this section, we give a characterization of while-loops that are equivalent to finite Markov reward chains. The idea of our criterion is that each variable has to fall into one of the following two categories:

**Definition 9 (Homogeneous and Bounded Variables).** Let $P \in pGCL$ be a program, $B$ be a guard and $x_i$ be a program variable. Then:

- $x_i$ is called **homogeneous** for $P$ if $J(P)K(x_i \cdot G) = x_i \cdot J(P)(G)$ for all $G \in \text{PGF}$.
- $x_i$ is called **bounded** by $B$ if the set $\{\sigma_i | \sigma \in B\}$ is finite.

Intuitively, homogeneity of $x_i$ means that it does not matter whether one increments the variable before or after the execution of $P$. Thus, a homogeneous variable behaves like an increment-only counter even if this may not be explicit in the syntax. In Example 8, the variables $c$ and $d$ in Program 1.4 are homogeneous (for both the loop-body and the loop itself). Moreover, $x$ and $t$ are clearly bounded by the loop guard. We can now state our characterization.

### 5.2 Rational PGFs

In several of the examples from the previous sections, we considered PGFs which were rational functions, that is, fractions of two polynomials. Since those are a particularly simple class of PGFs, it is natural to ask which programs have rational semantics. In this section, we present a semantic characterization of a class of while-loops whose output distribution is a (multivariate) discrete phase-type distribution [25,26]. This implies that the resulting PGF of such programs is an effectively computable rational function for any given input state. Let us illustrate this by an example.

**Example 8 (Dueling Cowboys).** Program 1.4 models two dueling cowboys [22]. The hit chance of the first cowboy is $a$ and the hit chance of the second cowboy is $b$, where $a, b \in [0,1]$. The cowboys shoot at each other in turns, as indicated by the variable $t$, until one of them gets hit ($x$ is set to 1). The variable $c$ counts the number of shots of the first cowboy and $d$ those of the second cowboy.

We observe that Program 1.4 is somewhat independent of the value of $c$, in the sense that moving the statement $c := c + 1$ to either immediately before or after the loop, yields an equivalent program. In our notation, this is expressed

---

6 These are not program variables.
Definition 10 (HB Loops). A loop $\text{while } (B) \{ P \}$ is called homogeneous-bounded (HB) if for all program states $\sigma \in B$, the PGF $J_{P K}(X \sigma)$ is a polynomial and for all program variables $x$ it either holds that $x$ is homogeneous for $P$ and the guard $B$ is independent of $x$, or that $x$ is bounded by the guard $B$.

In an HB loop, all the possible valuations of the bounded variables satisfying $B$ span the finite transient state space of a Markov reward chain in which the dimension of the reward vectors equals the number of homogeneous variables. The additional condition that $J_{P K}(X \sigma)$ is a polynomial ensures that there is only a finite amount of terminal (absorbing) states. Thus, we have the following:

Proposition 1. Let $W$ be a while-loop. Then $J_W K(X \sigma)$ is the (rational) PGF of a multivariate discrete phase-type distribution if and only if $W$ is equivalent to an HB loop that almost-surely terminates on input $\sigma$.

To conclude, we remark that there are various simple syntactic conditions for HB loops: For example, if $P$ is loop-free, then $J_{P K}(X \sigma)$ is always a polynomial. Similarly, if $x$ only appears in assignments of the form $x := x + k$, $k \geq 0$, then $x$ is homogeneous. Such updates of variables are e.g. essential in constant probability programs \[9\]. The crucial point is that such conditions are only sufficient but not necessary. Our semantic conditions thus capture the essence of phase-type distribution semantics more adequately while still being reasonably simple (albeit — being non-trivial semantic properties — undecidable in general).

6 Conclusion

We have presented a denotational distribution transformer semantics for probabilistic while-programs where the denotations are generation functions (GFs). Moreover, we have provided a simple invariant-style technique to prove that a given GF overapproximates the program’s semantics and identified a class of (possibly infinite-state) programs whose semantics is a rational GF encoding a phase-type distribution. Directions for future work include the (semi-)automated synthesis of invariants and the development of notions on how precise overapproximations by invariants actually are. On that end, a rule for verifying underapproximations (e.g. à la \[12\], which provides inductive rules for underapproximating expected values) would be a major step in that direction.

Another direction for future work is to support $\mathbb{Z}$-valued program variables. For expected values, work on verifying signed random variables exists \[18\] — for PGFs, the situation is less clear. An obvious choice would be to employ formal Laurent series, but those only allow for finitely many negative indices, thus eluding distributions with both infinite positive and infinite negative support.

References

**Definition 10 (HB Loops).** A loop while \((B) \{P\}\) is called homogeneous-bounded (HB) if for all program states \(\sigma \in B\), the PGF \([P](X^\sigma)\) is a polynomial and for all program variables \(x\) it either holds that:

- \(x\) is homogeneous for \(P\) and the guard \(B\) is independent of \(x\), or that
- \(x\) is bounded by the guard \(B\).

In an HB loop, all the possible valuations of the bounded variables satisfying \(B\) span the finite transient state space of a Markov reward chain in which the dimension of the reward vectors equals the number of homogeneous variables. The additional condition that \([P](X^\sigma)\) is a polynomial ensures that there is only a finite amount of terminal (absorbing) states. Thus, we have the following:

**Proposition 1.** Let \(W\) be a while-loop. Then \([W](X^\sigma)\) is the rational PGF of a multivariate discrete phase-type distribution if and only if \(W\) is equivalent to an HB loop that almost-surely terminates on input \(\sigma\).

To conclude, we remark that there are various simple syntactic conditions for HB loops: For example, if \(P\) is loop-free, then \([P](X^\sigma)\) is always a polynomial. Similarly, if \(x\) only appears in assignments of the form \(x := x + k\), \(k \geq 0\), then \(x\) is homogeneous. Such updates of variables are e.g. essential in constant probability programs [9]. The crucial point is that such conditions are only sufficient but not necessary. Our semantic conditions thus capture the essence of phase-type distribution semantics more adequately while still being reasonably simple (albeit — being non-trivial semantic properties — undecidable in general).

### 6 Conclusion

We have presented a denotational distribution transformer semantics for probabilistic while-programs where the denotations are generation functions (GFs). Moreover, we have provided a simple invariant-style technique to prove that a given GF overapproximates the program’s semantics and identified a class of (possibly infinite-state) programs whose semantics is a rational GF encoding a phase-type distribution. Directions for future work include the (semi-)automated synthesis of invariants and the development of notions on how precise overapproximations by invariants actually are. On that end, a rule for verifying underapproximations (e.g. à la [12], which provides inductive rules for underapproximating expected values) would be a major step in that direction.

Another direction for future work is to support \(Z\)-valued program variables. For expected values, work on verifying signed random variables exists [18] — for PGFs, the situation is less clear. An obvious choice would be to employ formal Laurent series, but those only allow for finitely many negative indices, thus eluding distributions with both infinite positive and infinite negative support.

### References

Springer (2019)
ration logic: a logic for reasoning about probabilistic pointer programs. PACMPL
3(POPL), 34:1–34:29 (2019)
4. Bichsel, B., Gehr, T., Vechev, M.T.: Fine-grained semantics for probabilistic pro-
5. Boreale, M.: Analysis of probabilistic systems via generating functions and Padé
6. Bouissou, O., Goubault, E., Putot, S., Chakarov, A., Sankaranarayanan, S.: Uncer-
tainty propagation using probabilistic affine forms and concentration of measure
qualitative and quantitative termination problems for affine probabilistic programs.
9. Giesl, J., Giesl, P., Hark, M.: Computing expected runtimes for constant probabili-
10. Graham, R., Knuth, D., Patashnik, O.: Concrete Mathematics: A Foundation for
Computer Science. Addison-Wesley (1994)
11. Gretz, F., Katoen, J., McIver, A.: Operational versus weakest pre-expectation se-
manitics for the probabilistic guarded command language. Perform. Evaluation 73,
110–132 (2014)
Springer (2002)
14. Icard, T.: Calibrating generative models: The probabilistic Chomsky-
Schützenberger hierarchy. Journal of Mathematical Psychology 95 (2020)
champaign, IL, 2019
17. Kaminski, B.L.: Advanced weakest precondition calculi for probabilistic programs.
expectations. In: 32nd Annual ACM/IEEE Symposium on Logic in Computer Sci-
Society (2017)
Computer Society (1979)
22. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Sys-
23. McIver, A., Morgan, C., Kaminski, B.L., Katoen, J.: A new proof rule for almost-
A Proofs of Section 4

A.1 Proofs of Section 4.1

Lemma 1 (Completeness of \(\preceq\) on \(\text{FPS}\)).

\((\text{FPS}, \preceq)\) is a complete lattice.

Proof. We start by showing that \((\text{FPS}, \preceq)\) is a partial order. Let \(F,G,H \in \text{FPS}\), \(\sigma \in \mathbb{N}^k\). For reflexivity, consider the following:

\[ G \preceq G \iff \forall \sigma \in \mathbb{N}^k: [\sigma] G \leq [\sigma] G \iff \text{true}. \]

For antisymmetry, consider the following:

\[ G \preceq H \text{ and } H \preceq G \text{ implies } \forall \sigma \in \mathbb{N}^k: [\sigma] G \leq [\sigma] H \text{ and } [\sigma] H \leq [\sigma] G \text{ implies } [\sigma] G = [\sigma] H. \]

For transitivity, consider the following:

\[ G \preceq H \text{ and } H \preceq F \text{ implies } \forall \sigma \in \mathbb{N}^k: [\sigma] G \leq [\sigma] H \text{ and } [\sigma] H \leq [\sigma] F \text{ implies } \forall \sigma \in \mathbb{N}^k: [\sigma] G \leq [\sigma] F \text{ implies } G \preceq F. \]

Next, we show that every set \(S \subseteq \text{FPS}\) has a supremum \(\sup S = \sum_{\sigma \in \mathbb{N}^k} \sup F \in S [\sigma] F \cdot X^{\sigma}\) in \(\text{FPS}\). In particular, notice that \(\sup \emptyset = \sum_{\sigma \in \mathbb{N}^k} \sigma \cdot X^{\sigma}\). The fact that \(\sup S \in \text{FPS}\) is trivial since \(\sup F \in S [\sigma] F \in \mathbb{R}_\infty \geq 0\) for every \(\sigma \in \mathbb{N}^k\). Furthermore, the fact that \(\sup S\) is an upper bound on \(S\) is immediate since \(\preceq\) is defined coefficient-wise. Finally, \(\sup S\) is also the least upper bound, since, by definition of \(\preceq\), we have \([\sigma] \sup S = \sup F \in S [\sigma] F\).
A Proofs of Section 4

A.1 Proofs of Section 4.1

Lemma 1 (Completeness of ≤ on FPS). (FPS, ≤) is a complete lattice.

Proof. We start by showing that (FPS, ≤) is a partial order. Let $F, G, H \in \text{FPS}$, $\sigma \in \mathbb{N}^k$. For reflexivity, consider the following:

$$G \leq G$$

iff $\forall \sigma \in \mathbb{N}^k: [\sigma]_G \leq [\sigma]_G$

iff true.

For antisymmetry, consider the following:

$$G \leq H \text{ and } H \leq G$$

implies $\forall \sigma \in \mathbb{N}^k: [\sigma]_G \leq [\sigma]_H$ and $[\sigma]_H \leq [\sigma]_G$

implies $\forall \sigma \in \mathbb{N}^k: [\sigma]_G = [\sigma]_H$

implies $G = H$.

For transitivity, consider the following:

$$G \leq H \text{ and } H \leq F$$

implies $\forall \sigma \in \mathbb{N}^k: [\sigma]_G \leq [\sigma]_H$ and $[\sigma]_H \leq [\sigma]_F$

implies $\forall \sigma \in \mathbb{N}^k: [\sigma]_G \leq [\sigma]_F$

implies $G \leq F$.

Next, we show that every set $S \subseteq \text{FPS}$ has a supremum

$$\sup S = \sum_{\sigma \in \mathbb{N}^k} \sup_{F \in S} [\sigma]_F X^\sigma$$

in FPS. In particular, notice that $\sup \emptyset = \sum_{\sigma \in \mathbb{N}^k} 0 \cdot X^\sigma$. The fact that $\sup S \in \text{FPS}$ is trivial since $\sup_{F \in S} [\sigma]_F \in \mathbb{R}_{\geq 0}$ for every $\sigma \in \mathbb{N}^k$. Furthermore, the fact that $\sup S$ is an upper bound on $S$ is immediate since $\leq$ is defined coefficient-wise. Finally, $\sup S$ is also the least upper bound, since, by definition of $\leq$, we have $[\sigma]_{\sup S} = \sup_{F \in S} [\sigma]_F$.

The following proofs rely on the Monotone Sequence Theorem (MST), which we recall here: If $(a_n)_{n \in \mathbb{N}}$ is a monotonically increasing sequence in $\mathbb{R}_{\geq 0}$, then $\sup_n a_n = \lim_{n \to \infty} a_n$. In particular, if $(a_n)_{n \in \mathbb{N}}$ and $(b_n)_{n \in \mathbb{N}}$ are monotonically increasing sequences in $\mathbb{R}_{\geq 0}$, then

$$\sup_n a_n + \sup_n b_n = \lim_{n \to \infty} a_n + \lim_{n \to \infty} b_n = \lim_{n \to \infty} a_n + b_n = \sup_n a_n + b_n.$$
Theorem 1 (ω-continuity of pGCL Semantics). The semantic functional $\llbracket \cdot \rrbracket$ is ω-continuous, i.e. for all programs $P \in \text{pGCL}$ and all increasing ω-chains $F_1 \preceq F_2 \preceq \ldots$ in FPS,

$$\llbracket P \rrbracket \left( \sup_{n \in \mathbb{N}} F_n \right) = \sup_{n \in \mathbb{N}} \llbracket P \rrbracket (F_n).$$

Proof. By induction on the structure of $P$. Let $S = \{F_1, F_2, \ldots\}$ be an increasing ω-chain in FPS. First, we consider the base cases.

The case $P = \text{skip}$. We have

$$\llbracket P \rrbracket (\sup S) = \sup F \in S \{ F \} = \sup F \in S \{ \llbracket P \rrbracket (F) \}.$$ 

The case $P = x_i := E$. Let $S = \hat{G} = \sum_{\sigma \in \mathbb{N}^k} [\sigma]_{\hat{G}} \cdot X^\sigma$, where for each $\sigma \in \mathbb{N}^k$ we have $[\sigma]_{\hat{G}} = \sup_{F \in S} [\sigma]_F$. We calculate

$$\llbracket P \rrbracket (\sup S)$$

$$= \llbracket P \rrbracket \left( \hat{G} \right)$$

$$= \llbracket P \rrbracket \left( \sum_{\sigma \in \mathbb{N}^k} [\sigma]_{\hat{G}} \cdot X^\sigma \right)$$

$$= \llbracket P \rrbracket \left( \sum_{\sigma \in \mathbb{N}^k} \left[ \hat{G} \cdot X_1^{\sigma_1} \cdots X_i^{\sigma_i} \cdots X_k^{\sigma_k} \right] \right)$$

$$= \sum_{\sigma \in \mathbb{N}^k} \left[ \hat{G} \cdot X_1^{\sigma_1} \cdots X_i^{\sigma_i} \cdots X_k^{\sigma_k} \right]$$

$$= \sup_{F \in S} \left[ \sum_{\sigma \in \mathbb{N}^k} [\sigma]_F \cdot X_1^{\sigma_1} \cdots X_i^{\sigma_i} \cdots X_k^{\sigma_k} \right]$$

$$= \sup_{F \in S} \left[ \sum_{\sigma \in \mathbb{N}^k} [\sigma]_F \cdot X_1^{\sigma_1} \cdots X_i^{\sigma_i} \cdots X_k^{\sigma_k} \right]$$

(sup on FPS is defined coefficient–wise)

$$= \sup_{F \in S} \llbracket P \rrbracket \left( \sum_{\sigma \in \mathbb{N}^k} [\sigma]_F \cdot X_1^{\sigma_1} \cdots X_i^{\sigma_i} \cdots X_k^{\sigma_k} \right)$$

$$= \sup_{F \in S} \llbracket P \rrbracket (F)$$
As the induction hypothesis now assume that for some arbitrary, but fixed, programs \( P_1, P_2 \) and all increasing \( \omega \)-chains \( S_1, S_2 \) in FPS it holds that both

\[
\llbracket P_1 \rrbracket (\sup S_1) = \sup_{F \in S_1} \llbracket P_1 \rrbracket (F) \quad \text{and} \quad \llbracket P_2 \rrbracket (\sup S_2) = \sup_{F \in S_2} \llbracket P_2 \rrbracket (F).
\]

We continue with the induction step.

**The case** \( P = \{ P_1 \} [p] \{ P_2 \} \). We have

\[
\llbracket P \rrbracket (\sup S) = p \cdot \llbracket P_1 \rrbracket (\sup S) + (1 - p) \cdot \llbracket P_2 \rrbracket (\sup S)
\]

\[
= p \cdot \left( \sup_{F \in S} \llbracket P_1 \rrbracket (F) \right) + (1 - p) \cdot \left( \sup_{F \in S} \llbracket P_2 \rrbracket (F) \right) \quad \text{(I.H. on } P_1 \text{ and } P_2) \]

\[
= \left( \sup_{F \in S} p \cdot \llbracket P_1 \rrbracket (F) \right) + \left( (1 - p) \cdot \llbracket P_2 \rrbracket (F) \right) \quad \text{(scalar multiplication is defined point–wise)}
\]

\[
= \sup_{F \in S} (p \cdot \llbracket P_1 \rrbracket (F) + (1 - p) \cdot \llbracket P_2 \rrbracket (F)) \quad \text{(apply MST coefficient–wise.)}
\]

\[
= \sup_{F \in S} \llbracket \{ P_1 \} [p] \{ P_2 \} \rrbracket (F)
\]

\[
= \sup_{F \in S} \llbracket P \rrbracket (F).
\]

**The case** \( P = \text{if } (B) \{ P_1 \} \text{ else } \{ P_2 \} \). We have

\[
\llbracket P \rrbracket (\sup S) = \llbracket P_1 \rrbracket ((\sup S)\_B) + \llbracket P_2 \rrbracket ((\sup S)\_{\neg B})
\]

\[
= \llbracket P_1 \rrbracket \left( \sup_{F \in S} (F)_B \right) + \llbracket P_2 \rrbracket \left( \sup_{F \in S} (F)_{\neg B} \right) \quad \text{(restriction defined coefficient–wise)}
\]

\[
= \sup_{F \in S} \llbracket P_1 \rrbracket ((F)_B) + \sup_{F \in S} \llbracket P_2 \rrbracket ((F)_{\neg B}) \quad \text{(I.H. on } P_1 \text{ and } P_2) \]

\[
= \sup_{F \in S} \llbracket \text{if } (B) \{ P_1 \} \text{ else } \{ P_2 \} \rrbracket (F) \quad \text{(apply MST coefficient–wise)}
\]

\[
= \sup_{F \in S} \llbracket P \rrbracket (F).
\]

**The case** \( P = \text{while}(B)\{ P_1 \} \). Recall that for every \( G \in \text{FPS} \),

\[
\llbracket P \rrbracket (G) = \text{lfp} \ \Phi_{B,P_1} (G)
\]

\[
= \left( \sup_{n \in N} \Phi^n_{B,P_1} (0) \right) (G) \ .
\]

Hence, it suffices to show that

\[
\left( \sup_{n \in N} \Phi^n_{B,P_1} (0) \right) (\sup S) = \sup_{F \in S} \left( \left( \sup_{n \in N} \Phi^n_{B,P_1} (0) \right) (F) \right) .
\]
Assume for the moment that for every $n \in \mathbb{N}$ and all increasing $\omega$-chains $S$ in $\mathrm{FPS}$,

$$
\left(\Phi_{B,P_1}^n(0)\right)(\sup S) = \sup_{F \in S} \left(\Phi_{B,P_1}^n(0)\right)(F).
$$

We then have

$$
\left(\sup_{n \in \mathbb{N}} \Phi_{B,P_1}^n(0)\right)(\sup S) = \sup_{n \in \mathbb{N}} \left(\Phi_{B,P_1}^n(0)(\sup S)\right) = \sup_{n \in \mathbb{N}} \sup_{F \in S} \left(\Phi_{B,P_1}^n(0)(F)\right) = \sup_{F \in S} \left(\sup_{n \in \mathbb{N}} \left(\Phi_{B,P_1}^n(0)(F)\right)\right),
$$

which is what we have to show. It remains to prove Equation 1 by induction on $n$.

**Base case $n = 0$.** We have

$$
\left(\Phi_{B,P_1}^0(0)\right)(\sup S) = \sup_{F \in S} = \sup_{F \in S} \left(\Phi_{B,P_1}^0(0)\right)(F).
$$

**Induction step.** We have

$$
\left(\Phi_{B,P_1}^{n+1}(0)\right)(\sup S) = \Phi_{B,P_1} \left(\Phi_{B,P_1}^n(0)\right)(\sup S) = \langle\sup S\rangle_{\neg B} + \Phi_{B,P_1}^n(0)\left(\sup_{F \in S} \left[\sup_{n \in \mathbb{N}} \left(\Phi_{B,P_1}^n(0)(F)\right)\right]\right) = \langle\sup S\rangle_{\neg B} + \sup_{F \in S} \left(\Phi_{B,P_1}^n(0)(\sup_{n \in \mathbb{N}} \left(\Phi_{B,P_1}^n(0)(F)\right))\right) = \sup_{F \in S} \left(\Phi_{B,P_1}^{n+1}(0)(F)\right).
$$

This completes the proof.

**Theorem 2 (Well-definedness of FPS Semantics).** The semantics functional $\llbracket \cdot \rrbracket$ is well-defined, i.e., the semantics of any loop $\textbf{while} (B) \{P\}$ exists uniquely and can be written as

$$
\llbracket \textbf{while} (B) \{P\} \rrbracket = \text{lfp} \Phi_{B,P} = \sup_{n \in \mathbb{N}} \Phi_{B,P}^n(0).
$$
Proof. First, we show that the unfolding operator \( \Phi_{B,P} \) is \( \omega \)-continuous. For that, let \( f_1 \subseteq f_2 \subseteq \ldots \) be an \( \omega \)-chain in \( \text{FPS} \to \text{FPS} \). Then,

\[
\Phi_{B,P}\left(\sup_{n \in \mathbb{N}}\{f_n\}\right) = \lambda G. \langle G \rangle_{\neg B} + \left(\sup_{n \in \mathbb{N}}\{f_n\}\right) ([P] \langle \langle G \rangle_B \rangle)
\]

= \lambda G. \langle G \rangle_{\neg B} + \sup_{n \in \mathbb{N}}\{f_n([P] \langle \langle G \rangle_B \rangle)\}

(sup on \( \text{FPS} \to \text{FPS} \) is defined point-wise)

= \sup_{n \in \mathbb{N}}\{\lambda G. \langle G \rangle_{\neg B} + f_n([P] \langle \langle G \rangle_B \rangle)\}

(apply monotone sequence theorem coefficient-wise)

= \sup_{n \in \mathbb{N}}\{\Phi_{B,P}(f_n)\} \quad \text{(Def. of } \Phi_{B,P}\text{)}

Since \( \Phi_{B,P} \) is \( \omega \)-continuous and \( (\text{FPS} \to \text{FPS}, \subseteq) \) forms a complete lattice (Lemma 1), we get by the Kleene fixed point Theorem [21] that \( \Phi_{B,P} \) has a unique least fixed point given by \( \sup_{n \in \mathbb{N}} \Phi_{B,P}^n(0) \).

**Theorem 3 (Mass Conservation).** For every \( P \in \text{pGCL} \) and \( F \in \text{FPS} \), we have \( [|P](F) \leq |F| \).

Proof. By induction on the structure of \( P \). For the loop-free cases, this is straightforward. For the case \( P = \text{while}(B)\{P_1\} \), we proceed as follows. For every \( r \in \mathbb{R}_{\geq 0} \), we define the set

\[
\text{FPS}_r = \{ F \in \text{FPS} \mid |F| \leq r \}
\]

of all FPSs whose mass is at most \( r \). First, we define the restricted unfolding operator

\[
\Phi_{B,P_1,r} : (\text{FPS}_r \to \text{FPS}_r) \to (\text{FPS}_r \to \text{FPS}_r), \quad \psi \mapsto \Phi_{B,P_1}(\psi).
\]

Our induction hypothesis on \( P_1 \) implies that \( \Phi_{B,P_1,r} \) is well-defined.

It is now only left to show that \( (\text{FPS}_r, \preceq) \) is an \( \omega \)-complete partial order, because then \( \Phi_{B,P_1,r} \) has a least fixed point in \( \text{FPS}_r \) for every \( r \in \mathbb{R}_{\geq 0} \). The theorem then follows by letting \( r = |G| \), because

\[
(lfp \Phi_{B,P_1})(G) = (lfp \Phi_{B,P_1,|G|})(G) \quad \text{implies} \quad |(lfp \Phi_{B,P_1})(G)| \leq |G|.
\]

(\( \text{FPS}_r, \preceq \) is an \( \omega \)-complete partial order. The fact that \( (\text{FPS}_r, \preceq) \) is a partial order is immediate. It remains to show \( \omega \)-completeness. For that, let \( f_1 \preceq f_2 \preceq \ldots \) be an \( \omega \)-chain in \( \text{FPS}_r \). We have to show that \( \sup_{n} f_n \in \text{FPS}_r \), which is the case if and only if

\[
|\sup_{n} f_n| = \sum_{\sigma \in \mathbb{N}^k} \sup_{n} [\sigma]f_n \leq r.
\]
Now let \( g: \mathbb{N} \rightarrow \mathbb{N}^k \) be some bijection from \( \mathbb{N} \) to \( \mathbb{N}^k \). We have

\[
\sum_{\sigma \in \mathbb{N}^k} \sup_{n} [\sigma]_{f_n}^n = \sum_{i=0}^{\infty} \sup_{n} [g(i)]_{f_n}^n \quad \text{(series converges absolutely)}
\]

\[
= \sup_{N} \sum_{i=0}^{N} \sup_{n} [g(i)]_{f_n}^n \quad \text{(rewrite infinite series as supremum of partial sums)}
\]

\[
= \sup_{N} \sum_{i=0}^{N} [g(i)]_{f_n}^n \quad \text{(apply monotone sequence theorem)}
\]

\[
= \sup_{n} \sum_{i=0}^{N} [g(i)]_{f_n}^n \quad \text{(swap suprema)}
\]

Now observe that \( \sup_{N} \sum_{i=0}^{N} [g(i)]_{f_n}^n = |f_n| \), which is a monotonically increasing sequence in \( n \). Moreover, since \( f_n \in \text{FPS}_r \), this sequence is bounded from above by \( r \). Hence, the least upper bound \( \sup_{n} |f_n| \) of the sequence \( |f_n| \) is no larger than \( r \), too. This completes the proof. \( \square \)

### A.2 Proofs of Section 4.2

**Lemma A.1 (Representation of \([\text{while}]\)).** Let \( W = \text{while} (B) \{P\} \) be a \( p\text{GCL} \) program. An alternative representation is:

\[
[W] = \lambda G. \sum_{i=0}^{\infty} \langle \varphi^i(G) \rangle_{\sim B}, \quad \text{where } \varphi(G) = [P] \langle (G)_{B} \rangle.
\]

**Proof.** First we show by induction, that \( \Phi_{B,P}^n(0)(G) = \sum_{i=0}^{n-1} \langle \varphi^i(G) \rangle_{\sim B}. \)

**Base case.** We have

\[
\Phi_{B,P}^0(0)(G) = 0 \quad \text{and } \sum_{i=0}^{n-1} \langle \varphi^i(G) \rangle_{\sim B} = 0.
\]
Induction step. We have
\[
\begin{align*}
\Phi_{B,P}^{n+1}(0)(G) &= \Phi_{B,P}(\Phi_{B,P}^n(0)(G)) \\
&= \langle G \rangle_B + \Phi_{B,P}^n(0)(\|P\| \langle G \rangle_B) \\
&= \langle G \rangle_B + \Phi_{B,P}(\Phi_{B,P}^n(0)(\varphi(G))) \\
&= \langle G \rangle_B + \sum_{i=0}^{n-1} \langle \varphi^{i+1} \rangle_B \\
&= \langle G \rangle_B + \sum_{i=1}^{n} \langle \varphi^i \rangle_B \\
&= \sum_{i=0}^{\infty} \langle \varphi^i(G) \rangle_B.
\end{align*}
\]
Overall, we thus get
\[
\|W\|_B(G) = \sup_{n \in \mathbb{N}} \{ \Phi_{B,P}^n(0) \}(G) = \sup_{n \in \mathbb{N}} \{ \Phi_{B,P}(0)(G) \} \quad \text{(sup on FPS → FPS is defined point–wise)}
\]
\[
= \sup_{n \in \mathbb{N}} \left\{ \sum_{i=0}^{n} \langle \varphi^i(G) \rangle_B \right\} \quad \text{(see above)}
\]
\[
= \sum_{i=0}^{\infty} \langle \varphi^i(G) \rangle_B. \quad \square
\]

**Theorem 4 (Linearity of pGCL Semantics).** For every program \(P\) and guard \(B\), the functions \(\langle \cdot \rangle_B\) and \(\|P\|\) are linear. Moreover, the unfolding operator \(\Phi_{B,P}\) maps linear transformers onto linear transformers.

**Proof.** Linearity of \(\langle \cdot \rangle_B\). We have
\[
\langle a \cdot G + F \rangle_B = \left( a \cdot \sum_{\sigma \in \mathbb{N}^k} \mu_\sigma X^\sigma + \sum_{\sigma \in \mathbb{N}^k} \nu_\sigma X^\sigma \right)_B
\]
\[
= \left( \sum_{\sigma \in \mathbb{N}^k} (a \cdot \mu_\sigma + \nu_\sigma) X^\sigma \right)_B
\]
\[
= \sum_{\sigma \in B} (a \cdot \mu_\sigma + \nu_\sigma)X^\sigma
\]
\[
= \sum_{\sigma \in B} a \cdot \mu_\sigma X^\sigma + \sum_{\sigma \in B} \nu_\sigma X^\sigma
\]
\[
= a \cdot \langle G \rangle_B + \langle F \rangle_B. \quad \square
\]
The case \( P = \text{skip} \). We have
\[
\llbracket \text{skip} \rrbracket (r \cdot F + G) = r \cdot F + G = r \cdot \llbracket \text{skip} \rrbracket (F) + \llbracket \text{skip} \rrbracket (G)
\]

The case \( P = x_i := E \)
\[
\llbracket [x_i := E] (r \cdot F + G) = \sum_{\sigma \in \mathbb{N}^k} \llbracket \llbracket [x_i := E] \rrbracket_{F + G}, X^\sigma_1 \cdot \ldots \cdot X^\sigma_k \rrbracket \nabla_{P}(E)
\]
\[
= \sum_{\sigma \in \mathbb{N}^k} (r \cdot \llbracket \llbracket [x_i := E] \rrbracket_{F}, X^\sigma_1 \cdot \ldots \cdot X^\sigma_k \rrbracket \nabla_{P}(E)) \cdot X^\sigma_k
\]
\[
= r \cdot \sum_{\sigma \in \mathbb{N}^k} \left( \llbracket \llbracket [x_i := E] \rrbracket_{F}, X^\sigma_1 \cdot \ldots \cdot X^\sigma_k \rrbracket \nabla_{P}(E) \cdot X^\sigma_k \right)
\]
\[
= r \cdot \sum_{\sigma \in \mathbb{N}^k} \left( \llbracket \llbracket [x_i := E] \rrbracket_{F}, X^\sigma_1 \cdot \ldots \cdot X^\sigma_k \rrbracket \nabla_{P}(E) \cdot X^\sigma_k \right)
\]

Next, we consider the induction step.

The case \( P = P_1 ; P_2 \). We have
\[
\llbracket P_1 ; P_2 \rrbracket (r \cdot F + G)
\]
\[
= \llbracket [P_2] \rrbracket \llbracket [P_1] \rrbracket (r \cdot F + G)
\]
\[
= \llbracket [P_2] \rrbracket (r \cdot \llbracket [P_1] \rrbracket (F) + \llbracket [P_1] \rrbracket (G))
\]
\[
= r \cdot \llbracket [P_2] \rrbracket (\llbracket [P_1] \rrbracket (F) + \llbracket [P_1] \rrbracket (G))
\]

The case \( P = \text{if} \ B \{ P_1 \} \ \text{else} \ { P_2 } \). We have
\[
\llbracket [\text{if} \ B \{ P_1 \} \ \text{else} \ { P_2 } ] \rrbracket (r \cdot F + G)
\]
\[
= \llbracket [\text{if} \ B \{ P_1 \} \ \text{else} \ { P_2 } ] \rrbracket (r \cdot (F + G)) + \llbracket [\text{if} \ B \{ P_1 \} \ \text{else} \ { P_2 } ] \rrbracket (r \cdot F + G)
\]
\[
= r \cdot (\llbracket [P_1] \rrbracket (F) + \llbracket [P_2] \rrbracket (G)) + \llbracket [\text{if} \ B \{ P_1 \} \ \text{else} \ { P_2 } ] \rrbracket (r \cdot F + G)
\]

Linearity of \( \llbracket P \rrbracket \). By induction on the structure of \( P \). First, we consider the base cases.

The case \( P = \text{skip} \). We have
\[
\llbracket \text{skip} \rrbracket (r \cdot F + G) = r \cdot F + G = r \cdot \llbracket \text{skip} \rrbracket (F) + \llbracket \text{skip} \rrbracket (G)
\]
The case $P = \{ P_1 \} \{ P_2 \}$.

\[
\{ \{ P_1 \} \{ P_2 \} \} (r \cdot F + G) \\
= p \cdot \{ P_1 \} (r \cdot F + G) + (1 - p) \cdot \{ P_2 \} (r \cdot F + G) \\
= p \cdot (r \cdot \{ P_1 \} (F) + \{ P_1 \} (G)) + (1 - p) \cdot (r \cdot \{ P_2 \} (F) + \{ P_2 \} (G)) \quad \text{(I.H. on $P_1$ and $P_2$)} \\
= r \cdot (p \cdot \{ P_1 \} (F) + (1 - p) \cdot \{ P_2 \} (F)) + p \cdot \{ P_1 \} (G) + (1 - p) \cdot \{ P_2 \} (G) \quad \text{(reorder terms)} \\
= r \cdot \{ \{ P_1 \} \{ P_2 \} \} (F) + \{ \{ P_1 \} \{ P_2 \} \}
\]

The case $P = \text{while} (B) \{ P_1 \}$.

\[
\{ \text{while} (B) \{ P_1 \} \} (r \cdot F + G) \\
= \sup_{n \in \mathbb{N}} \{ \Phi^n_{B,P_1}(0) \} (r \cdot F + G) \\
= \sup_{n \in \mathbb{N}} \{ \Phi^n_{B,P_1}(0)(r \cdot F + G) \} \quad \text{(sup on FPS $\rightarrow$ FPS defined point-wise)} \\
= \sup_{n \in \mathbb{N}} \{ r \cdot \Phi^n_{B,P_1}(0)(F) + \Phi^n_{B,P_1}(0)(G) \} \\
= r \cdot \sup_{n \in \mathbb{N}} \{ \Phi^n_{B,P_1}(0)(F) \} + \sup_{n \in \mathbb{N}} \{ \Phi^n_{B,P_1}(0)(G) \} \quad \text{(by straightforward induction on $n$ using I.H. on $P_1$)} \\
= r \cdot \{ \text{while} (B) \{ P_1 \} \} (F) + \{ \text{while} (B) \{ P_1 \} \} (G)
\]

Linearity of $\Phi_{B,P}(f)$ for linear $f$.

\[
\Phi_{B,P}(f) \left( \sum_{\sigma \in \mathbb{N}^k} \mu_\sigma X^\sigma \right) = \left< \sum_{\sigma \in \mathbb{N}^k} \mu_\sigma X^\sigma \right> - B + f \left( \left< \sum_{\sigma \in \mathbb{N}^k} \mu_\sigma \{ X^\sigma \} \right>_B \right) \\
= \left< \sum_{\sigma \in \mathbb{N}^k} \mu_\sigma X^\sigma \right> - B + f \left( \sum_{\sigma \in \mathbb{N}^k} \mu_\sigma \{ X^\sigma \} \right)_B \quad \text{(1. & 2.)} \\
= \left< \sum_{\sigma \in \mathbb{N}^k} \mu_\sigma X^\sigma \right> - B + \sum_{\sigma \in \mathbb{N}^k} \mu_\sigma \cdot f \left( \{ X^\sigma \} \right)_B \quad \text{(f lin.)} \\
= \sum_{\sigma \in \mathbb{N}^k} \mu_\sigma \left< X^\sigma \right>_B + \sum_{\sigma \in \mathbb{N}^k} \mu_\sigma \cdot f \left( \{ X^\sigma \} \right)_B \\
= \sum_{\sigma \in \mathbb{N}^k} \mu_\sigma \cdot \Phi_{B,P}(f)(X^\sigma)
Second, we show that $\tau$ is monotone: Assume $G \mu \sqsubseteq G \mu'$.

$$\tau(G \mu) = \sum_{\sigma \in N^k} (\mu(\{\sigma\}) \cdot X_{\sigma}) = \lambda S$$

$$\sum_{\sigma \in S} (\mu(\{\sigma\}) \cdot X_{\sigma}) \leq \lambda S$$

$$\sum_{\sigma \in S} (\mu'(\{\sigma\}) \cdot X_{\sigma}) \quad \text{(as } \mu(\{\sigma\}) \leq \mu'(\{\sigma\}) \text{ per definition of } \sqsubseteq)$$

$$= \tau(G \mu')$$

Third, we show that $\tau^{−1}$ is monotone: Assume $\mu \sqsubseteq \mu'$.

$$\tau^{−1}(\mu) = \sum_{\sigma \in N^k} \mu(\{\sigma\}) \cdot X_{\sigma}$$

$$\sum_{\sigma \in N^k} \mu(\{\sigma\}) \cdot X_{\sigma} \leq \sum_{\sigma \in N^k} \mu'(\{\sigma\}) \cdot X_{\sigma} \quad \text{(as } \mu(\{\sigma\}) \leq \mu'(\{\sigma\}) \text{ per definition of } \sqsubseteq)$$

$$= \tau^{−1}(\mu')$$

\[\square\]

**Lemma A.4.** Let $f : (P, \leq) \to (Q, \leq)$ be a monotone isomorphism for any partially ordered sets $P$ and $Q$. Then, $f^* : Hom(P, P) \to Hom(Q, Q), \phi \mapsto f \circ \phi \circ f^{−1}$ is also a monotone isomorphism.

**Proof.** Let $f$ be such a monotone isomorphism, and $f^*$ the corresponding lifting.

First, we note that $f^*$ is also bijective. Its inverse is given by $(f^*)^{-1} = (f^{-1})^*$. Second, $f^*$ is monotone, as shown in the following calculation.

$$f \leq g \Rightarrow \forall x. f(x) \leq g(x)$$

$$\Rightarrow \forall x. \tau \circ f(\tau^{-1} \circ \tau(x)) \leq \tau \circ g(\tau^{-1} \circ \tau(x))$$

$$\Rightarrow \forall x. \tau^* \circ f(\tau(x)) \leq \tau^* \circ g(\tau(x))$$

$$\Rightarrow \tau^* \circ f \leq \tau^* \circ g$$

$$\Rightarrow \tau^*(f) \leq \tau^*(g)$$

\[\square\]

**Lemma A.5.** Let $P, Q$ be complete lattices, and $\tau$ a monotone isomorphism. Also let $lfp$ be the least fixed point operator. Then the following diagram commutes.

**Proof.** We show this by showing $\tau^{-1} \circ \tau = id$ and $\tau \circ \tau^{-1} = id$.

$$\tau^{-1} \circ \tau \left( \sum_{\sigma \in N^k} \alpha_{\sigma} X_{\sigma} \right) = \tau^{-1} \left( \lambda N. \sum_{\sigma \in N} \alpha_{\sigma} \right)$$

$$= \sum_{\sigma \in N^k} \sum_{\sigma \in \{\sigma\}} \alpha_{\sigma} \cdot X_{\sigma} = \sum_{\sigma \in N^k} \alpha_{\sigma} X_{\sigma}$$

$$\tau \circ \tau^{-1}(\mu) = \tau \left( \sum_{\sigma \in N^k} \mu(\{\sigma\}) \cdot X_{\sigma} \right) = \lambda N. \sum_{\sigma \in N} \mu(\{\sigma\}) = \mu(N) = \mu$$

\[\square\]

**Lemma A.3.** The mappings $\tau$ and $\tau^{-1}$ are monotone linear maps.

**Proof.** First, we show that $\tau^{-1}$ is linear (and hence $\tau$, due to bijectivity):

$$\tau^{-1}(\mu + \nu) = \sum_{\sigma \in N^k} (\mu + \nu)(\{\sigma\}) \cdot X_{\sigma}$$

$$= \sum_{\sigma \in N^k} (\mu(\{\sigma\}) + \nu(\{\sigma\})) \cdot X_{\sigma}$$

(as $M$ forms a vector space with standard $+$)

$$= \sum_{\sigma \in N^k} (\mu(\{\sigma\}) \cdot X_{\sigma} + \nu(\{\sigma\}) \cdot X_{\sigma})$$

$$= \left( \sum_{\sigma \in N^k} \mu(\{\sigma\}) \cdot X_{\sigma} \right) + \left( \sum_{\sigma \in N^k} \nu(\{\sigma\}) \cdot X_{\sigma} \right) = \tau^{-1}(\mu) + \tau^{-1}(\nu)$$
Second, we show that $\tau$ is monotone:

Assume $G_\mu \sqsubseteq G_{\mu'}$.

$$
\tau(G_\mu) = \tau \left( \sum_{\sigma \in \mathbb{N}^k} \mu(\{\sigma\}) \cdot X^\sigma \right) = \lambda S. \sum_{\sigma \in S} \mu(\{\sigma\}) \\
\leq \lambda S. \sum_{\sigma \in S} \mu'(\{\sigma\}) \\
\quad \text{(as $\mu(\{\sigma\}) \leq \mu'(\{\sigma\})$ per definition of $\sqsubseteq$)} \\
= \tau \left( \sum_{\sigma \in \mathbb{N}^k} \mu'(\{\sigma\}) \cdot X^\sigma \right) = \tau(G_{\mu'})
$$

Third, we show that $\tau^{-1}$ is monotone:

Assume $\mu \sqsubseteq \mu'$.

$$
\tau^{-1}(\mu) = \sum_{\sigma \in \mathbb{N}^k} \mu(\{\sigma\}) \cdot X^\sigma \\
\sqsubseteq \sum_{\sigma \in \mathbb{N}^k} \mu'(\{\sigma\}) \cdot X^\sigma \quad \text{(as $\mu(\{\sigma\}) \leq \mu'(\{\sigma\})$ per definition of $\sqsubseteq$)} \\
= \tau^{-1}(\mu') \quad \square
$$

**Lemma A.4.** Let $f : (P, \leq) \to (Q, \leq)$ be a monotone isomorphism for any partially ordered sets $P$ and $Q$. Then,

$$
f^* : \text{Hom}(P, P) \to \text{Hom}(Q, Q), \quad \phi \mapsto f \circ \varphi \circ f^{-1}
$$

is also a monotone isomorphism.

**Proof.** Let $f$ be such a monotone isomorphism, and $f^*$ the corresponding lifting.

First, we note that $f^*$ is also bijective. Its inverse is given by $(f^*)^{-1} = (f^{-1})^*$.

Second, $f^*$ is monotone, as shown in the following calculation.

$$
f \leq g \quad \Rightarrow \quad \forall x. \quad f(x) \leq g(x) \\
\quad \Rightarrow \quad \forall x. \quad \tau \circ f(\tau^{-1} \circ \tau(x)) \leq \tau \circ g(\tau^{-1} \circ \tau(x)) \\
\quad \Rightarrow \quad \forall x. \quad \tau^* \circ f(\tau(x)) \leq \tau^* \circ g(\tau(x)) \\
\quad \Rightarrow \quad \forall y. \quad \tau^* \circ f(y) \leq \tau^* \circ g(y) \\
\quad \Rightarrow \quad \tau^*(f) \leq \tau^*(g) \quad \square
$$

**Lemma A.5.** Let $P, Q$ be complete lattices, and $\tau$ a monotone isomorphism. Also let $\text{lfp}$ be the least fixed point operator. Then the following diagram commutes.
Theorem 5. The $\text{FPS}$ semantics of $p\text{GCL}$ is an instance of Kozen's semantics, i.e., for all $p\text{GCL}$ programs $P$, we have

$$\tau \circ J_P K = T(P) \circ \tau.$$  

Proof. Let $\varphi \in \text{Hom}(P, P)$ be arbitrary.

\[
\lf p \varphi = \inf \{ p \mid \varphi(p) = p \}, \quad 
\tau (\lf p \varphi) = \tau (\inf \{ p \mid \varphi(p) = p \})
\]

\[
= \inf \{ \tau(p) \mid \varphi(p) = p \}
\]

\[
= \inf \{ \tau(p) \mid \varphi(\tau^{-1} \circ \tau(p)) = \tau^{-1} \circ \tau(p) \}
\]

\[
= \inf \{ \tau(p) \mid \tau \circ \varphi(\tau^{-1} \circ \tau(p)) = \tau(p) \}
\]

\[
= \inf \{ q \mid \tau \circ \varphi(\tau^{-1}(q)) = q \}
\]

\[
= \lf p \tau^*(\varphi)
\]

\[\square\]

Definition 11. Let $\mathfrak{T}$ be the program translation from $p\text{GCL}$ to a modified Kozen syntax, defined inductively:

\[
\mathfrak{T}(\text{skip}) = \text{skip},
\]

\[
\mathfrak{T}(x_i := E) = x_i := f_E(x_1, \ldots, x_k),
\]

\[
\mathfrak{T}(\{P\} \mid p \{Q\}) = \{\mathfrak{T}(P)\} \mid p \{\mathfrak{T}(Q)\},
\]

\[
\mathfrak{T}(P_1; Q) = \mathfrak{T}(P); \mathfrak{T}(Q),
\]

\[
\mathfrak{T}(\text{if}(B) \{P\} \text{ else } \{Q\}) = \text{if } B \text{ then } \mathfrak{T}(P) \text{ else } \mathfrak{T}(Q) \text{ fi},
\]

\[
\mathfrak{T}(\text{while } (B) \{P\}) = \text{while } B \text{ do } \mathfrak{T}(P) \text{ od },
\]

where $p$ is a probability, $k = |\text{Var}(P)|$, $B$ is a Boolean expression and $P, Q$ are $p\text{GCL}$ programs. The extended construct $\text{skip}$ as well as $\{P\} \mid p \{Q\}$ is only syntactic sugar and can be simulated by the original Kozen semantics. The intended semantics of these constructs are

\[
[\text{skip}] = \text{id}
\]

and

\[
[\{P\} \mid p \{Q\}] = p \cdot \mathfrak{T}(P) + (1 - p) \cdot \mathfrak{T}(Q).
\]

Lemma A.6. For all guards $B$, the following identity holds: $e_B \circ \tau = \tau \circ (\cdot)_B$.

Proof. For all $G_\mu = \sum_{\sigma \in \mathbb{N}^*} \mu(\{\sigma\}) \cdot X^\sigma \in \text{FPS}$:

\[
e_B \circ \tau(G_\mu) = e_B(\mu)
\]

\[
= \lambda S. \mu(S \cap B)
\]

\[
\tau \circ (G_\mu)_B = \tau \left( \sum_{\sigma \in B} \mu(\{\sigma\}) \cdot X^\sigma + \sum_{\sigma \notin B} 0 \cdot X^\sigma \right)
\]

\[
= \lambda S. \mu(S \cap B)
\]
Proof. The proof is done via induction on the program structure. We omit the
loop-free cases, as they are straightforward.

By definition, Ξ(while (B) {P}) = while B do P od. Hence, the corre-
sponding Kozen semantics is equal to lfp T_B,P, where

\[ T : (M → M) → (M → M), \quad S ↦ e_B + (S ∘ P ∘ e_B). \]

First, we show that \( τ^* \circ T_{B,P} \circ τ^* = Φ_{B,P} \), where \( τ^* \) is the canonical lifting
of \( τ \), i.e., \( τ^*(S) = τ \circ S \circ τ^{-1} \) for all \( S ∈ (FPS → FPS) \).

\[
[τ^* \circ T_{B,P} \circ τ^*](S) = τ^* \circ T_{B,P} \circ τ \circ S \circ τ^{-1} = τ^* (e_B + τ \circ S \circ τ^{-1} \circ P \circ e_B) = τ^{-1} \circ e_B \circ τ + τ^{-1} \circ S \circ τ^{-1} \circ P \circ e_B \circ τ = τ^{-1} \circ e_B \circ τ + S \circ τ^{-1} \circ P \circ e_B \circ τ = τ^{-1} \circ S \circ τ^{-1} \circ P \circ e_B \circ τ = (S \circ τ^{-1} \circ P \circ τ \circ (S \circ τ^{-1} \circ P \circ τ)) \circ (S \circ τ^{-1} \circ P \circ τ) = (S \circ τ^{-1} \circ P \circ τ) \circ S \circ τ^{-1} \circ P \circ τ = Φ_{B,P}(S).
\]

Having this equality at hand, we can easily prove the correspondence of our
while semantics to the one defined by Kozen in the following manner:

\[
τ \circ \llbracket while (B) \{ P \} \rrbracket = Ξ(while (B) \{ P \}) \circ τ
\quad \Leftrightarrow \quad τ \circ \text{lfp } Φ_{B,P} = \text{lfp } T_{B,P} \circ τ
\quad \Leftrightarrow \quad \text{lfp } Φ_{B,P} = τ^{-1} \circ \text{lfp } T_{B,P} \circ τ
\quad \Leftrightarrow \quad \text{lfp } Φ_{B,P} = τ^{-1} \circ \text{lfp } T_{B,P} \circ τ\quad \text{(Definition of } τ^*)
\quad \Leftrightarrow \quad \text{lfp } Φ_{B,P} = \text{lfp } (τ^{-1} \circ T_{B,P} \circ τ^*) \quad \text{(cf. Lemma A.5)}
\quad \Leftrightarrow \quad \text{lfp } Φ_{B,P} = \text{lfp } Φ_{B,P} \quad \Box
\]

B Proofs of Section 5

Theorem 6 (Superinvariants and Loop Overapproximations). Let \( Φ_{B,P} \)
be the unfolding operator of while (B) {P} (cf. Def. 7) and \( \psi : FPS → FPS \). Then

\[
Φ_{B,P}(ψ) ⊆ ψ \text{ implies } \llbracket while (B) \{ P \} \rrbracket ⊆ ψ.
\]
Thus $\hat{f}$ is a superinvariant.

Verification Python Script. In the following code, we have manually constructed $\Phi, f$ and $h$. By using SymPy, we compute the differences $\Phi(f) - f$ and $\Phi(h) - h$.

In the latter case SymPy evaluates to 0 which means $h$ is an invariant. In the first case it outputs a polynomial with positive coefficients which means $\Phi(f) \succeq f$.

```python
from sympy import *

i n i t p r i n t i n g ( )

x , c = symbols ( 'x , c ' )

i , j = symbols ( ' i , j ' , integer=True)

# define the higher order transformer
def Phi ( f ) :
    return c /2 * ( f . subs ( i , i -1) + f . subs ( i , i +1) )

# define the loop invariant guess ( i \neq 0) case
f = c ** j * ( ( c / (1 - c * c) ) * ( i % 2) + (1/(1 - c * c) ) * ( ( i +1) % 2) )

# Second invariant :
h = c ** j * ( ( 1 - sqrt(1 - c * c) ) / c ) ** i

print ( "Invariant verified " if verify ( f ) else "Unknown" )
print ( "Invariant verified " if verify (h) else "Unknown" )
```

Program 1.5. Python program checking the invariants.

Proof of Example 6.

$\Phi_B,P(\hat{f})(X^iC^j) = \left( \langle X^iC^j \rangle_{i=0} + \hat{f} \left( \frac{1}{2} \langle X^iC^j \rangle_{i>0} \cdot \frac{C}{X} + \frac{1}{2} \langle X^iC^j \rangle_{i>0} \cdot XC \right) \right)$

- case $i = 0$: $\Rightarrow (C^j + \hat{h}(0)) = C^j = f(X^0C^j)$
- case $i > 0$: $\Rightarrow \frac{C}{2} \left( \hat{f}(X^{i-1}C^j) + \hat{f}(X^{i+1}C^j) \right) = C^j \cdot \begin{cases} 1/C^2, & \text{i even} \\ 1/C^2, & \text{i odd} \end{cases} = f(X^iC^j)$

$\Rightarrow \Phi_B,P(\hat{f})(X^iC^j) \subseteq f(X^iC^j)$.

Thus $\hat{f}$ is a superinvariant.

$\Phi_B,P(\hat{h})(X^iC^j) = \langle X^iC^j \rangle_{i=0} + \hat{h} \left( \frac{1}{2} \langle X^iC^j \rangle_{i>0} \cdot \frac{C}{X} + \frac{1}{2} \langle X^iC^j \rangle_{i>0} \cdot XC \right)$

- case $i = 0$: $\Rightarrow (C^j + \hat{h}(0)) = 1 = h(X^0C^j)$
- case $i > 0$: $\Rightarrow \frac{C}{2} \left( \hat{h}(X^{i-1}C^j) + \hat{h}(X^{i+1}C^j) \right)$

$$= \frac{C^{j+1}}{2} \cdot \left( \frac{1 - \sqrt{1 - C^2}}{C} \right)^{i-1} + \left( \frac{1 - \sqrt{1 - C^2}}{C} \right)^{i+1}$$

$$= C^j \cdot \left( \frac{1 - \sqrt{1 - C^2}}{C} \right)^i = h(X^iC^j)$$

$\Rightarrow \Phi_B,P(\hat{h})(X^iC^j) = h(X^iC^j)$
Thus \( \hat{f} \) is a superinvariant.

\[ \Phi_{B,P}(\hat{f})(X_i) = \langle X_i \rangle_{i=0} + i \cdot \hat{f}(\langle X_i \rangle_{i>0} \cdot 1_{X_i}) + (1 - \frac{1}{i}) \cdot \hat{f}(\langle X_i \rangle_{i>0} \cdot X_i) \]

**Verification Python Script.** In the following code, we have manually constructed \( \Phi, f \) and \( h \). By using SymPy, we compute the differences \( \Phi(f) - f \) and \( \Phi(h) - h \). In the latter case SymPy evaluates to 0 which means \( h \) is an invariant. In the first case it outputs a polynomial with positive coefficients which means \( \Phi(f) \subseteq f \).

```python
from sympy import *
init_printing()
x, c = symbols('x, c')
i, j = symbols('i, j', integer=True)
def Phi(f):
    return c/2 * (f.subs(i,i-1) + f.subs(i,i+1))
def compute_difference(f):
    return (Phi(f) - f).simplify()
def verify_poly(poly):
    print("Check coefficients for non-positivity:")
    for coeff in poly.coeffs():
        if coeff > 0:
            return False
    return True
# actual verification method
def verify(f):
    pprint(f)
    result = compute_difference(f)
    if result.is_zero:
        print("Invariant is a fixpoint!")
        return True
    else:
        print("Invariant is not a fixpoint - check if remainder is Poly")
        try:
            return verify_poly(Poly(result))
        except PolynomialFailed:
            print("Invariant is not a Poly!")
            return False
        except:
            print("Unexpected Error")
raise
# define the higher order transformer
# define polynomial verifier
# actual verification method
f = c*i**j * ((c / (1-c**2)) * (i % 2) + (1/(1-c**2)) * ((i+1) % 2))

# define the loop invariant guess (i != 0) case
h = c*i**j * ( ( 1 - sqrt(1 - c**2) ) / c )**i
```

**Program 1.5.** Python program checking the invariants
Proof of Example 7.

\[ \Phi_{B,P}(\hat{f})(X^i) = \langle X^i \rangle_{i=0} + \frac{1}{i} \cdot \hat{f} \left( \langle X^i \rangle_{i>0} \cdot \frac{1}{X} \right) + \left( 1 - \frac{1}{i} \right) \cdot \hat{f} \left( \langle X^i \rangle_{i>0} \cdot X \right) \]

case \( i = 0 \):
\[
= 1 + \infty \cdot \hat{f}(0) + -\infty \cdot \hat{f}(0)
= 1 + \infty \cdot 0 + -\infty \cdot 0 = 1 = f(X^i)
\]

case \( i > 0 \):
\[
= 0 + \frac{1}{i} \cdot \hat{f}(X^{i-1}) + \left( 1 - \frac{1}{i} \right) \cdot \hat{f}(X^{i+1})
\]
\[
= \frac{1}{i} \cdot \left( 1 - \frac{1}{e} \cdot \sum_{n=0}^{i-3} \frac{1}{n!} \right) + \left( 1 - \frac{1}{i} \right) \cdot \left( 1 - \frac{1}{e} \cdot \sum_{n=0}^{i-1} \frac{1}{n!} \right)
\]
\[
= \frac{1}{i} - \left( \frac{1}{e} \cdot \sum_{n=0}^{i-3} \frac{1}{n!} \right) + \left( 1 - \frac{1}{e} \cdot \sum_{n=0}^{i-1} \frac{1}{n!} \right) - \frac{1}{i} + \left( \frac{1}{e} \cdot \sum_{n=0}^{i-1} \frac{1}{n!} \right)
\]
\[
= \left( 1 - \frac{1}{e} \cdot \sum_{n=0}^{i-1} \frac{1}{n!} \right) + \frac{1}{e} \cdot \left( \frac{1}{(i-2)!} + \frac{1}{(i-1)!} \right)
\]
\[
= \left( 1 - \frac{1}{e} \cdot \sum_{n=0}^{i-1} \frac{1}{n!} \right) + \frac{1}{e(i-1)!} = \left( 1 - \frac{1}{e} \cdot \sum_{n=0}^{i-2} \frac{1}{n!} \right)
\]
\[
= f(X^i)
\]
\[ \implies \Phi_{B,P}(\hat{f})(X^i) = f(X^i) \]

Mathematica input query:

Input: \[
\frac{1}{k} \cdot \left( 1 - \frac{1}{e} \cdot \sum_{n=0}^{k-3} \frac{1}{n!} \right) + \left( 1 - \frac{1}{k} \right) \cdot \left( 1 - \frac{1}{e} \cdot \sum_{n=0}^{k-1} \frac{1}{n!} \right) - \left( 1 - \frac{1}{e} \cdot \sum_{n=0}^{k-2} \frac{1}{n!} \right)
\]

Output: 0
In order to reason about the probabilistic, cost and game aspects, we study the model of turn-based multiplayer stochastic games [4] where transitions contain multidimensional cost (weight) vectors representing quantities like discrete time and energy consumption. Our model can be seen as a weight extension of PRISM-games [5], where we consider properties formulated in an extension of alternating-time temporal logic (ATL) [6] that contains operators that specify existence of strategies for player coalitions ensuring cost- and probability bounded next or until properties. Hence we can ask questions like "is the probability that player 1 and 3 can form a coalition such that they enforce that a certain state is reachable within a total cost of \( c \), greater than 0.8?"

Our verification approach is based on a novel reduction to the problem of finding fixed points on abstract dependency graphs (ADG) [7,8], a recently introduced formalism that extends classical dependency graphs by Liu and Smolka [9]. Dependency graphs allow us to assign Boolean values to nodes in the graph, whereas ADGs assign to nodes values from a more abstract domain. In our case, we use the domain of the unit interval, representing probabilities, extended with a special value called "certain-zero" [10] that allows for an early termination of the on-the-fly computation of the fixed point on the ADG. We formally prove the correctness of our encoding and provide an efficient implementation that allows us to take as input the models described in PRISM-games and perform model checking in an on-the-fly manner. On three different PRISM-games case studies (annotated with the cost information), we demonstrate that our implementation is performance-wise comparable to the state-of-the-art model checker PRISM-games on queries that include exact probability bounds. However, once we lower the probability threshold from the exact probability bound, our on-the-fly algorithm demonstrates the potential of significantly outperforming PRISM-games.

**Abstract.** We design and implement an efficient model checking algorithm for alternating-time temporal logic (ATL) on turn-based multiplayer stochastic games with weighted transitions. This logic allows us to query about the existence of multiplayer strategies that aim to maximize the probability of game runs satisfying resource-bounded next and until logical operators, while requiring that the accumulated weight along the successful runs does not exceed a given upper bound. Our method relies on a recently introduced formalism of abstract dependency graphs (ADG) and we provide an efficient reduction of our model checking problem to finding the minimum fixed-point assignment on an ADG over the domain of unit intervals extended with certain-zero optimization. As the fixed-point computation on ADGs is performed in an on-the-fly manner without the need of a priori generating the whole graph, we achieve a performance that is comparable with state-of-the-art model checker PRISM-games for finding the exact solutions and sometimes an order of magnitude faster for queries that ask about approximate probability bounds. We document this on a series of scalable experiments from the PRISM-games benchmark that we annotate with weight information.

**Keywords:** model-checking, weighted stochastic games, dependency graphs

1 Introduction

Advances in model checking over the last decades allow us to verify larger systems using less resources. More recently, addition of quantitative aspects to model checking techniques became an important research topic. In order to model real-world applications, modelling formalisms must reflect both probabilistic choices [1] that model the uncertainties in system behaviour and at the same time be able to reason about quantitative aspects such as cost [2]. Moreover, in order to take into account the unpredictable environment, we need to verify that the desirable properties hold for all possible environmental behaviours. These aspects are usually modelled as games—in our case multiplayer games [3] where the players form coalitions in order to enforce a given property.
In order to reason about the probabilistic, cost and game aspects, we study
the model of turn-based multiplayer stochastic games [4] where transitions con-
tain multidimensional cost (weight) vectors representing quantities like discrete
time and energy consumption. Our model can be seen as a weight extension
of PRISM-games [5], where we consider properties formulated in an extension
of alternating-time temporal logic (ATL) [6] that contains operators that spec-
ify existence of strategies for player coalitions ensuring cost- and probability
bounded next or until properties. Hence we can ask questions like ”is the prob-
bility that player 1 and 3 can form a coalition such that they enforce that a
certain state is reachable within a total cost of c, greater than 0.8”?

Our verification approach is based on a novel reduction to the problem of
finding fixed points on abstract dependency graphs (ADG) [7,8], a recently intro-
duced formalism that extends classical dependency graphs by Liu and Smolka [9].
Dependency graphs allow us to assign Boolean values to nodes in the graph,
wheras ADGs assign to nodes values from a more abstract domain. In our case,
we use the domain of the unit interval, representing probabilities, extended with
a special value called ”certain-zero” [10] that allows for an early termination of
the on-the-fly computation of the fixed point on the ADG. We formally prove the
correctness of our encoding and provide an efficient implementation that allows
us to take as input the models described in PRISM-games and perform model
checking in an on-the-fly manner. On three different PRISM-games case studies
(annotated with the cost information), we demonstrate that our implementation
is performance-wise comparable to the state-of-the-art model checker PRISM-
games on queries that include exact probability bounds. However, once we lower
the probability threshold from the exact probability bound, our on-the-fly algo-
rihm demonstrates the potential of significantly outperforming PRISM-games.

Related Work Since the introduction of stochastic games in the seminal work
by Shapley in the 50’s [3], a large number of variations and extensions of the
classical formalism have been studied by researchers in the verification commu-
nity. From a theoretical perspective, Condon [11,12] studies the complexity and
algorithms for (simple) stochastic two-player games where the objective is to
determine the winning probability for a given player. More recently, [13,14] con-
sider controller synthesis for turn-based stochastic two-player games with PCTL
winning objectives. Compared to our work, these papers consider controller syn-
thesis instead of model-checking, and do not consider quantitative games and
offer no implementation.

For quantitative verification of turn-based stochastic multiplayer games, [15]
presents the logic rPATL (Probabilistic Alternating-Time Temporal Logic with
Rewards) that naturally extends the logic Probabilistic Alternating-Time Temp-
oral Logic [16] (PATL) with reward-operators. PATL is itself a probabilistic
extension of ATL. A similar logic is introduced in [17], interpreted on concurrent
games. The logic rPATL allows one to state that a coalition of players has a stra-
 tegy such that either the probability of an event happening or an expected reward
measure, is within a given threshold. Verifying rPATL properties on stochastic
multiplayer games has been implemented in PRISM-games [5]. PRISM-games
supports analysis of various types of games, verification of multi-objective properties \[18\] and has been applied to several case-studies (see e.g \[19\|15\]). Compared to our approach, PRISM-games does not directly support multidimensional reward-bounded properties and the current implementation offers no on-the-fly verification techniques that we demonstrate can yield a considerable speedup. Another approach to computing measures on probabilistic models with multi-dimensional rewards and non-determinism (MDPs) is presented in \[20\]. A performance comparison is left for the future work.

Lastly, our work is a continuation of the work done in \[21\], where a special-purpose algorithm is developed for PCTL model-checking on models with multi-dimensional weights. We lift the approach to games by showing how to formally treat the game features in ADGs and we consider a new set of domain values that treat the probabilities symbolically while the weights are encoded explicitly; our novel encoding outperforms the pure symbolic implementation provided in \[21\] by orders of magnitudes. Finally, our approach is more generic as it relies on the notion of ADGs and variations of the logic and/or the model can often be dealt with by modifications of the ADG construction, without the need of changing the underlying fixed-point algorithm.

### 2 Turn-based Stochastic Games

Before introducing turn-based stochastic games, we present some preliminaries. For any set \(X\), \(X^n\) is the set of all \(n\)-dimensional vectors with elements from \(X\) and \(x^n\) denotes the \(n\)-dimensional vector where \(x \in X\) is at all coordinates. Thus, \(\mathbb{N}^n\) is the set of all \(n\)-dimensional vectors of natural numbers and \(0^n\) is the 0-vector. We assume a fixed dimensionality \(n > 0\) and any vector is written in boldface e.g. \(\mathbf{x} = (x_1, \ldots, x_n)\) and \(\mathbf{y} = (y_1, \ldots, y_n)\) are vectors. For any such two vectors, we let \(\mathbf{x} \geq \mathbf{y}\) if and only if \(x_i \geq y_i\) for all \(1 \leq i \leq n\). For any countable non-empty set \(X\), we let \(D(X) = \{\mu: X \to [0, 1] \mid \sum_{x \in X} \mu(x) = 1\}\) denote the set of probability distribution on \(X\). For any distribution \(\mu \in D(X)\), the **support** of \(\mu\) is defined as \(\text{support}(\mu) = \{x \in X \mid \mu(x) > 0\}\). By \(D_{\text{fin}}(X) \subseteq D(X)\) we
denote the set of all distributions on $X$ with finite support. For any two sets $X$ and $Y$ we denote by $f : X \to Y$ that $f$ is a partial function from domain $\text{dom}(f) = X$ to range $\text{ran}(f) = Y$. For a set $X$, let $X^*$ be the set of all finite strings over $X$ and for any string $w = a_1a_2a_3\cdots a_n \in X^*$, let $|w| = n$ denote the length of $w$ and for all $1 \leq i \leq |w|$, let $w[i] = a_i$ be the $i$th symbol of $w$. The empty string is denoted by $\varepsilon$.

2.1 Definition of Stochastic Games

We now present turn-based stochastic multiplayer games $\mathcal{M}$, where the states are partitioned into a number of sets, each set owned by a player of the game. The game begins in a state owned by one of the players and proceeds in turns, by letting the owner of the current state play one of the available actions after which the game then transitions to the next state by a probabilistic choice. Each such transition has an associated cost vector, that can naturally be interpreted as the cost of the transition. Hence, given a strategy for each player in the game, any non-determinism is resolved and the induced model is what is known as a Markov reward model with impulse rewards $\{\mathcal{M}_1, \mathcal{M}_2\}$. It is a folklore result that deterministic strategies are sufficient (see e.g. [24]). We assume a fixed finite set of atomic propositions $\text{AP}$.

**Definition 1.** A Markov reward model (MRM) is a tuple $\mathcal{M} = (M, \to, \ell)$ where

- $M$ is a finite set of states,
- $\to : M \to \mathcal{P}(\mathbb{N} \times M)$ is the transition function and
- $\ell : M \to 2^{\text{AP}}$ is the labelling function.

For any state $m \in M$, the probability of transitioning to another state $m'$ with cost $w$ is given by $\to(m)(w, m')$. A $w$-successor of a state $m$ is any state $m'$ such that $\to(m)(w, m') > 0$. A path is an infinite sequence of transitions $\pi = (m_1, w_1, m_2), (m_2, w_2, m_3)\cdots$ where $s_{i+1}$ is a $w_i$-successor of $s_i$ for all $i \geq 1$. We let Paths$(m)$ denote the set of all paths starting in $m$ and for any path $\pi \in \text{Paths}(m)$ we let $\pi[i]$ denote the $i$th state of $\pi$ and by $\pi[n]$ denote the finite prefix of $\pi$ ending in state $\pi[n]$. We let $W(\pi)(j) = \sum_{i=1}^{j-1} w_i$ denote the accumulated cost up until the state $\pi[j]$. Finally, we let Paths$(\mathcal{M})$ be the set of all paths of $\mathcal{M}$. An example of an MRM can be seen in Figure [16].

In order to measure events of any MRM $\mathcal{M} = (M, \to, \ell)$, we introduce the classical cylinder set construction from [1] Chapter 10. For any finite sequence $w = (m_1, w_1, m_2), (m_2, w_2, m_3)\cdots(m_n-1, w_n-1, m_n)$, the cylinder set of $w$, $C(w)$ is the set of all paths having $w$ as a prefix, i.e., $C(w) = \{\pi \in \text{Paths}(\mathcal{M}) \mid \pi_n = w\}$ and the measure associated to the cylinder of $w$ is given by $\mathbb{P}_M(C(w)) = \prod_{i=1}^{n-1} \to(m_i)(w_i, m_{i+1})$. We can now define the probability space $(M^\omega, \Sigma, \mathbb{P}_M)$ where $\Sigma$ is the smallest $\sigma$-algebra that contains the cylinder sets of all finite alternating sequences of states and costs.

We are now ready to lift MRMs to stochastic games. Let Act be a fixed finite set of actions.
Definition 2. A turn-based stochastic multiplayer game is a structure \( \mathcal{G} = (\Pi, M, \{M_i\}_{i \in \Pi}, \rightarrow, \ell) \) where

- \( \Pi \) is a finite set of players,
- \( M \) is a finite set of states,
- \( \{M_i\}_{i \in \Pi} \) is a partition of \( M \) and for any \( i \in \Pi, M_i \) is a finite set of states controlled by player \( i \),
- \( \rightarrow : M \times \text{Act} \to \mathcal{D}(\mathbb{N}_n \times M) \) is the finite (partial) transition function, and
- \( \ell : S \to 2^\Pi \) is a labelling function, assigning atomic propositions to states.

For any state \( m \in M \) we let \( \text{Act}(m) = \{ \alpha \in \text{Act} \mid (m, \alpha) \in \text{dom}(\rightarrow) \} \) denote the set of enabled actions in state \( m \) and assume any game to be non-blocking by requiring all states to have at least one enabled action, i.e \( \text{Act}(m) \neq \emptyset \). An \( \alpha \)-successor of a state \( m \) is any state \( m' \) such that the probability of transitioning from \( m \) by playing the \( \alpha \) action is strictly positive for some cost vector \( w \in \mathbb{N}^n \), i.e \( \rightarrow(m, \alpha)(w, m') > 0 \). We let \( \text{succ}(m)_\alpha \) be the set of all \( \alpha \)-successors of \( m \). A path is an infinite sequence of transitions \( \pi = (m_1, \alpha_1, w_1, m_2, \alpha_2, w_2, m_3, \ldots) \) where \( m_{i+1} \) is an \( \alpha_i \)-successor of \( m_i \) with cost vector \( w_i \) for all \( i \geq 1 \). For any action \( \alpha \in \text{Act}(m) \) we let \( k = \min \{ w \mid \rightarrow(m, \alpha)(w, m', \ell(m')) > 0 \} \) be the smallest possible transition cost when playing action \( \alpha \) in \( m \) and say that \( \alpha \) is \( k \)-enabled in \( m \) whenever \( k' \geq k \) with \( \text{Act}_{\leq k}(m) \subseteq \text{Act}(m) \) being the set of all \( k' \)-enabled actions in \( m \). Thus, the set \( \text{Act}_{\leq k}(m) \) contains the actions available to the player owning state \( m \), if only transitions with a cost at most \( k \) are permitted. We extend the path notation introduced for MRM by letting \( \text{Paths}^*_i \) be the set of all finite paths that end in a state owned by player \( i \in \Pi \) and for any such finite path \( \pi \in \text{Paths}^*_i \), the last state is given by \( \text{last}(\pi) \).

Remark 1. Notice that if \( |\Pi| = 1 \), the resulting model is a Markov decision process (MDP) with impulse rewards and if furthermore \( |\text{Act}| = 1 \), the model is an MRM. Hence, turn-based stochastic multiplayer games subsume both MDPs and MRMs.

In the rest of the paper, we restrict the class of games, by assuming that the accumulated cost of any loop of any game is of strictly positive magnitude. Formally, for any state \( m \in M \), it is the case that for all paths \( \pi \in \text{Paths}(m) \) such that \( \pi[j] = m \) for some \( j \in \mathbb{N} \) (a loop), we have that \( \mathcal{W}(\pi)(j) \neq 0^n \).

Example 1. Figure 1 depicts a simple turn-based stochastic game \( \mathcal{G} \) with two players \( \Pi = \{\bigcirc, \blacklozenge\} \). The states depicted as circles, \( m_1 \) and \( m_3 \) belong to player \( \bigcirc \) while the state \( m_2 \) belongs to player \( \blacklozenge \). The transition function is depicted by edges labelled by a given enabled action, followed by the cost of the transition and probabilities to successor states. The labelling of each state is given next to the state. In case the probability distribution assigns probability 1 to a single state, there is no branching and we simply label the edge with the action, probability 1 and the associated weight.

Starting from the state \( m_1 \), player \( \bigcirc \) is in control and may choose either of the actions \( \beta \) and \( \alpha \). For \( \beta \), there is a small probability, \( \frac{1}{10} \), of transitioning to
state $m_3$ whereas for action $\alpha$, the game transitions to $m_2$ with probability $\frac{1}{2}$. In $m_2$, player $\Box$ may choose to let the game stay in state $m_2$ by the self-loop, or decide to transition to $m_3$.

If the two players are considered opponents and the goal of player $\bigcirc$ is to maximize the probability of reaching a state labelled $b$ ($m_3$) within a given bound on the accumulated cost of reaching $b$, the only safe option is to always choose the action $\beta$ in state $m_1$ as player $\Box$ can force the game to stay in state $m_2$ if it is ever reached. On the other hand, if the two players work together, player $\Box$ always plays the action $\beta$ in $m_2$ to ensure that state $m_3$ is reached.

### 2.2 Strategies

As indicated by Example 4, any game unfolds by applying concrete strategies for each player, specifying which action to play in a given state. We now formally define strategies by first fixing a game $G = (H, M, \{M_i\}_{i \in H}, \to, \ell)$. Given a player, $i \in H$, a (history-dependent deterministic) strategy for player $i$ in $G$ is a function $\sigma : \text{Paths}_i^* \to \text{Act}$, that associates an action with each finite path ending in a state owned by player $i$. Thus, a strategy prescribes which action a player should play in a given state, given the full history of the game. For a strategy to be sound, only actions enabled in the given state must be played. Formally, a strategy $\sigma$ for player $i$ is sound if for any finite path $\pi \in \text{Paths}_i^*$ with $\text{last}(\pi) = m_i \in M_i$, it holds that $\sigma(\pi) \in \text{Act}(m_i)$. We let $\mathcal{S}_i$ denote the set of all sound strategies for player $i$ in $G$.

**Remark 2.** If $\sigma(\pi_1) = \sigma(\pi_2)$ for all $\pi_1, \pi_2 \in \text{Paths}_i^*$ with $\text{last}(\pi_1) = \text{last}(\pi_2)$, we say that $\sigma$ is a memoryless strategy for player $i$, as the action prescribed depends only on the last state of the game.

Strategies naturally extend to sets of players by considering what is commonly known as a coalition of players. A coalition strategy for any coalition $C \subseteq H$ in $G$, is a set of sound strategies, $\{\sigma_i\}_{i \in C}$, such that $\sigma_i \in \mathcal{S}_i$ for all $i \in C$. We let $\mathcal{S}_C$ denote the set of all coalition strategies for the coalition $C$, use $\sigma_C$ to range over elements of $\mathcal{S}_C$ and let $\mathcal{C} = H \setminus C$ be the coalition containing the players in the complement of $C$. Given a state $m \in M$, coalition strategies $\sigma_C$ and $\sigma_\mathcal{C}$, a unique MRM is induced from $G$ by resolving the non-deterministic choices as prescribed by $\sigma_C$ and $\sigma_\mathcal{C}$. We let $\mathbb{P}^{\sigma_C, \sigma_\mathcal{C}}$ denote the probability measure on the induced MRM.

**Example 2.** Consider again the game from Figure 1a and the memoryless strategies $\sigma_\bigcirc^\alpha$ and $\sigma_\bigcirc^\beta$, respectively defined for any $\pi_\bigcirc \in \text{Paths}_\bigcirc^*$ and $\pi_\bigcirc \in \text{Paths}_\bigcirc^*$ as $\sigma_\bigcirc^\alpha(\pi_\bigcirc) = \alpha$ and $\sigma_\bigcirc^\beta(\pi_\bigcirc) = \beta$. The induced MRM is the one depicted in Figure 1b.

### 3 Probabilistic Weighted ATL

As a specification language, we employ an extension of Alternating-time Temporal Logic (ATL [6]) to reason about whether or not a given coalition of players
can together enforce the game to enjoy a given property, regardless of the strategy of the remaining players of the game. Hence, a witness of satisfaction is a coalition-strategy. Our logic is similar to probabilistic resource-bounded ATL, recently proposed by Nguyen and Rakib [14] and can also be seen as a sub-logic of the logic rPATL [15] employed by PRISM-games, where expected reward measures are omitted. We restrict negation to atomic propositions.

**Definition 3 (Syntax).** The set of PWATL formulae is given by the following grammar:

\[ \phi ::= a \mid \neg a \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle C \rangle_{\lambda}[\psi] \]  
(State Formulae)

\[ \psi ::= X_{\leq k} \phi \mid \phi U_{\leq k} \phi \]  
(Path Formulae)

where \( a \in AP, C \subseteq \Pi, \lambda \in \{0, 1\}, k \in \mathbb{N}^n \) and \( \triangleright = \{>, \geq\} \).

The set of PWATL state-formulae is denoted by \( \mathcal{L}_{ATL} \). A formula \( \langle C \rangle_{\lambda}[\psi] \in \mathcal{L}_{ATL} \) is satisfied by a state \( m \in M \) of a game \( G = (\Pi, M, \{M_i\}_{i \in \Pi}, \rightarrow, \ell) \), if there exists a coalition strategy \( \sigma_C \) for the players in \( C \subseteq \Pi \) such that, no matter which coalition strategy \( \sigma_{\neg C} \) is assigned to the remaining players in \( \neg C \), measuring paths that satisfy \( \psi \) in the MRM induces from \( G \) by \( \sigma_C \) and \( \sigma_{\neg C} \), yields a probability \( p \) such that \( p \triangleright \lambda \).

**Definition 4 (Semantics).** For a given game \( G = (\Pi, M, \{M_i\}_{i \in \Pi}, \rightarrow, \ell) \), state \( m \in M \), and path \( \pi \in \text{Paths} \), the satisfaction of PWATL formulae is given inductively as follows:

\[ G, m \models a \quad \text{iff} \quad a \in \ell(m) \]

\[ G, m \models \neg a \quad \text{iff} \quad a \notin \ell(m) \]

\[ G, m \models \phi_1 \wedge \phi_2 \quad \text{iff} \quad G, m \models \phi_1 \text{ and } G, m \models \phi_2 \]

\[ G, m \models \phi_1 \vee \phi_2 \quad \text{iff} \quad G, m \models \phi_1 \text{ or } G, m \models \phi_2 \]

\[ G, m \models \langle C \rangle_{\lambda}[\psi] \quad \text{iff} \quad \exists \sigma_C \in \mathcal{S}_C. \forall \sigma_{\neg C} \in \mathcal{S}_{\neg C}. \]

\[ \mathcal{P}^{G, \sigma_C, \sigma_{\neg C}}(\{ \pi \in \text{Paths}(m) \mid G, \pi \models \psi \}) \triangleright \lambda \]

\[ G, \pi \models \phi_1 U_{\leq k} \phi_2 \quad \text{iff} \quad \exists j \in \mathbb{N}, G, \pi[j] \models \phi_2, W(\pi)(j) \leq k \]

\[ \text{and } G, \pi[i] \models \phi_1 \text{ for all } i < j \]

\[ G, \pi \models X_{\leq k} \phi \quad \text{iff} \quad G, \pi[2] \models \phi \text{ and } W(\pi)(1) \leq k \]

**Example 3.** Consider once again the game in Figure 1a and the formula \( \phi = \langle C \rangle_{\geq 1}[a U_{\leq 8} b] \) with \( C = \{\Box, \Diamond\} \) By the memoryless strategies from Example 2 we have

\[ \mathcal{P}^{G, \sigma_C, \sigma_{\neg C}}(\{ \pi \in \text{Paths}(m_1) \mid G, \pi \models a U_{\leq 8} b \}) = \frac{1}{2} \]

where \( \sigma_C = \{\sigma_C^a, \sigma_C^b\} \). This is easily verified by inspecting the induced MRM in Figure 1b. Hence, the two memoryless strategies do not prove \( G, m_1 \models \phi \).

To construct a strategy that proves \( G, m_1 \models \phi \), we modify the strategy for player \( \Box \). Instead of always playing action \( a \), the choice of action depends on the
accumulated cost of the finite history: for any finite path \( \pi_0 \in \text{Paths}_\omega \) of length \( j \), the strategy \( \sigma_0^j \) is defined as

\[
\sigma_0^j(\pi_0) = \begin{cases} 
\beta & \text{if } W(\pi_0)(j) \leq 4 \\
\alpha & \text{otherwise}
\end{cases}
\]

4 Model Checking Through Dependency Graphs

In this section we demonstrate how the PWATL model-checking problem for turn-based stochastic multiplayer games can be reduced to computing fixed points on so-called abstract dependency graphs \([7]\). For a model-checking problem \( G, m \models \phi \), the corresponding abstract dependency graph represents the decomposition of the problem into sub-problems (dependencies) given by the inductive definition of PWATL semantics.

4.1 Abstract Dependency Graphs

An abstract dependency graph \([7]\) is a (directed) graph consisting of a collection of vertices \( V \), together with a function that to each \( v \in V \) assigns a set of vertices being the dependencies of \( v \) and a function for computing the value of \( v \), given the value of all its dependencies. The vertex values are drawn from a triple \( D = (D, \sqsubseteq, \perp) \) where \( (D, \sqsubseteq) \) is a partial order, \( \perp \in D \) the least element of \( D \) and \( \sqsubseteq \) must satisfy the ascending chain condition: for any infinite chain \( d^1 \sqsubseteq d^2 \sqsubseteq d^3 \ldots \) of elements \( d^i \in D \), there exists an integer \( k \) such that \( d^k = d^{k+j} \) for all \( j > 0 \). This kind of ordering is referred to in \([7]\) as a Noetherian ordering relation with least element (NOR). For any NOR we assume the elements are finitely representable, meaning that elements can be represented by finite strings.

For the computation of the value of each vertex we consider the application of monotone functions to the values of all its dependencies. Formally, for any \( n \in \mathbb{N} \), \( F(D, n) \) on a NOR \( (D, \sqsubseteq, \perp) \) is the set of all monotone functions \( f : D^n \to D \) of arity \( n \), where \( f \) is monotone if \( d_i \sqsubseteq d'_i \) for all \( i, 1 \leq i \leq n \), implies \( f(d_1, \ldots, d_n) \sqsubseteq f(d'_1, \ldots, d'_n) \) for any \( d_1, \ldots, d_n, d'_1, \ldots, d'_n \in D \), and we let \( F(D) = \bigcup_{n \geq 0} F(D, n) \) be the collection of all such functions. We assume all functions \( f \in F(D, n) \) for any \( n \in \mathbb{N} \) to be effectively computable, meaning that for any \( f \in F(D, n) \) and \( d_1, \ldots, d_n \in D \), there exists an algorithm that terminates and computes the finite representation of \( f(d_1, \ldots, d_n) \in D \).

We are now ready to define abstract dependency graphs.

**Definition 5 (Abstract Dependency Graph \([7]\)).** An abstract dependency graph (ADG) is a tuple \( G = (V, E, \mathcal{D}, \mathcal{E}) \) where

- \( V \) is a finite set of vertices,
- \( E : V \to V^* \) is an edge function from vertices to sequences of vertices such that \( E(v)[i] \neq E(v)[j] \) for every \( v \in V \) and every \( 1 \leq i < j \leq |E(v)| \), i.e. the co-domain of \( E \) contains only strings over \( V \) where no symbol appears more than once,
- $\mathcal{D}$ is NOR with finitely representable elements, and
- $\mathcal{E}$ is a labelling function $\mathcal{E} : V \rightarrow \mathcal{F}(\mathcal{D})$ such that $\mathcal{E}(v) \in \mathcal{F}(\mathcal{D},|E(v)|)$ for each $v \in V$, i.e. each edge $E(v)$ is labelled by an effectively computable monotone function $f$ of arity that corresponds to the length of $E(v)$.

In the following, we assume a fixed ADG $G = (V,E,\mathcal{D},\mathcal{E})$. For each vertex $v \in V$, $E(v)$ is a string containing all the vertices that represent dependencies of $v$ and $\mathcal{E}(v)$ is the function computing the value of $v$ given the values of all the dependencies of $v$ in $E(v)$. An assignment is then a function $A : V \rightarrow D$, mapping each vertex to an element of the NOR $\mathcal{D} = (D,\sqsubseteq,\bot)$. We let $\mathfrak{A}$ denote the set of all assignments and lift the ordering from $\mathcal{D}$ to assignments: for any two assignments $A_1,A_2 \in \mathfrak{A}$, $A_1 \sqsubseteq A_2$ iff $\forall v \in V, A_1(v) \sqsubseteq A_2(v)$. It follows that $(\mathfrak{A},\sqsubseteq)$ is a NOR, with minimum element $A_\bot$ defined for any $v \in V$ as $A_\bot(v) = \bot$. We define the minimum fixed-point assignment $A_{\text{min}}$ for $G$ as the minimum fixed point of the function $F : \mathfrak{A} \rightarrow \mathfrak{A}$, defined for any $v \in V$ as $F(A)(v) = \mathcal{E}(v)(A(v_1),A(v_2),\ldots,A(v_k))$ where $E(v) = v_1v_2\cdots v_k$. As each $\mathcal{E}(v)$ is monotone, it follows that $F$ is a monotone function. In [7] it is proven, by applying standard reasoning for fixed points of monotonic functions [22], that $A_{\text{min}}$ exists and is computable by repeated application of $F$ on $A_\bot$. We end this section by presenting the result of [7]. For any $A \in \mathfrak{A}^k$ let $F^i(A)$ be the $i$th repeated application of $F$ on $A$, defined for $i = 0$ as $F^0(A) = A$ and $F^i(A) = F(F^{i-1}(A))$ for $i > 0$.

**Theorem 1** ([7]). There exists $j \in \mathbb{N}$ such that $F^k(A_\bot) = A_{\text{min}}$ for all $k \geq j$.

### 4.2 The Reduction

We fix a game $G = (\Pi,M,\{M_i\}_{i \in \Pi},\rightarrow,\ell)$ for the remainder of this section and present the encoding of the problem $G, m \models \phi$ for some state $m \in M$ and PWATL formula $\phi \in \mathcal{L}_{\text{ATL}}$ by reduction to computing the minimal fixed point of a suitable abstract dependency graph $G = (V,E,\mathcal{D},\mathcal{E})$. In general, vertices of the graph are pairs $(m,\phi)$ where $m$ is a state of $G$ and $\phi \in \mathcal{L}_{\text{ATL}}$ is a state-formula. These are referred to as concrete vertices. As our approach is symbolic, we introduce another type of vertex. For this, we let $\mathcal{L}_{\text{ATL}}' = \{\langle C \rangle \models [\phi_1 U_{\leq k} \phi_2] | k \in \mathbb{N}, \phi_1,\phi_2 \in \mathcal{L}_{\text{ATL}}\} \cup \{\langle C \rangle \models [X_{\leq k} \phi] | k \in \mathbb{N}, \phi \in \mathcal{L}_{\text{ATL}}\}$ be the set of all symbolic state-formulae. The symbolic vertices are then on the form $(m,\phi_\gamma)$, where $\phi_\gamma \in \mathcal{L}_{\text{ATL}}'$. We proceed by defining the domain $\mathcal{D}$.

**The domain $\mathcal{D}$** During the fixed point computation, the value of any node is, in general, a number that represents a lower bound on the probability of satisfaction. However, as we employ the certain-zero optimization of [10], we use also a special value $\overline{0}$, indicating that the value is 0 and can never change. Hence, 0 is a lower bound whereas $\overline{0}$ is an upper bound on the probability of satisfaction. We define the ordering depicted in Figure 3, where the dotted line represent all number between 0 and 1: $0 \sqsubseteq \overline{0}$

![Fig. 3: Ordering $\sqsubseteq$](image-url)
rules define the edge function \( E \) for any vertex on the form \( (v, \phi) \). Hence, \( E \) is also a dependency, if the cost of transitioning to \( v \) must be \( 0 \) or \( 1 \) and the rest of the graph is constructed by induction on \( V \). In order to satisfy the inner path formula \( \phi \), i.e., each successor receives a Boolean value of \( 0 \) or \( 1 \). As the value of \( \phi \) is \( 0 \), it is certain that \( m \) does not satisfy \( \phi \) and the algorithm terminates. For symbolic vertices \( (m, \phi) \), assigning a probability \( p \) to the vertex indicates that there exists a strategy for the coalition \( C \), such that measuring paths from \( m \) that satisfy the path-formula \( \psi \), yields a probability at least \( p \), no matter the strategy for the remaining players in \( \overline{C} \). Hence, \( \mathcal{G}, m \models (C)_{\phi p} \psi \).

Anticipating the definition of the vertex labelling function, we define the operations \( \min, \max, + \) and \( \cdot \) on elements from the domain \( \mathcal{D} \). If the operands are regular probabilities in \([0, 1]\), the operations are defined in the natural way. Otherwise, for the certain zero value \( 0 \) and for any probability \( p \in [0, 1] \) we let \( \min\{0, p\} = 0 \), \( \max\{0, p\} = p \), \( 0 + p = p \) and \( 0 \cdot p = 0 \). Hence, \( 0 \) behaves like \( 0 \) when used in operations with regular probabilities. If both operands are \( 0 \) we let \( \min\{0, 0\} = 0 \), \( \max\{0, 0\} = 0 \), \( 0 + 0 = 0 \) and \( 0 \cdot 0 = 0 \).
Graph construction We define the set of vertices $V$ and for each $v \in V$, the edge function $E(v)$ and labelling function $\mathcal{E}(v)$. The root of the graph is $(m, \phi) \in V$ and the rest of the graph is constructed by induction on $\phi$.

For any vertex on the form $v = (m_\ast, \phi_\ast)$, where $\phi_\ast \in \mathcal{L}_{\text{ATL}}$, the following rules define the edge function $E(v)$ and labelling function $\mathcal{E}(v)$.

- $[\phi_\ast = a ]$: The formula has no dependencies and can be verified directly by inspecting the labelling of the state. Hence, $E(v) = \varepsilon$ and if $a \notin \ell(m_\ast)$ then $\mathcal{E}(v) = 1$, otherwise $\mathcal{E}(v) = 0$.

- $[\phi_\ast = \neg a ]$: We let $E(v) = \varepsilon$, $\mathcal{E}(v) = 1$ if $a \notin \ell(m_\ast)$ and $\mathcal{E}(v) = 0$ otherwise.

- $[\phi_\ast = \phi_1 \lor \phi_2 ]$: We let the vertices $(m_\ast, \phi_1), (m_\ast, \phi_2) \in V$ be the dependencies of $v$, hence $E(v) = (m_\ast, \phi_1)(m_\ast, \phi_2)$. As each successor receives a Boolean value, disjunction is naturally defined as the maximum of the values of the two successor vertices and we let $\mathcal{E}(v)(p_1, p_2) = \max\{p_1, p_2\}$.

- $[\phi_\ast = \langle\langle C \rangle\rangle_{\lambda} \phi_1 U_{\leq k} \phi_2 ]$: We let the symbolic vertex $v' = (m_\ast, \langle\langle C \rangle\rangle_{\lambda} \phi_1 U_{\leq k} \phi_2) \in V$ be the dependency of $v$, i.e. $E(v) = v'$. As the value of $v'$ is the probability $p$ of satisfying the inner path formula, we let the value of $v$ be 1 if and only if $p \triangleright \lambda$:

$$
\mathcal{E}(v)(p) = \begin{cases} 
1 & \text{if } p \triangleright \lambda \\
0 & \text{if } p = 0 \land (\lambda > 0) \\
0 & \text{otherwise}
\end{cases}
$$

- $[\phi_\ast = \langle\langle C \rangle\rangle_{\lambda} X_{\leq k} \phi ]$: We let the symbolic vertex $v' = (m_\ast, \langle\langle C \rangle\rangle_{\lambda} X_{\leq k} \phi) \in V$ be the dependency of $v$, i.e. $E(v) = v'$. The labelling of $v$ is given by:

$$
\mathcal{E}(v)(p) = \begin{cases} 
1 & \text{if } p \triangleright \lambda \\
0 & \text{if } p = 0 \land (\lambda > 0) \\
0 & \text{otherwise}
\end{cases}
$$

For any vertex $v = (m_\ast, \phi_\ast)$ with $\phi_\ast \in \mathcal{L}_{\text{ATL}}$, the edge function $E(v)$ and labelling function $\mathcal{E}(v)$ are given by the following rules:

- $[\phi_\ast = \langle\langle C \rangle\rangle_{\lambda} \phi_1 U_{\leq k} \phi_2 ]$: In order to satisfy the inner path formula $\phi_1 U_{\leq k} \phi_2$ for any path starting in $m_\ast$, either $\phi_2$ must be satisfied by $m_\ast$ or $\phi_1$ must be satisfied by $m_\ast$. Hence, we let $v_1 = (m_\ast, \phi_1), v_2 = (m_\ast, \phi_2)$ with $v_1, v_2 \in V$ be the two immediate dependencies of $v$. In case $\phi_2$ is not satisfied by $m_\ast$ but $\phi_1$ is, the satisfaction of the inner path formula is due to the successors of $m_\ast$. Hence, any successor of $m_\ast$ is also a dependency, if the cost of transitioning to the successor is within the formula bound $k$. We let $\text{Act}_k(m_\ast) = \{\alpha_1, \ldots, \alpha_n\}$ be the $k$-enabled actions in $m_\ast$ and for any $\alpha_k \in \text{Act}_k(m_\ast)$ let $\text{succ}(m_\ast)_{\alpha_k} =$
We start from an assignment to compute the minimal fixed point assignment to the remaining vertices. Hence, if all assignments have been pre-computed, we now repeatedly apply the fixed point operator \( q \).

Given a monotonic function \( q \) for a symbolic node, responsible for computing a weighted minimum, \( q \) is given by our encoding. The greyed out shapes are not vertices but part of the encoded model-checking problem \( G \). Edges connecting the vertices correspond to the specific monotone functions \( E \).

Let \( G \) be a graph, \( V \) a set of vertices, \( E \) a set of edges. For any state \( v \) of \( G \), the empty set. In general, separate unlabelled edges encode a maximum, while a non-empty set encodes the model-checking problem \( G \) with a simply connected subgraph. For vertex \( v \), let \( E(v) \) be a set of edges. For \( v \), \( E(v) \) is visualised by vertices having either no outgoing edge or an edge pointing to vertex \( v \).

For any vertex \( v \), let \( q(v) \) be the function computed at \( v \) such that \( q(v) = \max \{ q(v), q(v_{\text{successor}}) \} \). If \( q(v) = 0 \), then \( v \) is visualised by a vertex having no incoming edge.

Theorem 2 (Correctness). The result can be seen in Figure 2b after 3 iterations, the fixed point has been computed with a value of 1 assigned to vertex \( v \) and \( v' \) such that \( v' = q(v) \).

For defining the labelling of \( v \) as \( E(v) = \{ q_1, q_2, q_1^a, \ldots, q_j^a, \ldots, q_{j_n}^a \} \), we let \( q_{i}^a = \sum_{i=1}^{j_n} p_i \cdot q_i^a \) be the weighted sum of successor values for any action \( \alpha_i \in \text{Act}_k(m) \). The exact labelling function of \( m \) depends on whether \( m \) is owned by a player in the coalition or not.

If \( m \in M_i \) for some player \( i \) we let

\[
E(v)(q_1, q_2, q_1^a, \ldots, q_j^a, \ldots, q_{j_n}^a) = \max \{ q_2, \min \{ q_1, q_1^a \}, \ldots, \min \{ q_1, q_j^a \} \}.
\]

Otherwise, if \( m \notin M_i \) for all players \( i \) we let

\[
E(v)(q_1, q_2, q_1^a, \ldots, q_j^a, \ldots, q_{j_n}^a) = \min \{ q_2, \min \{ q_1, q_1^a \}, \ldots, q_{j_n}^a \}.
\]

\( \phi = \{ \langle C \rangle_{\gamma_2}(X_{\leq k} \phi) \} \) Let \( \text{Act}_k(m) = \{ \alpha_1, \ldots, \alpha_n \} \) be the set of \( k \)-enabled actions in \( m \) and for any \( \alpha_i \in \text{Act}_k(m) \) let \( \text{succ}(m)_{\alpha_i} = \{ m_1^\alpha, \ldots, m_{j_n}^\alpha \} \) be the set of all \( \alpha_i \)-successors of \( m \) where, for all \( 1 \leq i \leq j_n \), \( w_i^\alpha < k \) is the cost and \( p_i^\alpha \) is the probability of transitioning to \( m_i^\alpha \), respectively.

For each \( m_i^\alpha \) we let \( v_i^\alpha = (m_i^\alpha, \langle C \rangle_{\gamma_2}(\phi_1 U_{\leq k} - w_i^\alpha \phi_2)) \) \( \in V \) be a dependency of \( m \). Hence, the edge function of \( v \) is given as

\[
E(v) = v_1 \cdot v_1^a \cdot v_{j_n}^a.
\]

For defining the labelling of \( v \) as \( E(v) = \{ q_1, q_2, q_1^a, \ldots, q_j^a, \ldots, q_{j_n}^a \} \), we let \( q_{i}^a = \sum_{i=1}^{j_n} p_i \cdot q_i^a \) be the weighted sum of successor values for any action \( \alpha_i \in \text{Act}_k(m) \). The exact labelling function of \( m \) depends on whether \( m \) is owned by a player in the coalition or not.

If \( m \in M_i \) for some player \( i \) we let

\[
E(v)(q_1, q_2, q_1^a, \ldots, q_j^a, \ldots, q_{j_n}^a) = \max \{ q_2, \min \{ q_1, q_1^a \}, \ldots, \min \{ q_1, q_j^a \} \}.
\]

Otherwise, if \( m \notin M_i \) for all players \( i \) we let

\[
E(v)(q_1, q_2, q_1^a, \ldots, q_j^a, \ldots, q_{j_n}^a) = \min \{ q_2, \min \{ q_1, q_1^a \}, \ldots, q_{j_n}^a \}.
\]

Monotonicity of the constructed labelling function \( E \) follows from the fact that the functions max, min, sum and product are monotonic functions. By applying the above definitions repeatedly from the root \((m, \phi)\), we obtain an abstract dependency graph encoding of the problem \( G, m \models \phi \).
Example 4. Consider again the stochastic game depicted in Figure 1. For any \( k \in \mathbb{N} \) we let \( \phi^k = \langle \langle O, \lozenge \rangle \rangle_{\leq \frac{k}{2}} [a U \leq k] \) and \( \phi^\infty = \langle \langle O, \lozenge \rangle \rangle_{> \frac{K}{2}} [a U \leq k] \). We now encode the model-checking problem \( G, \mathcal{F}_1 \models \phi^k \) into an abstract dependency graph \( G = (V, E, \mathcal{D}, \mathcal{E}) \). A part of the resulting graph is visualised in Figure 2a. Edges connecting the vertices correspond to the specific monotone functions given by our encoding. The greyed out shapes are not vertices but part of the monotonic function for a symbolic node, responsible for computing a weighted sum of successor values, \( q^x_y \), as prescribed by the encoding. We let \( E(v_i) = \varepsilon \) for \( 5 \leq i \leq 10 \), \( E(v_i) = 0 \) for \( 8 \leq i \leq 10 \) and \( E(v_i) = 1 \) for \( 5 \leq i \leq 7 \). This is visualised by vertices having either no outgoing edge or an edge pointing to the empty set. In general, separate unlabelled edges encode a maximum, while a minimum is computed over each unlabelled edge. For vertex \( v_2 \), the edge function is given by \( E(v_2) = v_3 v_4 v_5 v_8 v_{11} v_{12} \) and the function computed at \( v_2 \) is thus

\[
E(v_2)(q_3, q_4, q_5, q_8, q_{11}, q_{12}) = \max \left\{ q^x_y, \min\{q_3, q^x_y\}, \min\{q_5, q^x_y\}, \min\{q_8, q^x_y\}, \min\{q_{11}, q^x_y\}, \min\{q_{12}, q^x_y\} \right\}
\]

where \( q^x_y = \frac{1}{2} \cdot q_1 + \frac{1}{4} \cdot q_3 \) and \( q^x_y = \frac{1}{3} \cdot q_4 + \frac{2}{3} \cdot q_{12} \). The dashed edge encodes the function

\[
E(v_1)(q_2) = \begin{cases} 1 & \text{if } q_2 > \frac{1}{2} \\ 0 & \text{if } q_2 = 0 \\ 0 & \text{otherwise} \end{cases}
\]

Theorem 2 (Correctness). Let \( G = (\Pi, M, \{M_i\}_{i \in \Pi}, \rightarrow, \ell) \) be a game, \( m \in M \) a state and \( \phi \in \mathcal{C}_{\text{ATL}} \) a property. For the abstract dependency graph rooted by \( (m, \phi) \), it holds that \( G, m \models \phi \) iff \( A_{\min}((m, \phi)) = 1 \).

As our domain \( \mathcal{D} \) does not satisfy the ascending chain condition, we cannot reuse the termination argument from [7]. We instead prove the termination by relying on our assumption that all loops are of strictly positive magnitude.

Theorem 3 (Termination). There exists \( k \in \mathbb{N} \) such that \( F^j(A_{\Lambda}) = A_{\min} \) for all \( j \geq k \).

Example 5. Consider the abstract dependency graph in Figure 2a. For vertices \( v_{11}, \ldots, v_{14} \), the minimal fixed point assignment is given by \( A_{\min}(v_{11}) = A_{\min}(v_{12}) = \frac{1}{10} \) and \( A_{\min}(v_{13}) = A_{\min}(v_{14}) = 1 \). Assuming that these assignments have been pre-computed, we now repeatedly apply the fixed point operator to compute the minimal fixed point assignment to the remaining vertices. Hence, we start from an assignment \( A' \) such that \( A'(v_i) = A_{\min}(v_i) \) for \( 11 \leq i \leq 14 \) and \( A'(v_{14}) = A_{\Lambda}(v_{14}) \) otherwise. The result can be seen in Figure 2b. After 3 iterations, the fixed point has been computed with a value of 1 assigned to \( v_1 \), hence by Theorem 2, we can conclude \( G, m_1 \models \langle \langle O, \lozenge \rangle \rangle_{> \frac{5}{2}} [a U \leq s] \).

5 Implementation and Experimental Evaluation

We evaluate our implementation on three different PRISM-games case studies: task-graph-scheduling [26, 27], robot coordination [28] and collective decision making for sensor networks [15]. A package to reproduce our results can be found...
The experiments show that for formulae that query the exact or slightly above probability, our on-the-fly approach achieves verification times comparable or better than those of PRISM-games. Our approach takes slightly more time to derive that a formula does not hold which is expected for an on-the-fly method. Our running times in general improve as we allow for more slack in the $\lambda$ bound. The robot experiment achieves on average about twice as fast verification for the 'below10' and 'below20' queries. In the sensor experiment, the certain-zero approach in combination with on-the-fly verification achieves for the 'below20' on average seven times faster verification, sometimes showing an order of magnitude improvement. Regarding the memory consumption, our method uses on average 3.4 times less memory on the robot experiment, 11.0 times less memory on the sensor experiment and 1.5 times less memory on task graphs.

### 6 Conclusion

We presented an on-the-fly technique for answering whether a turn-based stochastic multiplayer game with weighted transitions satisfies a given alternating-time temporal logic formula with upper-bounds on the accumulated weight in the temporal operators and lower-bounds on the probabilities that a certain path formula is satisfied. Our approach reduces the problem to the computation of minimum fixed point on a recently introduced notion of abstract dependency graphs, using a novel reduction relying on a special abstract domain that includes the certain-zero optimization. We formally prove the correctness of our reduction and provide an efficient C++ implementation. On a series of experiments, we compare the performance of our approach with PRISM-games and show in several instances the advantage of using on-the-fly algorithm compared to the traditional value-iteration method. Our current implementation does not explicitly output winning strategies, however, this information can be recovered in a straightforward way from the fixed point computed on the constructed ADG.

Fig. 4: R-A-B-C is a 2-robot model with A collaborating robots, cost-bound of B on a grid of size C with queries of the type $\langle r_1, \ldots, r_A \rangle_\lambda$ (crash $U_\leq(B,B,B)$ goail). S-X-Y is a sensor model with 4 sensors with X collaborating sensors with a cost-bound of Y and the query $\langle s_1, \ldots, s_X \rangle_\lambda$ (true $U_\leq(Y,Y)$ decision made). T-Q-R is task graph problem and checks whether all tasks can be completed within at most Q time using R energy by the query $\langle \text{sched} \rangle_\lambda$ (true $U_\leq(Q,R)$ tasks complete).

Our open-source implementation is written in C++ without platform specific code. To obviate the need to create our own parser for PRISM models, we modify the export functionality in PRISM-games to construct an explicit transition system that becomes an input to implementation. Furthermore, as PRISM-games do not directly support verification of multidimensional cost-bounded properties, we cannot rely on built-in reward structures and instead introduce variables to capture the accumulated cost. For each model-checking question, we bound the variables by a precision derived from the property, effectively creating a bounded unfolding of the original model, sufficient for verifying the query in question. As the model is bounded by the query precision, it is sufficient to verify in PRISM-games the corresponding unbounded query to solve the original model-checking problem.

### 5.1 Results

Experimental results are gathered by running our implementation and PRISM-games on a Ubuntu 14.04 cluster with AMD Opteron 6376 processors. Each
experiment has a maximum timeout of two hours and 14GB of virtual memory, however, these limits are not exceeded in any experiment. Figure 1 displays the experimental data. The verified formulae are of the form \( \langle C \rangle_{\forall \lambda} (\psi) \) and specified in the caption of the table—the weight dimension being 3 for the robot experiment and 2 for the remaining two. The column labelled with ‘prism’ shows the time (in seconds) it took PRISM-games to verify a query (as PRISM-games computes the exact solution, the times do not vary for the different variants of the formula). The columns for ‘above’ \( (\lambda = p + 0.000001) \), ‘exact’ \( (\lambda = p) \), ‘below10’ \( (\lambda = p - \frac{p}{10}) \) and ‘below20’ \( (\lambda = p - \frac{p}{20}) \) describe the different instantiations of \( \lambda \) used in the queries, where \( p \) is the exact probability computed by PRISM-games. Hence, it is always the case that a formula is satisfied for ‘exact’, ‘below10’, ‘below20’ and never for ‘above’. The remaining columns, e.g., \( \frac{\text{prism}}{\text{above}} \), show the speedup-ratio. As both tools rely on the explicit engine of PRISM-games for model construction, we report only the time spent on verification, as the model construction time is identical for both tools.

The experiments show that for formulae that query the exact or slightly above probability, our on-the-fly approach achieves verification times comparable or better than those of PRISM-games. Our approach takes slightly more time to derive that a formula does not hold which is expected for an on-the-fly method. Our running times in general improve as we allow for more slack in the bound. The robot experiment achieves on average about twice as fast verification for the ‘below10’ and ‘below20’ queries. In the sensor experiment, the certain-zero approach in combination with on-the-fly verification achieves for the ‘below20’ on average seven times faster verification, sometimes showing an order of magnitude improvement. Regarding the memory consumption, our method uses on average 3.4 times less memory on the robot experiment, 11.0 times less memory on the sensor experiment and 1.5 times less memory on task graphs.

6 Conclusion

We presented an on-the-fly technique for answering whether a turn-based stochastic multiplayer game with weighted transitions satisfies a given alternating-time temporal logic formula with upper-bounds on the accumulated weight in the temporal operators and lower-bounds on the probabilities that a certain path formula is satisfied. Our approach reduces the problem to the computation of minimum fixed point on a recently introduced notion of abstract dependency graphs, using a novel reduction relying on a special abstract domain that includes the certain-zero optimization. We formally prove the correctness of our reduction and provide an efficient C++ implementation. On a series of experiments, we compare the performance of our approach with PRISM-games and show in several instances the advantage of using on-the-fly algorithm compared to the traditional value-iteration method. Our current implementation does not explicitly output winning strategies, however, this information can be recovered in a straightforward way from the fixed point computed on the constructed ADG.
References

Testing Your (Static Analysis) Truths

Ignacio Casso, José F. Morales, P. López-García, and Manuel V. Hermenegildo

Abstract. Static analysis is nowadays an essential component of many software development toolsets, attracting significant research interest and practical application. Unfortunately, the ever-increasing complexity of static analyzers makes their coding error-prone. At the same time, the correctness and reliability of software analyzers is critical if they are to be inserted in production compilers and development environments. While there have been some notorious successes in the validation of compilers, comparatively little work exists on the systematic validation of static analyzers. Contributing factors here may be the intrinsic difficulty of formally verifying code that is quite complex and of finding suitable oracles for testing it. In this paper, we propose a simple, automatic method for testing abstract interpretation-based static analyzers. Broadly, it consists in checking, over a suite of benchmarks, that the properties inferred statically are satisfied dynamically. The main advantage of our approach is its simplicity, which stems directly from framing it within the Ciao assertion-based validation framework, and its blended static/dynamic assertion checking approach. We show that in this setting, the analysis can be tested with little effort by combining the following components already present in the framework: 1) the static analyzer, which outputs its results as the original program source with assertions interspersed; 2) the assertion run-time checking mechanism, which instruments a program to ensure that no assertion is violated at run time; 3) the random test case generator, which generates random test cases satisfying the properties present in assertion preconditions; and 4) the unit-test framework, which executes those test cases. We show how a combination of these elements and a trivial program transformation work together to compose a tool that can effectively discover and locate errors in the different components of the static analyzer. We apply our approach to test some of CiaoPP’s analysis domains over a wide range of programs, successfully finding non-trivial, previously undetected bugs, with a low degree of effort.

Keywords: Static Analysis, Run-time Checks, Random Testing, Assertions, Abstract Interpretation, Program Analysis, (Constraint) Logic Programming.

1 Introduction and Motivation

Static analysis tools are nowadays a crucial component of the development environments for many programming languages. They are widely used in different steps of

Testing Your (Static Analysis) Truths *

Ignacio Casso\textsuperscript{1[0000–0001–9196–7951]}, José F. Morales\textsuperscript{1[0000–0001–6098–3895]}, P. López-García\textsuperscript{1,3[0000–0002–1092–2071]}, and Manuel V. Hermenegildo\textsuperscript{1,2[0000–0002–7583–323X]}

\textsuperscript{1} IMDEA Software Institute, Madrid, Spain
\textsuperscript{2} ETSI Informática, Universidad Politécnica de Madrid (UPM), Madrid, Spain
\textsuperscript{3} Spanish Council for Scientific Research (CSIC), Spain
\{ignacio.casso, josef.morales, pedro.lopez, manuel.hermenegildo\}@imdea.org

Abstract. Static analysis is nowadays an essential component of many software development toolsets, attracting significant research interest and practical application. Unfortunately, the ever-increasing complexity of static analyzers makes their coding error-prone. At the same time, the correctness and reliability of software analyzers is critical if they are to be inserted in production compilers and development environments. While there have been some notorious successes in the validation of compilers, comparatively little work exists on the systematic validation of static analyzers. Contributing factors here may be the intrinsic difficulty of formally verifying code that is quite complex and of finding suitable oracles for testing it. In this paper, we propose a simple, automatic method for testing abstract interpretation-based static analyzers. Broadly, it consists in checking, over a suite of benchmarks, that the properties inferred statically are satisfied dynamically. The main advantage of our approach is its simplicity, which stems directly from framing it within the Ciao assertion-based validation framework, and its blended static/dynamic assertion checking approach. We show that in this setting, the analysis can be tested with little effort by combining the following components already present in the framework: 1) the static analyzer, which outputs its results as the original program source with assertions interspersed; 2) the assertion run-time checking mechanism, which instruments a program to ensure that no assertion is violated at run time; 3) the random test case generator, which generates random test cases satisfying the properties present in assertion preconditions; and 4) the unit-test framework, which executes those test cases. We show how a combination of these elements and a trivial program transformation work together to compose a tool that can effectively discover and locate errors in the different components of the static analyzer. We apply our approach to test some of CiaoPP’s analysis domains over a wide range of programs, successfully finding non-trivial, previously undetected bugs, with a low degree of effort.

Keywords: Static Analysis, Run-time Checks, Random Testing, Assertions, Abstract Interpretation, Program Analysis, (Constraint) Logic Programming.

1 Introduction and Motivation

Static analysis tools are nowadays a crucial component of the development environments for many programming languages. They are widely used in different steps of

the software development cycle, such as code optimization and verification, and they are the subject of significant research interest and practical application. Unfortunately, modern analyzers are often very large and complex software artifacts, and this makes them prone to bugs. This is a limitation to their applicability in real-life production compilers and development environments, where they are typically used in critical tasks like verification or code optimization, that need to rely strongly on the soundness of the analysis results.

However, the validation of static analyzers is a challenging problem, which is not well covered in the literature or by existing tools. Well-established methodologies or even guidelines to this end do not really exist. This is due to the fact that direct application of formal methods is not always straightforward with code that is so complex and large, even without considering the problem of having precise specifications to check against—a clear instance of the classic problem of who checks the checker. In current practice, extensive testing is the most extended and realistic option, but it poses some significant challenges too. Testing separate components of the analyzer misses integration testing, and designing proper oracles for testing the complete tool is really challenging.

Our objective in this paper is to develop a simple, automatic method for testing abstract interpretation-based static analyzers. Although the approach is general, we develop it for concreteness in the context of the Ciao [20] logic programming-based, multiparadigm language. The Ciao programming environment includes an abstract interpretation-based static analyzer, CiaoPP, which faces this very problem. As other “classic” analyzers, this analyzer has evolved for a long time, incorporating a large number of abstract domains, features, and techniques, adding up to over 1/2 million lines of Ciao code. These components have in turn reached over the years different levels of maturity. While the essential parts, such as the fixpoint algorithms and the classic abstract domains, have been used routinely for a long time now and it is unusual to find bugs, other parts are less developed and yet others are prototypes or even proofs of concept. A recent, shallow effort of applying a new testing tool to some parts of the Ciao analyzers as a case study [10] revealed subtle bugs, not only in the less-developed parts of the system, but also in corner cases of the parts that are considered more mature, such as, e.g., in the handling of rarely-used built-ins.

Another feature of Ciao that will be instrumental to our approach is the use of a unified assertion language and framework across its different components [21, 22], which together implement its unique blend of static and dynamic assertion checking. These components include: 1) the PLAI static analyzer [38, 24, 18], which expresses the inferred information as Ciao assertions interspersed within the original program; 2) the assertion runtime-checking framework [43, 44], which instruments the code to ensure that any assertions remaining after static verification are not violated at run time; 3) the (random) test case generation framework [10], which generates random test cases satisfying the properties present in an assertion preconditions; 4) the unit-test framework [35], which executes those test cases.

In this paper, we propose an algorithm that combines these four basic components in a novel way that allows testing the static analyzer almost for free. Intuitively, it consists in checking, over a suite of benchmarks, that the properties inferred statically are satisfied dynamically. The overall testing process, for each benchmark, can be summarized as follows: first the code is analyzed and the analysis results are expressed by the analyzer as assertions interspersed within the original code. Then these assertions are switched into run-time checks, that will ensure that violations of those assertions are reported at run time. Finally, random test cases are generated and executed to

2
exercise those run-time checks. If any assertion violation is reported, since these assertions (the analyzer output) must cover all possible concrete executions, it means that the assertion was incorrectly inferred by the analyzer and thus that an error in the analyzer has been found. This process can be easily automated, and if it is repeated for an extensive and varied enough suite of benchmarks, it can be used to effectively validate (even if not fully verify) the analyzer or to discover new bugs. Furthermore, the implementation, when framed within the Ciao assertion-based validation framework, is very simple, since, as we will show, only a basic code transformation and a simple driver need to be implemented to obtain a very useful, working system.

The idea of checking at run time the properties or assertions inferred by the analysis for different program points, is not new. For example, successfully applied this technique for checking a range of different aliasing analyses. However, these approaches require the development of tailored instrumentation or monitoring, and require significant effort in their design and implementation. We argue that the testing approach is made more applicable, general, and scalable by the use of a unified assertion-based framework for static analysis and dynamic debugging, as the one of Ciao. As mentioned before, framing things in such a framework, the approach can be implemented with the already existing components in the system, in a very simple way, so much so that our initial prototype was, in fact, barely 50 lines of code long. We argue also that our approach is particularly useful in a mixed production and research setting like that of CiaoPP, in which there is a mature and domain-parametric abstract interpretation framework used routinely, but new, experimental abstract domains and overall improvements are in constant development. Those domains can easily be tested relying only on the existing abstract-interpretation framework, runtime-checking framework, and unified assertion language, provided only that the assertion language is extended to include the properties relevant for the domains.

The rest of the paper is structured as follows. Section 2 gives background knowledge needed to describe the main ideas and contributions of this paper. In particular, we recall some relevant aspects of the CiaoPP unified assertion framework. Then, Section 3 gives an overview of our approach illustrating it with an example. Section 4 presents our concrete algorithm to combine the different elements of the framework for the task of testing the static analyzer. In Section 5 we show some examples and applications of our approach. In Section 6 we apply the idea to testing the analysis results for a wide range of CiaoPP’s abstract domains and properties. Finally, Section 7 discusses related work and Section 8 summarizes our conclusions and plans for future work.

2 Preliminaries

In this section we review in some more detail those aspects of the Ciao model that are relevant to our approach, including the assertion language and the blended static and dynamic assertion checking framework built around it. A more detailed presentation can be found in [4, 21, 40, 23, 35, 20] and their references.

The Assertion Language. Ciao assertions are linguistic constructs, which allow expressing properties of programs. There are two types of assertions in Ciao that are relevant herein: predicate assertions and program-point assertions. The first ones are declarations that provide partial specifications of a predicate. They have the following syntax: :- [Status] pred Head : [Calls] => [Success] + [Comp], indicating that if a call to the goal Head satisfies precondition Calls, it must satisfy post-condition Success on success and global computational properties Comp. Program-point assertions are reserved literals that appear in clause bodies and describe the constraint store at the
corresponding program point. Their syntax is \([\text{Status}]\text{(State)}\). Examples of both types of assertions are provided in the code fragment below:

```
1  :- check pred append(X,Y,Z) : (\text{list}(X) \text{.} \text{list}(Y)) \Rightarrow \text{list}(Z) + \text{is\_det}.
2  :- check pred append(X,Y,Z) : (\text{var}(X) \text{.} \text{var}(Y) \text{.} \text{list}(Z)) \Rightarrow (\text{list}(X) \text{.} \text{list}(Y)) + \text{non\_det}.
3  append([],X,X).
4  append([X|Xs],Ys,[X|Zs]) :-
5      append(Xs,Ys,Zs),
6      check((\text{list}(Xs) \text{.} \text{list}(Ys) \text{.} \text{list}(Zs))).
```

Assertion fields \textit{Calls}, \textit{Success}, \textit{Comp} and \textit{State}, are conjunctions of \textit{properties}. Such properties are predicates, typically written in the source language (user-defined or in libraries), and thus runnable, so that they can be used as run-time checks, and which, for our purposes, are typically \textit{native} to \textit{CiaoPP}, i.e., abstracted and inferred by some domain in \textit{CiaoPP}. This includes a wide range of properties, from types, modes and variable sharing, to determinism, (non)failure and resource consumption. We refer the reader to \cite{39, 23, 20} and their references for a full description of the \textit{Ciao} assertion language.

Assertions are used everywhere in \textit{Ciao}, from documentation and foreign interface definitions to static analysis and dynamic debugging. Depending on their origin and intended use they have a different status, the \textit{Status} field in the syntax described above. Assertion statuses relevant herein include \textit{true}, which is used for assertions that are output from the analysis (and thus must be safe approximations), or the default status \textit{check}, which indicates that the validity of the assertion is unknown and it must be checked, statically or dynamically. We will return to this crucial issue below.

Fig. \ref{fig:1} depicts the overall architecture of the \textit{Ciao} unified assertion framework. Hexagons represent tools, and arrows indicate the communication paths among them. The input to the process is the user program, optionally including a set of assertions; this set always includes any assertion present for predicates exported by any libraries used (left part of Fig. \ref{fig:1}).

\textit{Static Analysis}. One use of \textit{Ciao} assertions is as an interface to the static analyzer. As mentioned above, assertions can be used to indicate what we want the analyzer to
check (the default `check` status), or to guide the analysis by feeding it information that it might be unable to infer by itself (`trust` status). The latter includes as a special case providing information on the entry points to the module being analyzed (i.e., on the calls to the predicates exported by the module – `entry` status). But more importantly for this paper, assertions are one of the possible output formats in which the analysis results are produced by the static analyzer (assertions with `true` status). If this type of output is chosen, a new source file for the analyzed program will be created, exactly as the original but with `true program-point` assertions interspersed between every two consecutive literals of each clause, and with one or more `true predicate` assertions for each predicate.

The technical and theoretical details of how this is achieved are omitted for space constraints. For our purposes it is sufficient to say that the CiaoPP analyzer is abstract interpretation-based, and its design consists of a common abstract-interpretation framework (the fixpoint algorithm(s)) parameterized by different, “pluggable” abstract domains. Depending on the domain or combination of domains selected for the analysis, different properties will be inferred and will appear in the emitted `true` assertions.

**Run-time Checking.** Static analysis can be used for compile-time checking of assertions (the Static Comparator & Simplifier, in Fig. [1]) but the inherent imprecision of the analysis can lead to some assertions, specially those with user-defined properties that are not native to abstract domains, to not be proved or disproved statically (although perhaps they are simplified). In those cases, the remaining unproved (parts of) assertions are written into the output program with `check` status and then this output program can optionally be instrumented with run-time checks. These dynamic checks will encode the meaning of the `check` assertions, ensuring that an error is reported at run-time if any of these remaining assertions is violated (the dynamic part of the model). Note that the fact that properties are written in the source language and runnably is essential in this process, and allows checking new user-defined and native properties without having to extend the run-time checking framework. This results in a very rich set of properties being checkable in Ciao, including types, modes, variable sharing, failure, exceptions, determinism, choice-points, resources, and more, blending smoothly static and dynamic techniques.

**Unit Tests, Test Case Generation, and Assertion-based Testing.** Test inputs can be provided by the user, by means of `test` assertions (unit tests), and used to test the test assertion itself as well as, through the `runtime-checking` mechanism, any other assertion in any predicate called by the test case, that was not eliminated in the static checking. The unit-testing framework in principle requires the user to manually write individual test cases for each assertion to be tested. However, the Ciao model also includes mechanisms for generating test cases automatically from the assertion preconditions, using the corresponding property predicates as generators. This has been extended recently [10] to a full random test case generation framework, which automatically generates, using the same technique, random test cases that satisfy assertion preconditions. We refer to the combination of this test generation mechanism with the run-time checking of the intervening assertions as `assertion-based testing`, that is, generating and running relevant test cases which exercise the `run-time checks` of the assertions in a program, thus testing if those assertions are correct. This yields similar results to `property-based testing` [11] but in a more integrated way within the overall model. Such automatic generation is supported for native properties, `regular types`, and user-defined properties as long as they are restricted to pure Prolog with arithmetic
or mode and sharing constraints. In particular, it is always supported for the native properties used by the different analyses in the assertions that they output.

3 Overview of the Approach

After introducing the relevant elements of the Ciao assertion model, we can now sketch the main idea of our approach with a motivating example. Assume we have this simple Prolog program, where the entry assertion indicates that the predicate is always called with its second argument instantiated to a list and the third a free variable:

```
:- entry prepend(X,Xs,Ys) : (list(Xs), var(Ys)).
```

Assume that we analyze it with a simple modes abstract domain that assigns to each variable in an abstract substitution one of the following abstract values: $g$ (variable is ground), $v$ (variable is free), $ng$ (variable is not ground), $nv$ (variables is not free), $ngv$ (variable is not ground nor free), or any (nothing can be said about the variable). Assume also that the analysis is incorrect because it does not consider sharing (aliasing) between variables, so when updating the abstract substitution after the Rest=Xs literal, the abstract value for Ys is not modified at all. The result of the analysis will be represented, as explained in the previous section, as a new source file with interspersed assertions, as shown in Fig. 2. Note that the correct result, if the analysis considered aliasing, would be that there is no groundness information for Ys at the end of the clause, since there is none for X and Xs at the beginning either. Ys could only be inferred to be nonvar, but instead is incorrectly inferred to be nonground too. Note also that unknown/1 properties would not actually appear in the analysis output, but are included for clarity.

What we would like at this point, is to be able to check dynamically the validity of the true assertions from the analyzer. Thanks to the different aspects of the Ciao model presented previously, the only thing needed in order to achieve this is to (1) turn the status of the true assertions produced by the analyzer into check, as shown in Fig. 3. This would normally not make any sense since these true assertions have been proved by the analyzer. But that is exactly what we want to check, i.e., whether the information inferred is incorrect. To do this, (2) we run the transformed program (Fig. 3) again through CiaoPP (Fig. 1) but without performing any analysis. In that case the check literals (stemming from the true literals of the previous run) will not be simplified in the comparator (since there is no abstract information to compare

---

```
:- entry prepend(X,Xs,Ys) : (list(Xs), var(Ys)).
```

```
:- true pred prepend(X,Xs,Ys)
  : (unknown(X), nonvar(Xs), var(Ys))
  => (unknown(X), nonvar(Xs), nonground(Ys), nonvar(Ys)).
```

```
prepend(X,Xs,Ys) :-
  true(((unknown(X), nonvar(Xs), var(Ys), var(Rest)))),
  Ys=[X|Rest],
  true(((unknown(X), nonvar(Xs), nonground(Ys), nonvar(Ys), var(Rest)))),
  Rest=Xs.
true(((unknown(X), nonvar(Xs), nonground(Ys), nonvar(Ys), nonvar(Rest)))).
```

---

**Fig. 2.** An incorrect simple mode analysis.
An abstract interpretation-based static analysis computes an over-approximation \( S^+ P \) of the collecting semantics \( S P \) of a program \( P \). Such collecting semantics can be broadly defined as a control flow graph for the program decorated at each node with the set of all possible states that could occur at run-time at that program point. Different approximations of this semantics will have smaller or larger sets of possible states at each program point. Let us denote by \( S' P \subset S'' P \) the relation that establishes that an approximation of \( S P \), \( S'' P \), is an over-approximation of another, \( S' P \). The analysis will be correct if indeed \( S P \subset S^+ P \).

Since \( S P \) is undecidable, this relation cannot be checked in general. However, if we had a good enough under-approximation \( S^- P \) of \( S P \), it can be tested as \( S^- P \subset S^+ P \). If it does not hold and \( S^- P \not\subset S^+ P \), then it would imply that \( S P \not\subset S^+ P \), and thus, the results of the analysis would be incorrect, i.e., the computed \( S^+ P \) would not actually be an over-approximation of \( S P \).

An under-approximation of the collecting semantics of \( P \) is easy to compute: it suffices with running the program with a subset \( I^- \) of the set \( I \) of all possible initial states. We denote the resulting under-approximation \( S^- I P \), and note that \( S P = S^- I P \), which would be computable if \( I \) is finite and \( P \) always terminates. That is the method that we propose for testing the analysis: selecting a large and varied enough \( I^- \), computing \( S^- I P \) and checking that \( S^- I P \subset S^+ P \).

A direct implementation of this idea is challenging. It would require tailored instrumentation and monitoring to build and deal with a partially constructed collecting semantic under-approximation as a programming structure, which then would need to be compared to the one the analysis handles. However, as we have seen the process can be greatly simplified by reusing some of the components already in the system, following these observations:

- We can work with one initial state \( i \) at a time, following this reasoning:
  \[ S^- I P \subset S^+ P \iff \forall i \in I^-, S\{i\} P \subset S^+ P. \]
- We can use the random test case generation framework for selecting each initial state \( i \).
- Instead of checking \( S\{i\} P \subset S^+ P \), we can instrument the code with run-time checks to ensure the execution from initial state \( i \) does not contradict the analysis at any point. That is, that the state of the program at any program point is contained in the over-approximation of the set of possible states that the analysis inferred and output as Ciao assertions.

4.2 The Algorithm

We now show the concrete algorithm for implementing our proposal, i.e., the driver that combines and inter-operates the different components of the framework to achieve the desired results. The essence of the algorithm (Alg. 1) is the following: non-deterministically choose a program \( P \) and a domain \( D \) from a collection of benchmarks and domains, and execute the \texttt{AnaTest}(\( P, D \)) procedure until an error is found or a limit is reached. Unless the testing part is ensured to explore the complete execution space, it could in principle be useful to revisit the same \((P, D)\) pair more than once. When the algorithm detects a faulty program-point assertion for some input \( \text{Error}(\text{input}) \), it means that the concrete execution reaches a state not captured by the analysis at any point, and thus the analysis is incorrect.

The same procedure can be followed to debug different analyses with different benchmarks. If the execution of any test case reports a runtime-checking error for one assertion, it will mean that the assertion was not correct and the analyzer computed an incorrect over-approximation of the semantics of the program. Alternatively, if this experiment, which can be automated easily, is run for an extensive suite of benchmarks without errors, we can gain more confidence that our analysis implementation is correct, even if perhaps imprecise (although of course we cannot have actual correctness in general by testing).

4 The Algorithm

In this section we present in more detail the actual algorithm for combining the components of the framework used in order to test the static analyzer.

4.1 Basic Reasoning Behind the Approach

We start by establishing more concretely the basic reasoning behind the approach in terms of abstract interpretation and safe upper and lower approximations. The mathematical notation in this subsection is purely for readability, as a proper formalization

\[\text{Fig. 3. The instrumented program.}\]

1: `:- entry prepend(X,Xs,Ys) : (list(Xs), var(Ys)).`
2: `:- check pred prepend(X,Xs,Ys)
3:    : (unknown(X), nonvar(Xs), var(Ys))
4:    => (unknown(X), nonvar(Xs), nonground(Ys), nonvar(Ys)).`
5: `prepend(X,Xs,Ys) :-
6:   check((nonvar(Xs), var(Ys), var(Rest))),
7:   Ys=X[Rest],
8:   check((nonvar(Xs), nonground(Ys), nonvar(Ys), var(Rest))),
9:   Rest=Xs,
10:  check((nonvar(Xs), nonground(Ys), nonvar(Ys), nonvar(Rest))).`

\[\text{\footnote{In the discussion above we have assumed for simplicity that the original program did not already contain check assertions. In that case these need to be treated separately and there are several options, including simply ignoring them for the process or actually turning them into trusts, so that we switch roles and trust the user-provided properties while checking the analyzer-inferred ones. This very interesting issue of when and whether to use the user-provided assertions to be checked during analysis, and its relation to run-time checking is discussed in depth in \cite{17}.}}\]
Algorithm 1

Analysis Testing Algorithm (for program $P$ and domain $D$)

1: procedure AnaTest($P$, $D$)
2: result ← None
3: $P_{an}$ ← analyze and annotate $P$ with domain $D$ (incl. program-point assertions).
4: $P_{check}$ ← $P_{an}$ where true assertion status is replaced by check
5: $P_{rtcheck}$ ← instrument $P_{check}$ with run-time checks
6: repeat
7: Choose an exported predicate $p$ and generate a test case input
8: if $p$ (input) in $P_{check}$ produces runtime errors then
9: result ← Error (input)
10: else if maximum number of test executions is reached then
11: result ← Timeout
12: until result $\neq$ None
13: return result

the (over-approximation of the) analysis. In such case it is possible to reconstruct (or store together with the test output) additional information to diagnose the problem. E.g., comparing the concrete execution trace (which is logged during testing) with the analysis graph (recoverable from $P_{an}$, the program annotated with analysis results), domain operations (inspecting the analysis graph), and transfer functions (from predicates that are native to each domain).

4.3 Other Details and Observations

We now discuss some details and observations on the algorithm that may have been left out or oversimplified in the algorithm sketch:

Analysis Crashes. An implicit assumption throughout our discussion so far is that the analysis always terminates without errors, but the results computed may be unsound. Of course, it is also possible that a bug in the analysis produces a crash, or even leads to non-termination. It is also possible that the analysis output is malformed (e.g., there are missing assertions in $P_{an}$). Those errors are of course also checked and reported by our tool. Non-termination is handled with timeouts and possible warnings (both for analyses and concrete executions).

Benchmark Selection. No prior requirement is imposed on the origin or characteristics of the benchmark suite. It could consist of automatically generated programs, an existing benchmark suite, or just real-life code. Each may have its own advantages and disadvantages (e.g., automatically generated code may test more convoluted or corner cases, but real-life code may find the bugs that actually occur in programs), but in principle, our approach is agnostic in this regard.

Entry Points. There is no restriction regarding the number of entry points or inputs to a program to be analyzed for. It is common in tools related to ours to use as benchmarks programs with a single entry point with no inputs (e.g., just a single \texttt{void main()} function as entry point for C). Our benchmarks are typically Ciao modules, and their entry points to analysis and testing are their exported predicates. In Ciao programs signatures and types (as well as entry assertions) are optional. Admissible inputs (i.e., the initial set of possible states for analysis or test case generation) can be specified by writing assertions for the exported predicates, by means of entry assertions, or skipped.

- We can work with one initial state $i$ at a time, following this reasoning:
  $S_{i}^{\bot} \subset_{P} S_{p}^{\dagger} \iff \forall i \in I^{-}, S_{i}^{\bot} \subset_{P} S_{p}^{\dagger}.$
- We can use the random test case generation framework for selecting each initial state $i$.
- Instead of checking $S_{i}^{\bot} \subset_{P} S_{p}^{\dagger}$, we can instrument the code with run-time checks to ensure the execution from initial state $i$ does not contradict the analysis at any point. That is, that the state of the program at any program point is contained in the over-approximation of the set of possible states that the analysis inferred and output as Ciao assertions.

4.2 The Algorithm

We now show the concrete algorithm for implementing our proposal, i.e., the driver that combines and inter-operates the different components of the framework to achieve the desired results. The essence of the algorithm (Alg. 1) is the following: non-deterministically choose a program $P$ and a domain $D$ from a collection of benchmarks and domains, and execute the \texttt{ANA\textsc{Test}}($P$, $D$) procedure until an error is found or a limit is reached. Unless the testing part is ensured to explore the complete execution space, it could in principle be useful to revisit the same ($P$, $D$) pair more than once. When the algorithm detects a faulty program-point assertion for some input (\texttt{ERROR(input)}), it means that the concrete execution reaches a state not captured by
the (over-approximation of the) analysis. In such case it is possible to reconstruct (or store together with the test output) additional information to diagnose the problem. E.g., comparing the concrete execution trace (which is logged during testing) with the analysis graph (recoverable from \( P_{an} \), the program annotated with analysis results), domain operations (inspecting the analysis graph), and transfer functions (from predicates that are native to each domain).

### 4.3 Other Details and Observations

We now discuss some details and observations on the algorithm that may have been left out or oversimplified in the algorithm sketch:

**Analysis Crashes.** An implicit assumption throughout our discussion so far is that the analysis always terminates without errors, but the results computed may be unsound. Of course, it is also possible that a bug in the analysis produces a crash, or even leads to non-termination. It is also possible that the analysis output is malformed (e.g., there are missing assertions in \( P_{an} \)). Those errors are of course also checked and reported by our tool. Non-termination is handled with timeouts and possible warnings (both for analyses and concrete executions).

**Benchmark Selection.** No prior requirement is imposed on the origin or characteristics of the benchmark suite. It could consist of automatically generated programs, an existing benchmark suite, or just real-life code. Each may have its own advantages and disadvantages (e.g., automatically generated code may test more convoluted or corner cases, but real-life code may find the bugs that actually occur in programs), but in principle, our approach is agnostic in this regard.

**Entry Points.** There is no restriction regarding the number of entry points or inputs to a program to be analyzed for. It is common in tools related to ours to use as benchmarks programs with a single entry point with no inputs (e.g., just a single \texttt{void main()} function as entry point for C). Our benchmarks are typically Ciao modules, and their entry points to analysis and testing are their exported predicates. In Ciao programs signatures and types (as well as entry assertions) are optional. Admissible inputs (i.e., the initial set of possible states for analysis or test case generation) can be specified by writing assertions for the exported predicates, by means of entry assertions, or skipped.

---

**Algorithm 1** Analysis Testing Algorithm (for program \( P \) and domain \( D \))

```latex
1: \textbf{procedure} \text{AnaTest}(P, D) \\
2: \hspace{1em} \text{result} \leftarrow \text{NONE} \\
3: \hspace{1em} P_{an} \leftarrow \text{analyze and annotate } P \text{ with domain } D \text{ (incl. program-point assertions).} \\
4: \hspace{1em} P_{check} \leftarrow P_{an} \text{ where true assertion status is replaced by } \text{check} \\
5: \hspace{1em} P_{check} \leftarrow \text{instrument } P_{check} \text{ with run-time checks} \\
6: \hspace{1em} \text{repeat} \\
7: \hspace{2em} \text{Choose an exported predicate } p \text{ and generate a test case } \text{input} \\
8: \hspace{2em} \text{if } p(\text{input}) \text{ in } P_{check} \text{ produces runtime errors then} \\
9: \hspace{3em} \text{result} \leftarrow \text{ERROR(\text{input})} \\
10: \hspace{2em} \text{else if maximum number of test executions is reached then} \\
11: \hspace{3em} \text{result} \leftarrow \text{TIMEOUT} \\
12: \hspace{1em} \text{until result } \neq \text{NONE} \hspace{1em} \text{return result}
```
al together. Note also that if our benchmarks had the restriction mentioned above (in our case, exporting only a main/0 predicate), then test case generation would not be needed for our algorithm.

Test Case Generation. In the absence of entry assertions, the test case generation framework has already some mechanisms to generate relevant test cases, instead of random, nonsensical inputs which would exercise few run-time checks before failing. However, these generators have limitations, and the assertion-based testing framework is in fact best used with assertions that have descriptive-enough call patterns, or with custom user-defined generators in their absence. To tackle this problem, our tool makes also use of test assertions when available in the benchmarks, using also the test cases specified in the benchmarks besides those randomly generated. This can help, e.g., when using a benchmark that works with files and has paths as input, for which relevant test cases would not likely be found with random generation. Note however that the tool would still work without any entry or test assertions; it would just become less effective.

Error Diagnosis and Debugging. It is important to note that although error diagnosis and debugging is primarily left for the user to manually perform, our tool facilitates the task in some aspects. Firstly, the assertion-based testing tool supports shrinking of failed test cases, so we can expect reasonably small variable substitutions in the errors reported. Note however that benchmark reduction, e.g., by delta debugging [49], is currently not supported. Secondly, as sketched in Algorithm [1] the error location and trace reported by the runtime-checks instrumentation provide an approximated idea of the point where the analysis went wrong, if not of the reason why. For example, if the runtime-check error points to a program-point assertion right after a call to a builtin, then we typically know that the analysis erred in the builtin handler.

Multivarience and Path-Sensitivity. As presented, our approach might miss some analysis errors even when the right test cases are used, since we have apparently disregarded multi-variance and path-sensitivity. In fact in CiaoPP the information inferred is fully multi-variant, and separate path information is kept to each variant. However, in order to produce an output that is easy for the programmer to inspect, i.e., that is close to the source program, when outputting the analysis results CiaoPP by default combines the different versions of each predicate (and the associated information) into a single code version and a single combined assertion for each program point and predicate. If this default output is used when implementing our approach, it is indeed entirely possible that the analysis errs at a program point in one path but the algorithm never detects it: this can happen if, for example, in another path leading to the same program point (such that the two paths an their corresponding analysis results are collapsed -lubbed- together at the same program point) the analysis infers a too general value (higher in the domain lattice) at that program point and thus, the error is not detected. However, this potential problem is easily addressed by simply changing the corresponding flag in CiaoPP so that the different versions are not collapsed and are instead materialized into different predicate instances. This is done in CiaoPP by selecting the versions transformation prior to emitting the output. In this case multiple versions may be generated for a given predicate, if there are separate paths to it with different abstract information, and the corresponding analysis information will be annotated separately for each abstract path through the program in the program text of the different versions, avoiding the problem mentioned above.
5 Applications and Examples

In this section we discuss interesting use cases and applications of our approach. As observed before, our testing technique can be seen as a sanity or coherence check, and thus it can be targeted to test different components of the system depending on which ones are assumed to be trusted. Some examples follow. A few of them have actually been implemented and we report on them in the following section. We hope to implement the others for the future versions of the paper.

Debugging Abstract Domains. The first application of our approach, which has been illustrated in the examples, is to test the abstract domains. In general the Ciao abstract interpretation engine (the fixpoint algorithms and all the surrounding infrastructure of the system, into which the domains are “plugged-in”) includes the components of the analyzer we trust most, since they have been used and refined for more than 30 years. Thus, it makes sense to take this as the trusted base and try to find errors in the domains. This situation is realistic and frequent, since CiaoPP is at the same time a production and a research tool, and new domains are constantly being developed. In order to test a new domain with the algorithm proposed, two components need to be present. The first one is a translation interface from the abstract values in the domain to Ciao properties, which is needed to express the analysis results as assertions. But note that this is actually already a requirement for any abstract domain that intends to make full use of the framework, so it is normally implemented anyway in all domains. The other component is to have builtin checks for those properties to be used by the run-time checking framework, if those properties are declared native and not written in the source language and thus already runnable and checkable. This is also a standard requirement on domains to be able to make full use of the framework, so they are typically also implemented with the domain. In particular, all current Ciao abstract domains include the functionalities mentioned, and can be tested as is with the proposed approach. We show the results for some of them in the case study described in Sec. 6.

Debugging Trust Assertions and Custom Transfer Functions. One feature of CiaoPP’s analyses is that they can be guided by the user, which can feed the analyzer with information that can be assumed to be true at points where otherwise the analysis would lose precision. We have already introduced in Sec. 2 one of these mechanisms, trust assertions, but there are others. One is custom abstract transfer functions, similar to those that need to be implemented for abstracting each builtin within each domain, but that the user can provide for any predicate. A particular instance of this mechanism is when the user specifies that one predicate is indistinguishable from or should behave like another with respect to a domain: the equiv declaration. Our approach can be used to test these mechanisms too. Both to test that they are applied correctly by the analyzer, if the user-provided information is trusted to be correct, and to test that the user-provided information is correct, if what is trusted is that the information is applied correctly. The latter is in particular very useful, since even a completely sound analyzer can produce unsound results if it assumes some property to be true when it is actually not, and thus there will always be the need to test such properties.

Testing the Abstract Interpretation Engine. Another idea that comes to mind is whether we can test the abstract interpretation engine (the fixpoint algorithms and all the surrounding infrastructure of the framework) instead of the domains, by using
domains that are simple enough to be used as a trusted base. While the classic algorithms are quite stable, new fixpoints are also added to the system (e.g., recently a modular and incremental fixpoint) which can of course bring new bugs. A first abstract domain that could be useful for this purpose is the concrete domain itself (which is actually implemented in CiaoPP as the pd -partial deduction- domain). If we give the analysis a singleton set of initial states as entry point, the analyzer should behave as an interpreter for the program starting from that initial state, provided the program terminates. The assertions resulting from this “analysis” will use the $$=2$$ property and be essentially a program which is adorned at each program point with the concrete states(s) that the analyzer infers will be occurring at run time, expressed as conjunctions of substitutions using $$=2$$. Then, when running this program, the run-time checks would check that the variables are indeed instantiated to the concrete values inferred. Non-deterministic programs could be equally handled with member/2 ($\in$) instead of $$=2$$ ($= $$). A second domain that could be useful in this context is the pdh domain, which can be used to perform reachability analysis. The properties appearing in the assertions resulting from this analysis would just be possibly_reachable/0 (T) and not_reachable/0 (⊥), which indicates if a program point is definitely unreachable at run-time.

The run-time checks would just report an error any time a check for the property not_reachable/0 (⊥) is invoked at run time. This test would then detect if the analyzer incorrectly marks reachable parts of the program as unreachable.

Testing the Overall Consistency of the Framework. So far we have focused on applications in testing analysis soundness. But doing so has the implicit assumption that there are clear semantics and specifications for the analyzer to follow, and that is not always the case. Sometimes the semantics is underspecified, and then a discrepancy between what the analysis infers and what the program executes is not so much an error but a disparity in the interpretation of such and under-specification. In those cases our tool helps ensure that at least the analysis and run-time semantics are consistent. A relevant example can be found in the case of the abstraction of built-ins within abstract domain implementations. For some of them the specification is not complete (sometimes even the ISO-Prolog standard) and again our tool can at least check for inconsistencies in the interpretations made by the analyses and the run-time system.

In this same line, the tool has helped us find inconsistencies between the understanding of Ciao properties in the analysis and in the runtime-checks framework. With many properties this cannot happen (e.g., with pure predicates) because both the analysis and the run-time checking derive the semantics from the actual code defining the property. But for more complex properties the implementations may be different, perhaps developed by different people, with different interpretations of the property semantics. An actual example is the property cardinality/3, which provides upper and lower bounds to the number of solutions that a predicate might produce. It is a property that has not seen a lot of use (determinacy and/or non-failure are the ones used most frequently), and our experimental evaluation exposed that for cardinality/3 the analysis was considering only different solutions while the runtime-checks framework counted also repeated ones.

Integration Testing of the Analyzer and Third Parties. Finally, even if every piece of the analyzer is validated separately, our tool can still help in testing how all its

\[5\] Note that this, combined with non-failure analysis [15, 5], can also infer definitely_reachable/0, but that is a more complex domain.
While we are planning on performing a larger set of experiments, the results so far are promising and have allowed us to draw some interesting conclusions and observations. A good number of bugs and inconsistencies were indeed found using the technique, many of them known but also some new ones. First, our experiment was successful in finding known bugs in previous versions of the analyses, that have now been fixed, and also in revealing known limitations of different analyses for some language features. For example, the fact that some of the aliasing domains do not support rational terms was easily detected, and also that many domains do not support attributed variables. Some new, but still not unexpected bugs were found in one of the most experimental domains (etermsvar). Furthermore, also a few new bugs were found even in mature domains. These are typically related to the handling of rarely-used built-ins, which explains why they have gone unnoticed, but they are still bugs and have been (or are being) fixed. In addition, while the testing process was aimed at the domains, it also uncovered some bugs in related components of the Ciao assertion framework and their integration, which have been fixed too. We thus conclude that our approach is indeed effective in revealing and discovering bugs and inconsistencies in the domains and also in the overall framework.

Another overall conclusion from the experiment is that benchmark selection is very important when focusing our approach on testing specific domains. No bugs were found for the most mature domains using standard benchmarks and the undergraduate Prolog assignments. The subtle bugs mentioned before in less-used built-ins were found instead when using benchmarks extracted from Ciao’s code base, i.e., in complex, system code. On the other hand, a good number of errors were found in the experimental domain with even the simpler benchmarks. In fact, in this case, the many errors triggered obfuscated sometimes the real (possibly multiple) origin of the problems, but this is to be expected in immature code: consider for example that just the ISO-standard contains a very large set of built-ins and the implementation of an experimental domain typically does not support all of them.

Finally, it is important to point out that we also found out that there are some bugs that are unlikely to be found with benchmarks like the ones used in the tests, because they are bugs that will probably never occur in realistic programs. One example is the simple bug found in [10] for the handler of the builtin =/2 in the sharing-freeness domain. The code did not consider that the two arguments could be the same variable, and thus the analysis failed for any program with the literal X=X. Since that literal always succeeds and is redundant in every program, it will likely not appear in any reasonable benchmark and this error would not be detected by our tool. To find bugs of this kind with our approach, randomly generated benchmarks would be needed.

### Related Work

The need for validating program analyzers was discussed by [8], and the topic has motivated interesting research over the past years. On the formal verification side, there have been some pen-and-paper proofs, such as that of the Astree analyzer [12], some automatic and interactive proofs, such as [16, 42], and some verification attempts, which include [2, 30, 25]. Testing efforts for program analyzers include e.g., static analyzers [47, 50, 13, 27], symbolic execution engines [26], refactoring engines [14], compilers [48, 28, 45, 29, 41, 31], SMT solvers [3], among others. Most of these testing approaches

---

<table>
<thead>
<tr>
<th>Abstract Domain</th>
<th>Properties Abstracted</th>
<th>Maturity Level</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>shfr</td>
<td>aliasing, modes</td>
<td>mature</td>
<td>[37]</td>
</tr>
<tr>
<td>def</td>
<td>aliasing, modes</td>
<td>intermediate</td>
<td>[19]</td>
</tr>
<tr>
<td>gr</td>
<td>aliasing, modes</td>
<td>intermediate</td>
<td>[6]</td>
</tr>
<tr>
<td>eterm</td>
<td>types</td>
<td>mature</td>
<td>[46]</td>
</tr>
<tr>
<td>etermvar</td>
<td>types</td>
<td>experimental</td>
<td>[46]</td>
</tr>
<tr>
<td>nf</td>
<td>failure</td>
<td>mature</td>
<td>[15, 6]</td>
</tr>
<tr>
<td>det</td>
<td>determinism</td>
<td>mature</td>
<td>[32, 33]</td>
</tr>
</tbody>
</table>

Table 1. Domains used for the evaluation of the approach.
Results. While we are planning on performing a larger set of experiments, the results so far are promising and have allowed us to draw some interesting conclusions and observations. A good number of bugs and inconsistencies were indeed found using the technique, many of them known but also some new ones. First, our experiment was successful in finding known bugs in previous versions of the analyses, that have now been fixed, and also in revealing known limitations of different analyses for some language features. For example, the fact that some of the aliasing domains do not support rational terms was easily detected, and also that many domains do not support attributed variables. Some new, but still not unexpected bugs were found in one of the most experimental domains (eternuswar). Furthermore, also a few new bugs were found even in mature domains. These are typically related to the handling of rarely-used built-ins, which explains why they have gone unnoticed, but they are still bugs and have been (or are being) fixed. In addition, while the testing process was aimed at the domains, it also uncovered some bugs in related components of the Ciao assertion framework and their integration, which have been fixed too. We thus conclude that our approach is indeed effective in revealing and discovering bugs and inconsistencies in the domains and also in the overall framework.

Another overall conclusion from the experiment is that benchmark selection is very important when focusing our approach on testing specific domains. No bugs were found for the most mature domains using standard benchmarks and the undergraduate Prolog assignments. The subtle bugs mentioned before in less-used built-ins were found instead when using benchmarks extracted from Ciao's code base, i.e., in complex, system code. On the other hand, a good number of errors were found in the experimental domain with even the simpler benchmarks. In fact, in this case, the many errors triggered obfuscated sometimes the real (possibly multiple) origin of the problems, but this is to be expected in immature code: consider for example that just the ISO-standard contains a very large set of built-ins and the implementation of an experimental domain typically does not support all of them.

Finally, it is important to point out that we also found out that there are some bugs that are unlikely to be found with benchmarks like the ones used in the tests, because they are bugs that will probably never occur in realistic programs. One example is the simple bug found in [10] for the handler of the builtin =/= in the sharing-freeness domain. The code did not consider that the two arguments could be the same variable, and thus the analysis failed for any program with the literal X=X. Since that literal always succeeds and is redundant in every program, it will likely not appear in any reasonable benchmark and this error would not be detected by our tool. To find bugs of this kind with our approach, randomly generated benchmarks would be needed.

7 Related Work
The need for validating program analyzers was discussed by [8], and the topic has motivated interesting research over the past years. On the formal verification side, there have been some pen-and-paper proofs, such as that of the Astree analyzer [12], some automatic and interactive proofs, such as [16] [12], and some verification attempts, which include [25] [30]. Testing efforts for program analyzers include e.g., static analyzers [47] [50] [13] [27], symbolic execution engines [26], refactoring engines [14], compilers [48] [28] [45] [29] [41] [31], SMT solvers [9], among others. Most of these testing approaches

---

6 We are working on including the technique as part of the Ciao continuous integration infrastructure, and plan to report on a larger number of CiaoPP analyses over a wider range of programs.
use programs in the target language as test cases and apply testing techniques like fuzzing (e.g., [48, 26, 3]) or differential testing [34], (e.g., [48, 28, 26, 3, 27]). In [7] and [30] abstract domain properties are tested, the later using QuickCheck [11]. Among the different approaches mentioned, the closest to ours are those that cross-check dynamically observed and statically inferred properties [47, 50, 13, 1].

In [47] the actual pointer aliasing in concrete executions is cross-checked with the pointer aliasing inferred by an aliasing analyzer. Compared to us, they require significant tailored instrumentation which cannot be reused for testing other analyses. However, their approach is agnostic to the (C) aliasing analyzer.

Another cross-check is done in [50] for C model checkers and the reachability property, but they obtain the assertions dynamically, and check them statically, complementarily to our approach. Unlike us, they again need tailored instrumentation that cannot be reused to test other analyses, and their benchmarks must be deterministic and with no input, the later limiting the power of the approach as a testing tool. However, their approach is agnostic to the (C) model checker.

In [13] a wide range of static analysis tests are performed over randomly generated programs. Among others, they check dynamically, at the end of the program, one assertion inferred statically, and they perform the sanity check of ensuring that the analyzer behaves as an interpreter when run from a singleton set of initial states.

8 Conclusions and Future Work

We have proposed a simple, automatic method for testing abstract interpretation-based static analyzers based on checking that the properties inferred statically are satisfied dynamically. We have leveraged the Ciao unified assertion language and framework, and have constructed a prototype implementation of our method with little effort by combining components already present in the framework: the static analyzer, the runtime-checker, the random test-case generator, and the unit-tester. We just wrote a very reduced amount of glue code that pilots the combination and interplay of the intervening components. We have applied our prototype to a good number of the abstract interpretation-based analyses in CiaoPP, which represent different levels of code maturity. The results are encouraging and show that our tool can effectively discover and locate, not only old errors in previous versions (that are obviously less interesting since they were fixed in newer versions), but also new, interesting and unexpected, non-trivial, previously undetected bugs.

We have left as future work other interesting sanity checks enabled by Ciao’s integrated and unified assertion language and framework, such as testing the assertion simplifier, which simplifies programs discarding (parts of) check assertions that have been proven statically. This could be done by analyzing a benchmark without assertions, simplifying the assertions output, and checking that there are no assertions left. We also plan to use the test case generation framework to do differential testing of several program optimizations and transformations over a suite of benchmarks, by just checking that they produce the same outputs for the same randomly generated inputs. A recent paper [9] suggested defining and using distances in abstract domains and between abstract semantics (i.e., between abstract AND-OR trees inferred by the analyzer). We plan to implement an instrumentation that uses such distances to test analysis precision and measure coverage within our approach: if the distance between the dynamic under-approximation and the static over-approximation of the program semantics is small, it means that the analysis was precise and the random inputs had good coverage; otherwise, either the analysis was imprecise, or the test case generation had poor coverage. We plan to investigate heuristics to distinguish both cases.
References


18
Slicing Unconditional Jumps with Unnecessary Control Dependencies

Carlos Galindo, Sergio P´erez, and Josep Silva

VRAIN
Universitat Polit`ecnica de Val`encia
Cam´ı de Vera s/n
E-46022 Val`encia, Spain

Abstract.
Program slicing is an analysis technique that has a wide range of applications, ranging from compilers to clone detection software, and that has been applied to practically all programming languages. Most program slicing techniques are based on a widely extended program representation, the System Dependence Graph (SDG). However, in the presence of unconditional jumps, there exist some situations where most SDG-based slicing techniques are not as accurate as possible, including more code than strictly necessary. In this paper, we identify one of these scenarios, pointing out the cause of the inaccuracy, and describing the initial solution to the problem proposed in the literature, together with an extension, which solves the problem completely. These solutions modify both the SDG generation and the slicing algorithm. Additionally, we propose an alternative solution, that solves the problem by modifying only the SDG generation, leaving the slicing algorithm untouched.

Keywords: Program analysis, Program slicing, Unconditional jumps

1 Introduction
Program slicing [20, 18] is a technique for program analysis and transformation whose main objective is to extract from a program the set of statements that affect a given set of variables in a specific statement, the so-called slicing criterion. The programs obtained with program slicing are called slices, and they are used in many areas such as debugging [1], program specialization [2], software maintenance [7], code obfuscation [13], etc.

There exist several algorithms and data structures to represent programs that can be used to compute slices, but the most efficient and broadly used data structure is the system dependence graph (SDG), introduced by Horwitz et al. [9]. It is computed from the program’s source code, and once built, a slicing criterion is chosen and mapped to the graph, that is then traversed with the algorithm proposed in [9] to compute the corresponding slice.


Slicing Unconditional Jumps with Unnecessary Control Dependencies

Carlos Galindo, Sergio Pérez, and Josep Silva

VRAIN
Universitat Politècnica de València
Carní de Vera s/n
E-46022 València, Spain
{cargaji,serperu,jsilva}@dsic.upv.es

Abstract. Program slicing is an analysis technique that has a wide range of applications, ranging from compilers to clone detection software, and that has been applied to practically all programming languages. Most program slicing techniques are based on a widely extended program representation, the System Dependence Graph (SDG). However, in the presence of unconditional jumps, there exist some situations where most SDG-based slicing techniques are not as accurate as possible, including more code than strictly necessary. In this paper, we identify one of these scenarios, pointing out the cause of the inaccuracy, and describing the initial solution to the problem proposed in the literature, together with an extension, which solves the problem completely. These solutions modify both the SDG generation and the slicing algorithm. Additionally, we propose an alternative solution, that solves the problem by modifying only the SDG generation, leaving the slicing algorithm untouched.

Keywords: Program analysis, Program slicing, Unconditional jumps

1 Introduction

Program slicing [20, 18] is a technique for program analysis and transformation whose main objective is to extract from a program the set of statements that affect a given set of variables in a specific statement, the so-called slicing criterion. The programs obtained with program slicing are called slices, and they are used in many areas such as debugging [1], program specialization [2], software maintenance [7], code obfuscation [13], etc.

There exist several algorithms and data structures to represent programs that can be used to compute slices, but the most efficient and broadly used data structure is the system dependence graph (SDG), introduced by Horwitz et al. [9]. It is computed from the program’s source code, and once built, a slicing criterion is chosen and mapped to the graph, that is then traversed with the algorithm proposed in [9] to compute the corresponding slice.
The SDG is the result of assembling a set of graphs that represent information about a program. Figure 1 depicts how the SDG is built using the control-flow graph (CFG) as the starting graph. First, using the CFG of each function definition in the code, two different graphs are built: (i) the control dependence graph (CDG) [6] and (ii) the data dependence graph (DDG) [19, 6]. The union of both graphs results in the program dependence graph (PDG) [14, 6], which represents all data and control dependencies inside a concrete function. Finally, PDG’s function calls, definitions and their parameters are linked with interprocedural arcs, generating the final SDG. The SDG can be traversed from a slicing criterion to produce a slice in linear time with the algorithm proposed in [9].

As all the aforementioned graphs conforming the SDG represent different relationships of the program, an improvement in the accuracy of these graphs results in a direct impact on the accuracy of the SDG. Throughout the years, the SDG has been augmented with different dependencies, and several techniques have been defined to properly represent complex situations: interprocedural alternatives to compute executable slices [4], extensions of the CFG to represent interprocedural control dependencies [17], object-oriented language representations and slicing [12], or program slicing in concurrent environments [10, 5] are some examples of the evolution of the SDG.

For the purpose of this paper, we are interested in the evolution of the unconditional control flow treatment for program slicing. In this specific area, the initial proposal was the one introduced by Ball and Horwitz [3]. In their work, the authors considered a simplified language with scalar variables and constants, assignment statements, jump statements (goto, break, halt, etc.), conditional statements (if-then, if-then-else), and loops (while and repeat). Despite the simplicity of the given programming language, the ideas proposed can be applied to any kind of unconditional jumps present in other programming languages. In this paper, we provide examples using the break statement in the Java programming language, even though the problem presented and its solution can be applied to any statement that represents an unconditional jump. The following example illustrates the problem identified by Ball and Horwitz after their proposal.

Example 1 (Unconditional jump subsumption [3]). Consider the Java method shown below on the left-hand side:
Carlos Galindo, Sergio P´erez, and Josep Silva

for each statement in the program, and

A are arcs that represent the execution flow between the nodes:

Statement node. Any statement that is not a conditional jump. These nodes have one outgoing edge pointing to the next statement of the program.

Predicate node. Any conditional jump statement, such as if, while, etc. These nodes have two outgoing edges labelled true and false, leading to the statements that would be executed regarding the condition evaluation.

The CFG of the Original program in Example 1 is shown in Figure 2 (left), where \( N \) are all the nodes and \( A \) all the solid black arcs in the graph (we will ignore the dashed red arcs for now, since they are not part of the CFG). In this graph, all nodes with just one outgoing arc of \( A \) represent statements, while all nodes with two outgoing arcs of \( A \) labeled with T or F represent predicates.

![ACFG and CDG of the code in Example 1.](image)

Fig. 2. ACFG (left) and CDG (right) of the code in Example 1.

This method contains a while statement, from which the execution may exit naturally or through any of the break statements. To represent the rest of statements and conditional expressions, uppercase letters are used; and, for simplicity, we can assume that there are no data dependencies between them.

Now consider statement C as the slicing criterion: each input that produces a computation in the original program that reaches C must produce a computation in the slice that also reaches C. Note that C is only executed when \( X \) is true and \( Y \) is false.

The code in the centre displays the computed slice by Ball and Horwitz’s approach; the code on the right-hand side is the minimal slice. As can be observed, the break in line 6 and its surrounding if statement (if (Z)) have been unnecessarily included in the slice, since the evaluation of Z does not influence the execution of a break after being the Y statement evaluated to true. Their inclusion would not be specially problematic, if it were not for the condition of the if statement (Z), which may include extra data dependencies that are unnecessary in the slice and that may lead to include other unnecessary statements, making the slice even more imprecise.

The rest of the paper is structured as follows: Section 2 illustrates the rationale behind the problem shown in Example 1, detailing how dependencies are generated, identifying when the problem shows up, and describing the solution proposed by Kumar and Horwitz in [11], where the authors introduced changes in two steps of the process shown in Figure 1. Section 3 proposes an alternative solution that is simpler and does not need to change the slicing algorithm, lowering the time complexity while preserving completeness at all times. Finally, Section 4 concludes the article outlining the main contributions.

2 Unconditional jumps and the PPDG

To keep the paper self-contained, we start with the definition of control flow graph.

Definition 1 (Control-flow graph). Given a program \( P \), the control flow graph of \( P \) is a graph \((N,A)\) where \( N \) is a set of nodes that contains one node
for each statement in the program, and A are arcs that represent the execution flow between the nodes:

**Statement node.** Any statement that is not a conditional jump. These nodes have one outgoing edge pointing to the next statement of the program.

**Predicate node.** Any conditional jump statement, such as if, while, etc. These nodes have two outgoing edges labelled true and false, leading to the statements that would be executed regarding the condition evaluation.

The CFG of the **Original program** in Example 1 is shown in Figure 2 (left), where N are all the nodes and A all the solid black arcs in the graph (we will ignore the dashed red arcs for now, since they are not part of the CFG). In this graph, all nodes with just one outgoing arc of A represent statements, while all nodes with two outgoing arcs of A labeled with T or F represent predicates.

![ACFG and CDG graphs](image)

**Fig. 2.** ACFG (left) and CDG (right) of the code in Example 1.

The control flow graph is the basis to calculate control dependencies in a program and, thus, the control dependence graph.

**Definition 2 (Control dependence).** Let \( G \) be a CFG, Let \( X \) and \( Y \) be nodes in \( G \). A node \( Y \) post-dominates a node \( X \) in \( G \) if every directed path from \( X \) to the End node passes through \( Y \). Node \( Y \) is control dependent on node \( X \) if and only if \( Y \) post-dominates one but not all of \( X \)'s CFG successors.

**Definition 3 (Control dependence graph).** Given a program \( P \) and its associated CFG \( G_{CFG} = (N, A) \), the control dependence graph (CDG) of \( P \) is a graph \( G_{CDG} = (N, A') \) where \( (x, y) \in A' \) if and only if node \( y \in N \) is control-dependent on node \( x \in N \).
Unconditional jump statements distort the usual understanding of control dependence, and they invalidate the standard representation of control dependencies in the CDG. Example 2 shows that the standard definition of control dependence is insufficient in presence of unconditional jumps.

Example 2 (Control dependencies induced by unconditional jumps). Consider the following code on the left-hand side and the slicing criterion \(x\) in the last line.

```plaintext
Original program
1  x = 0;
3  x++;
4  if (x>10)
5    break;
6 }
7  print(x);

Wrong slice
1  x = 0;
2  while (true) {
3    x++;
4    if (x>10)
5      break;
6 }
7  print(x);
```

The slice of this code is the whole code (everything is needed to reach the slicing criterion). Nevertheless, according to Definition 2, the \texttt{break} statement in line 5 does not control any other statement, that is, no statement depends on the \texttt{break} statement. Therefore, the (wrong) slice computed with the standard definition of control dependence would be the code on the right. This is an infinite loop that never reaches the slicing criterion. Clearly, the execution of \texttt{print(x)} is in some way controlled by the execution of \texttt{break} and, thus, unconditional jumps induce some kind of control dependencies that are not captured in Definition 2.

To deal with this problem (i.e. unconditional control flow statements), Ball and Horwitz [3] proposed a modification of the CFG in presence of unconditional control flow statements, which result in a CDG with augmented dependencies. This approach is the most popular one and the one used in most of the subsequent literature [15, 11, 16]. The main modification applied to the CFG consists in the introduction of a third category of nodes in the definition of the CFG:

\textbf{Pseudo-predicates}. Unconditional jumps (i.e. \texttt{break}, \texttt{goto}, \texttt{return}\footnote{The target of the jump in a \texttt{return} statement is the exit of the procedure it’s in, from which control will be handed back to the previous procedure in the call stack.}, etc.) are treated like predicates, where the outgoing edge labelled \texttt{false} is marked as non-executable—because there is no possible execution where such edge would be possible, according to the definition of the CFG [8]. For unconditional jumps, the \texttt{true} edge leads to the statement at the jump destination, and the \texttt{false} edge to the statement that would be executed if the jump was skipped.

The graph obtained from adding the \texttt{false} arcs to the pseudo-predicate nodes of a CFG is called the \textit{Augmented CFG} (ACFG). As a consequence of the appearance of pseudo-predicate nodes, in an ACFG every statement between an unconditional jump and its destination is control-dependent on it (see Definition 2), as can be seen in Example 3.
Example 3 (Control dependencies generated by unconditional jumps). Consider again the ACFG in Figure 2 (left), which represents the code in Example 1. Here, solid arrows represent edges that come out from statements, predicates, and true pseudo-predicate branches; and dashed red arrows represent the non-executable (false) branches of pseudo-predicates. When we transform this ACFG to a CDG, we obtain the CDG in Figure 2 (right), where the slice with respect to variable C is represented with grey nodes.

Even though Ball and Horwitz solved the exposed problem with the definition of the ACFG, there was still a problem they were not able to solve. This problem is represented in the code of Example 1. It appears when there are two different unconditional jumps with the same jump destination. Due to the false pseudo-predicate arcs in the ACFG, all the statements between the first unconditional jump and the second one become directly control-dependent on the first jump, including the second one. Similarly, all the statements located between the second jump and the destination statement become directly control-dependent on the second jump. As a result of the transitive dependence, when any statement between the second jump and the destination statement is required, the inclusion of both unconditional jump statements in the slice is unavoidable. The inclusion of the first jump statement will increase the size of the slice with all its dependencies, leading to an imprecise slice. The solution proposed in [3] is complete, but not as accurate as it was expected to be.

Ball and Horwitz were aware of the aforementioned problem and, some years later, Kumar and Horwitz proposed a solution in [11]. Their solution was based on two main modifications:

1. **A new definition of control dependence in the presence of pseudo-predicates.** “Node Y is control-dependent on node X if and only if Y post-dominates, in the CFG, one but not all of X’s ACFG successors”. The resulting graph was called the pseudo-predicate PDG (PPDG).

2. **A new slicing algorithm.** The new algorithm established some restrictions in the slicing traversal. “To compute the slice from node S, include S itself and all of its data and control-dependence predecessors in the slice. Then follow backwards all data-dependence edges, and all control-dependence edges whose targets are not pseudo-predicates; add each node reached during this traversal to the slice.”

By the introduction of these novelties, the accuracy of the slice was improved, since it is not possible to add in the slice two pseudo-predicate nodes that jump to the same destination unless one of them is the slicing criterion itself. This approach solved the problem of Example 1, proposed in [3].

3 Alternative solution: unnecessary control dependencies

In this section, we propose an alternative solution to the unconditional jump problem shown in the previous section. The key idea of our approach is to identify
which edges of the CDG are responsible for the inaccurate slices and define a method to avoid building them in the graph generation process.

To properly reason about the accuracy of our approach, we provide a formal definition of slicing criterion and slice.

**Definition 4 (Slicing criterion).** Let $P$ be a program. A slicing criterion $C$ of $P$ is a tuple $(s, v)$ where $s$ is a statement in $P$ and $v$ is a set of variables that are used or defined in $s$.

**Definition 5 (CDG slice).** Given a CDG $G = (N, A)$ and a slicing criterion $(s, v)$, where $n \in N$ represents $s$ in $G$, a CDG slice of $n$ is a subgraph $G' = (N', A')$ such that:

1. $N' \subseteq N$.
2. $\forall n' \in N'$, $n$ is control dependent on $n'$ and $n'$ is needed to execute $n$ the same number of times as in $G$ (the original program).
3. $A' = \{(x, y) \in A \mid x, y \in N'\}$.

The standard slicing algorithm, denoted $\text{slice}(G, C)$, collects all nodes that are reachable from the node in $G$ associated with the slicing criterion $C$ traversing backwards the CDG arcs.

We have identified a general situation in which some control dependencies should be omitted. If those control dependencies are removed from the CDG, then the standard slicing algorithm is still complete and precision is kept the same or improved. Consider a CDG $G$ with two unconditional jump statements $x$ and $y$ that jump to the same destination, with an arc $(x, y)$ in $G$. There exists a CDG $G'$ with the same set of nodes and a set of arcs obtained by deleting all the control arcs in $G$ with $y$ as target, that produces more accurate program slices. Formally,

**Theorem 1.** Let $G = (N, A)$ be a CDG. Let $x \in N$ be any unconditional jump statement. Let $y \in N$ be an unconditional jump statement without any variable use or definition that jumps to the same destination as $x$. Let $G' = (N, A')$ where $A' = (A \setminus \{(w, y) \mid w \in N\})$. For all slicing criterion $C$, $\text{slice}(G', C)$ is a CDG slice.

**Proof.** We prove the theorem by means of a generic code that captures all possible scenarios that can happen under the conditions of the theorem. We consider two unconditional jump statements, $x$ as the first jump statement and $y$ as the second one. First, $x$ and $y$ cannot be sequential statements because in that case $y$ would be dead code. This forces us to enclose $x$ inside a conditional structure. As $y$ does not define or use any variable, we add the statement $s_1$ and place an external conditional structure to also prevent it to be dead code. This generic code is depicted in Figure 3 (left). Any statement or groups of them added to this code before or after $x$ or $y$ would produce a similar topology that would not affect the proof. The reason is that any statement represented by a set of nodes has only one successor in the CFG and can never be the source of a control dependence (see Section 2.3 in [3]).
We distinguish two possible scenarios according to the slice computed by $\text{slice}(G', n)$:

(i) $y \not\in \text{slice}(G', n)$ (Figure 4, left). In this case, node $y$ is not needed to execute $n$ and, thus, the removal of the arcs that end in $y$ do not affect the computation of $\text{slice}(G', n)$ because they are never traversed. Therefore, all nodes needed to execute $n$ belong to the slice (condition 2 in Definition 5) and also all arcs induced by them are kept in the slice (condition 3 in Definition 5). Hence, $\text{slice}(G', n)$ is a CDG slice.

(ii) $y \in \text{slice}(G', n)$ (Figure 4, right). First, according to Definition 4, node $y$ cannot be selected as slicing criterion, as it does not define or use any variables of the program according to the theorem conditions imposed on $y$. Then, because no data dependence exists on $y$, the only possibility to include $y$ in $\text{slice}(G', n)$ is because some statement between $y$ and the jump destination of $y$ is included in the slice ($s_1$ in our graph in Figure 4 (right)). Because of that, there is an execution path where $y$ affects the execution of this statement. In the case that $\text{cond}_1$ was a loop, $y$ would be control dependent on $\text{cond}_1$ itself, including $\text{cond}_1$ in $\text{slice}(G', n)$ but, in this case, we would obtain the same result because $s_1$ is also control dependent on $\text{cond}_1$ and thus, included in $\text{slice}(G', n)$.

We have two possible scenarios to execute $n$ (see the ACFG in Figure 3 (centre)):

- $s_1$ is executed. Then, $\text{cond}_1$ is false and $\text{cond}_2$, $x$, and $y$ are not executed (they can be excluded from $\text{slice}(G', n)$).
- Either $x$ or $y$ are executed. As the result of executing $x$ and $y$ is functionally the same (the program execution continues at the destination of $y$), there is no difference between taking one path of $\text{cond}_2$ or another. Therefore, $\text{cond}_2$, $x$ and $y$ can be replaced by $y$ without modifying the behaviour of the program; making the control dependency arcs from $\text{cond}_2$ and $x$ to $y$ unnecessary.

In the three cases, the removal of the arcs that end in $y$ ensure that the three conditions in Definition 5 hold. Thus, $\text{slice}(G', n)$ is a CDG slice.

Algorithm 1 formalizes the new CDG generation process, which removes the unnecessary arcs. To perform that task, the algorithm uses an ACFG as the starting point. The algorithm uses the following functions and sets:

- $\text{genControlArcs}$: It inputs an ACFG and outputs all control arcs that can be obtained according to Definition 2.
- $\text{unjumps}$: This is a set with all nodes that represent an unconditional jump.
- $\text{jumpDest}$: This function inputs a CDG node $n$ that represents an unconditional jump statement and outputs the destination of the jump.

Algorithm 1 first generates all control dependencies in the ACFG. Then, each control dependency $n \rightarrow n'$ is inspected to determine whether both $n$ and $n'$ are unconditional jumps with the same destination. If this is the case, then all control arcs that target node $n'$ are removed. This forms the set $A'$. Finally,

---

Fig. 3. Piece of code (left), its ACFG (centre) and its associated CDG (right).

Fig. 4. CDG of our approach and CDG slices w.r.t. $x$ (left) and $s_1$ (right).
Algorithm 1

**CDG Generation Algorithm**

**Input:**
An ACFG \( G \)

**Output:**
A CDG \( G' = (N', A_c) \)

1. \( A_c = \text{genControlArcs}(G) \)
2. For all \((n_s, n_e) \in A_c\) do
   3. if \((n_s, n_e) \in \text{unjumps} \land \text{jumpDest}(n_s) = \text{jumpDest}(n_e))\) then
      4. \( A_c = A_c \setminus (x, n_e) \) \( \forall x \in N \)
   5. end if
   6. end for
3. \( N' = N \setminus \{\text{End}\} \)
4. \( G' = (N', A_c) \)

\( N' \) is calculated by removing the \( \text{End} \) node from \( N \) and the CDG \( G' = (N', A') \) is obtained.

With this generation process, the CDG produced is more accurate than the one produced by Ball and Horwitz. For instance, the CDG associated to the Original program in Example 1 is shown in Figure 5. The CDG slice associated to the slicing criterion \( C \) is shown in grey, and it corresponds to the Minimal slice in Example 1. As can be seen, nodes \( \text{break} \) and \( \text{if}(Z) \) are no longer part of the slice. The structure of this graph represents now a more realistic control dependence, where unconditional jumps to common destinations are not dependent on each other.

![CDG obtained by applying Algorithm 1 to the code in Example 1.](image)

It is worth remarking the main difference between the solution presented in [11] and our approach: the amount of steps of the slicing process that are modified. Both approaches introduce a modification in the CDG generation process. While the amount of arcs generated by Kumar and Horwitz may be lower

We distinguish two possible scenarios according to the slice computed by \( \text{slice}(G', n) \):

(i) \( y \notin \text{slice}(G', n) \) (Figure 4, left). In this case, node \( y \) is not needed to execute \( n \) and, thus, the removal of the arcs that end in \( y \) do not affect the computation of \( \text{slice}(G', n) \) because they are never traversed. Therefore, all nodes needed to execute \( n \) belong to the slice (condition 2 in Definition 5) and also all arcs induced by them are kept in the slice (condition 3 in Definition 5). Hence, \( \text{slice}(G', n) \) is a CDG slice.

(ii) \( y \in \text{slice}(G', n) \) (Figure 4, right). First, according to Definition 4, node \( y \) cannot be selected as slicing criterion, as it does not define or use any variables of the program according to the theorem conditions imposed on \( y \). Then, because no data dependence exists on \( y \), the only possibility to include \( y \) in \( \text{slice}(G', n) \) is because some statement between \( y \) and the jump destination of \( y \) is included in the slice (\( s_1 \) in our graph in Figure 4 (right)). Because of that, there is an execution path where \( y \) affects the execution of this statement. In the case that \( \text{cond}_1 \) was a loop, \( y \) would be control dependent on \( \text{cond}_1 \) itself, including \( \text{cond}_1 \) in \( \text{slice}(G', n) \) but, in this case, we would obtain the same result because \( s_1 \) is also control dependent on \( \text{cond}_1 \) and thus, included in \( \text{slice}(G', n) \).

We have two possible scenarios to execute \( n \) (see the ACFG in Figure 3 (centre)):

- \( s_1 \) is executed. Then, \( \text{cond}_1 \) is \( \text{false} \) and \( \text{cond}_2 \), \( x \), and \( y \) are not executed (they can be excluded from \( \text{slice}(G', n) \)).
- Either \( x \) or \( y \) are executed. As the result of executing \( x \) and \( y \) is functionally the same (the program execution continues at the destination of \( y \)), there is no difference between taking one path of \( \text{cond}_2 \) or another. Therefore, \( \text{cond}_2 \), \( x \), and \( y \) can be replaced by \( y \) without modifying the behaviour of the program; making the control dependency arcs from \( \text{cond}_2 \) and \( x \) to \( y \) unnecessary.

In the three cases, the removal of the arcs that end in \( y \) ensure that the three conditions in Definition 5 hold. Thus, \( \text{slice}(G', n) \) is a CDG slice.

Algorithm 1 formalizes the new CDG generation process, which removes the unnecessary arcs. To perform that task, the algorithm uses an ACFG as the starting point. The algorithm uses the following functions and sets:

- \( \text{genControlArcs}(\cdot) \). This function inputs an ACFG and outputs all control arcs that can be obtained according to Definition 2.
- \( \text{unjumps} \). This is a set with all nodes that represent an unconditional jump.
- \( \text{jumpDest}(\cdot) \). This function inputs a CDG node \( n \) that represents an unconditional jump statement and outputs the destination of the jump.

Algorithm 1 first generates all control dependencies in the ACFG. Then, each control dependency \( n \to n' \) is inspected to determine whether both \( n \) and \( n' \) are unconditional jumps with the same destination. If this is the case, then all control arcs that target node \( n' \) are removed. This forms the set \( A' \). Finally,
or greater than the amount of arcs generated in the initial proposal \( [3] \), the amount of arcs generated in our approach is always equal or lower than in the initial proposal. In addition, the approach by Kumar and Horwitz needs to change the standard SDG-traversal algorithm, introducing an overhead when calculating slices. On the contrary, in our approach the SDG-traversal algorithm remains untouched, keeping the slicing process as a graph reachability problem and ensuring the slicing cost proposed by Ottenstein and Ottenstein in \( [14] \).

4 Conclusions

Ball and Horwitz proposed the first program slicing technique with a specific treatment for unconditional jumps. Even though their technique produces complete slices in all cases, they were aware that accuracy could be improved, and they proposed a challenging example (analogous to Example 1) where the computed slice was bigger than needed. Some years later, Kumar and Horwitz solved this accuracy problem changing the definition of control dependencies and redefining the standard slicing algorithm.

In this paper, we propose an alternative approach that solves the problem performing fewer changes to the standard approach. Our approach only needs to change the CDG produced, and all the other phases of program slicing (including SDG traversal) remain unchanged. We have theoretically proven the correctness of our approach.

5 Acknowledgements

This work has been partially supported by the EU (FEDER) and the Spanish MCI/AEI under grants TIN2016-76843-C4-1-R and PID2019-104735RB-C41, and by the Generalitat Valenciana under grant Prometeo/2019/098 (DeepTrust).

References


\begin{algorithm}
\begin{algorithmic}
\State \textbf{Input:} An ACFG \( G = (N, A) \).
\State \textbf{Output:} A CDG \( G' = (N', A_c) \).
\State \textbf{1:} \( A_c = \text{genControlArcs}(G) \)
\State \textbf{2:} \textbf{for} all \( (n_x, n_c) \in A_c \) \textbf{do}
\State \textbf{3:} \quad \textbf{if} \( (n_x, n_c) \in \text{unjumps} \land \text{jumpDest}(n_x) == \text{jumpDest}(n_c) \) \textbf{then}
\State \textbf{4:} \quad \quad \( A_c = A_c \setminus (x, n_c) \) \forall x \in N \)
\State \textbf{5:} \quad \textbf{end if}
\State \textbf{6:} \textbf{end for}
\State \textbf{7:} \( N' = N \setminus \{ \text{End} \} \)
\State \textbf{8:} \( G' = (N', A_c) \)
\end{algorithmic}
\end{algorithm}

\( N' \) is calculated by removing the End node from \( N \) and the CDG \( G' = (N', A') \) is obtained.

With this generation process, the CDG produced is more accurate than the one produced by Ball and Horwitz. For instance, the CDG associated to the Original program in Example 1 is shown in Figure 5. The CDG slice associated to the slicing criterion \( C \) is shown in grey, and it corresponds to the Minimal slice in Example 1. As can be seen, nodes break and if(Z) are no longer part of the slice. The structure of this graph represents now a more realistic control dependence, where unconditional jumps to common destinations are not dependent on each other.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{cdg.png}
\caption{CDG obtained by applying Algorithm 1 to the code in Example 1.}
\end{figure}

It is worth remarking the main difference between the solution presented in [11] and our approach: the amount of steps of the slicing process that are modified. Both approaches introduce a modification in the CDG generation process. While the amount of arcs generated by Kumar and Horwitz may be lower
or greater than the amount of arcs generated in the initial proposal ([3]), the
amount of arcs generated in our approach is always equal or lower than in the ini-
tial proposal. In addition, the approach by Kumar and Horwitz needs to change
the standard SDG-traversal algorithm, introducing an overhead when calculating
slices. On the contrary, in our approach the SDG-traversal algorithm remains
untouched, keeping the slicing process as a graph reachability problem and en-
suring the slicing cost proposed by Ottenstein and Ottenstein in [14].

4 Conclusions

Ball and Horwitz proposed the first program slicing technique with a specific
treatment for unconditional jumps. Even though their technique produces com-
plete slices in all cases, they were aware that accuracy could be improved, and
they proposed a challenging example (analogous to Example 1) where the com-
puted slice was bigger than needed. Some years later, Kumar and Horwitz solved
this accuracy problem changing the definition of control dependencies and re-
defining the standard slicing algorithm.

In this paper, we propose an alternative approach that solves the problem
performing fewer changes to the standard approach. Our approach only needs to
change the CDG produced, and all the other phases of program slicing (including
SDG traversal) remain unchanged. We have theoretically proven the correctness
of our approach.

5 Acknowledgements

This work has been partially supported by the EU (FEDER) and the Span-
ish MCI/AEI under grants TIN2016-76843-C4-1-R and PID2019-104735RB-C41,
and by the Generalitat Valenciana under grant Prometeo/2019/098 (DeepTrust).

References

for WS-BPEL programs based on predicate switching and program slicing. Journal
3. T. Ball and S. Horwitz. Slicing programs with arbitrary control-flow. In Proceedingsof the First International Workshop on Automated and Algorithmic Debugging,
4. D. Binkley. Precise executable interprocedural slices. ACM Letters on Program-
5. Z. Chen and B. Xu. Slicing concurrent java programs. SIGPLAN Not., 36(4):41–47,
6. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph
and its use in optimization. ACM Transactions on Programming Languages and
Abstract. Repeated recursion unfolding is a new approach that repeatedly unfolds a recursion with itself and simplifies it while keeping all unfolded rules. Each unfolding doubles the number of recursive steps covered. This reduces the number of recursive rule applications to its logarithm at the expense of introducing a logarithmic number of unfolded rules to the program.

Efficiency crucially depends on the amount of simplification inside the unfolded rules. We prove a super-linear speedup theorem in the best case, i.e. speedup by more than a constant factor. Our optimization can lower the time complexity class of a program. In this paper, the super-linear speedup is within bounds: it holds up to an arbitrary but chosen upper bound on the number of recursive steps.

We also report on the first results with a prototype implementation of repeated recursion unfolding. A simple program transformation completely removes recursion up to the chosen bound. The actual runtime improvement quickly reaches several orders of magnitude.

Keywords. Program Transformation, Program Optimization, Super-linear Speedup, Recursion, Speedup Theorem, Time Complexity.

1 Introduction

In the context of rule-based and logic programming, unfolding is the program transformation that replaces a call (goal) in the body of a rule by the body of a rule whose head is matched by the goal. This saves a rule application at runtime. More importantly, the resulting code can be simplified for efficiency.

Example 1 (Summation). Consider the following simple recursive program written in CHR (Constraint Handling Rules). It adds all numbers from 1 to n.

\[
\begin{align*}
\text{r}_0 & : \text{sum}(N,S) \iff N > 1 \land S := N + S \text{1, sum}(N - 1, S) \\
\text{sum}(N,S) & \iff N = 1 \lor S = 1.
\end{align*}
\]

Head, guard and body of a rule are separated by \(\iff\) and \(|\) respectively. The rules can be understood as a procedure definition for \text{sum}, each rule covering a
Revised 2023-03-28

Repeated Recursion Unfolding for Super-Linear Speedup within Bounds

Thom Frühwirth

Ulm University, Germany
thom.fruehwirth@uni-ulm.de

July 31, 2020

Abstract. Repeated recursion unfolding is a new approach that repeatedly unfolds a recursion with itself and simplifies it while keeping all unfolded rules. Each unfolding doubles the number of recursive steps covered. This reduces the number of recursive rule applications to its logarithm at the expense of introducing a logarithmic number of unfolded rules to the program.

Efficiency crucially depends on the amount of simplification inside the unfolded rules. We prove a super-linear speedup theorem in the best case, i.e., speedup by more than a constant factor. Our optimization can lower the time complexity class of a program. In this paper, the super-linear speedup is within bounds: it holds up to an arbitrary but chosen upper bound on the number of recursive steps.

We also report on the first results with a prototype implementation of repeated recursion unfolding. A simple program transformation completely removes recursion up to the chosen bound. The actual runtime improvement quickly reaches several orders of magnitude.

Keywords. Program Transformation, Program Optimization, Super-Linear Speedup, Recursion, Speedup Theorem, Time Complexity.

1 Introduction

In the context of rule-based and logic programming, unfolding is the program transformation that replaces a call (goal) in the body of a rule by the body of a rule whose head is matched by the goal. This saves a rule application at runtime. More importantly, the resulting code can be simplified for efficiency.

Example 1 (Summation). Consider the following simple recursive program written in CHR (Constraint Handling Rules). It adds all numbers from 1 to n.

\[ r_0 : \text{sum}(N, S) \leftrightarrow N > 1 \mid S := N + S1, \text{sum}(N-1, S1) \]
\[ \text{sum}(N, S) \leftrightarrow N = 1 \mid S = 1. \]

Head, guard and body of a rule are separated by \( \leftrightarrow \) and \( \mid \), respectively. The rules can be understood as a procedure definition for \( \text{sum} \), each rule covering a
case. When a call matches the head of a rule and the guard condition holds, the body of the rule is executed. In CHR code, upper case letters stand for logical variables.

Unfolding the recursive rule with itself will result in

\[ r_1 : \text{sum}(N, S) \iff N > 1, N-1 > 1 \mid S := N+S1, N' = N-1, S1 := N'+S1', \text{sum}(N'-1, S1'), \]

which can be simplified to

\[ r_1 : \text{sum}(N, S) \iff N > 2 \mid S := 2* N-1+S1', \text{sum}(N-2, S1'). \]

Note that this rule \( r_1 \) cannot replace the original recursive rule because it only applies in case \( N > 2 \). It behaves like applying the original rule \( r_0 \) twice. It saves computation time, because we only need about half as many recursive steps as with the original rule alone. Since the simplification of the unfolded rule results in code of similar size and time complexity as the original rule, we can expect to halve the runtime.

If we would now unfold the recursive goal in rule \( r_1 \) with rule \( r_0 \) we have unfolded thrice overall. We can expect a speedup of roughly three times if we replace \( r_1 \) by the new rule. But we can do even better than that if we keep rule \( r_1 \) and unfold rule \( r_1 \) with itself. The resulting rule after simplification is

\[ r_2 : \text{sum}(N, S) \iff N > 4 \mid S := 4 * N-6 + S1, \text{sum}(N-4, S1). \]

This rule results in a four-fold speedup.

We can repeat this process: unfolding the newly generated recursive rule with itself until we reach a given upper bound. We never unfold with the base case. It is ignored. This is what we call repeated recursion unfolding. With each unfolding, the number of original recursive steps simplified into one recursive step of the unfolded rule will double.

Given \( n \) recursive steps with the original rule and \( \log_2(n) \) unfolded rules according to our scheme, it is possible to have at most \( \log_2(n) \) recursive rule applications instead of \( n \). If the unfolded rules take not more time than the original recursive rule, the time complexity class can be lowered and super-linear speedup in runtime can be achieved.

For our scheme to work, we always have to apply the best, most unfolded rule. To implement this behavior, we use another program transformation that removes the recursion altogether. We call this recursionless recursion.

**Overview of the Paper.** Section 2 introduces syntax and semantics of the CHR programming language using a single state transition.

Section 3 defines our program transformation scheme of repeated recursion unfolding with simplification and proves its correctness. We will use summation as our running example.

Section 4 proves that there exists a simple optimal rule application strategy for unfolded recursions with best-case simplification that results in super-linear speedup by lowering the time complexity classes by a factor of up to \( O(n) \).
Section 5 introduces recursionless recursion, a transformation that provides a semi-naive implementation of optimal rule applications. We prove that the worst-case overhead of this scheme is linear in the number of unfolded rules.

In Section 6 we conclude our summation example with benchmarks in our prototype implementation. It improves time complexity from linear to constant. Another example, naive list reversal, improves from quadratic to linear complexity. It runs faster than the hand-optimized built-in list reversal in Prolog, the implementation language of CHR.

Finally, we discuss related work and end with conclusions. The full version of the paper is available online and contains an extended proof and the source code of our examples.

2 Preliminaries

We recall the abstract syntax and the equivalence-based abstract operational semantics of CHR (Constraint Handling Rules) [Frü09] in this section. We also informally describe the refined operational semantics typically realized in sequential implementations of CHR.

2.1 Abstract Syntax of CHR

The CHR language is based on the abstract concept of constraints. Constraints are relations, distinguished predicates of first-order predicate logic. There are two kinds of constraints: built-in (pre-defined) constraints and user-defined (CHR) constraints which are defined by the rules in a CHR program. Built-in constraints can be used as tests in the guard as well as for auxiliary computations in the body of a rule. There are at least the built-in constraints true and false, syntactical equality = over finite terms and the usual relations over arithmetic expressions. Upper-case letters stand for (possibly empty) conjunctions of constraints in definitions, lemmas and theorems.

Definition 1 (CHR Program and Rules). A CHR program is a finite set of rules. A (generalized) simplification rule is of the form

\[ r : H \leftrightarrow C | B, \]

where \( r \) is an optional name (a unique identifier) of a rule. The head \( H \) is a conjunction of user-defined constraints, the optional guard \( C \) is a conjunction of built-in constraints, and the body \( B \) is a goal.

The local variables of a rule are those not occurring in the head of the rule. A renaming (variant, copy) of a goal is obtained by uniformly replacing its variables by other variables.

Conjunctions are understood as multisets of their atomic conjuncts. We often use simple commas to denote logical conjunction to avoid clutter. A goal is a conjunction of built-in and user-defined constraints. A state is a goal.
2.2 Abstract Operational Semantics of CHR

Computations in CHR are sequences of rule applications. The operational semantics of CHR is given by the state transition system. It relies on a structural equivalence between states that abstracts away from technical details in a transition [RBF09,Bet14].

State equivalence treats built-in constraints semantically and user-defined constraints syntactically. Basically, two states are equivalent if their built-in constraints are logically equivalent (imply each other) and their user-defined constraints form syntactically equivalent multisets in this context. For example,

\[ X = Y \wedge Y = < X \wedge c(X, Y) \equiv X = Y \wedge c(X, X) \neq \forall \exists (X, Y). \]

Let \( CT \) be a (decidable) constraint theory for the built-in constraints.

**Definition 2 (State Equivalence).** [RBF09] Let \( C_i \) be the built-in constraints, let \( B_i \) denote user-defined constraints, and let \( \mathcal{V} \) be a set of variables. Variables of a goal or state that do not occur in \( \mathcal{V} \) are called local variables of the goal or state. Two states \( S_1 = (C_1 \wedge B_1) \) and \( S_2 = (C_2 \wedge B_2) \) with local variables \( \bar{x} \) and \( \bar{y} \) that have been renamed apart are equivalent, written \( S_1 \equiv_{\mathcal{V}} S_2 \), if and only if

\[ CT \models \forall (C_1 \rightarrow \exists \bar{y}((B_1 = B_2) \wedge C_2)) \wedge \forall (C_2 \rightarrow \exists \bar{x}((B_1 = B_2) \wedge C_1)) \]

Note that this definitions implies

\[ CT \models \forall (\exists \bar{x}(B_1 \wedge C_1) \leftrightarrow \exists \bar{y}(B_2 \wedge C_2)). \]

It also makes sure that there is a one-to-one correspondence between user-defined constraints as enforced by \( B_1 = B_2 \). It allows the renaming of local variables. Local variables can be removed if logical equivalence is maintained. Occurrences of local variables can be substituted by terms if logical equivalence is maintained. These properties have been shown in [RBF09].

An example illustrates these properties of state equivalence and the effect of the global variables \( \mathcal{V} \):

\[ X = Y \wedge c(X, Y) \equiv_{\{X\}} c(X, X) \text{ but } X = Y \wedge c(X, Y) \neq_{\{X, Y\}} c(X, X). \]

We may drop \( \mathcal{V} \) from the equivalence if it is clear from the context.

Using this state equivalence, the abstract CHR semantics is defined by a single transition (computation step) between states. It defines the application of a rule. If the source state can be made equivalent to a state that contains the head and the guard of a renaming of a rule, then we can apply the rule by replacing the head by the body in the state. Any state that is equivalent to this target state is also in the transition relation.

**Definition 3 (Transition).** A CHR transition (computation step) \( S \rightarrow_r T \) is defined as follows, where \( S \) is called source state and \( T \) is called target state:

\[
\begin{align*}
S & \equiv_{\mathcal{V}} (H \wedge C \wedge G) \neq false \quad \forall H \Rightarrow C \mid B \quad (C \wedge B \wedge G) \equiv_{\mathcal{V}} T \\
S & \rightarrow_r T
\end{align*}
\]
where the rule \( r : H ⇔ C \mid B \) is a renaming of a rule from a given program \( \mathcal{P} \) such that its local variables do not occur in \( G \).

A computation (derivation) of a goal (query) \( S \) with variables \( V \) in a program \( \mathcal{P} \) is a connected sequence \( S_1 \rightarrow r_1, S_{i+1} \) beginning with the initial state (query) \( S_0 \) that is \( S \) and ending in a final state (answer, result) \( S_n \) or otherwise the sequence is infinite and the computation is non-terminating (diverging). We may drop the reference to the rules. The relation \( \rightarrow^* \) denotes the reflexive and transitive closure of \( \rightarrow \).

The goal \( G \) is called context of the rule application. It remains unchanged. It is arbitrary and may be empty. Note that CHR is a committed-choice language, i.e. there is no backtracking or undoing of rule applications.

2.3 Refined Operational Semantics of CHR

Almost all sequential CHR implementations execute queries and rule body constraints from left to right and apply rules top-down following their textual order of the program. This behavior has been formalized in the so-called refined semantics which is a concretization of the abstract operational semantics [DSGH04].

In this refined semantics, a CHR constraint in a goal can be understood as a procedure call that goes efficiently through the rules of the program. If the current goal matches the head constraints of a rule and if, under this matching, the guard check of the rule holds in the current context, the rule is applicable. Given a query, the rules of the program are applied to exhaustion. When a simplification rule is applied, the matched constraints are replaced by the body of the rule.

3 Repeated Recursion Unfolding

We recall a definition of rule unfolding in CHR, then define and prove correctness of simplification inside rules in order to introduce repeated recursion unfolding.

3.1 Rule Unfolding

For correct unfolding of rules in CHR, we follow the definition of [GMTW15], where also correctness of the unfolding function is shown. This means that we can safely add the unfolded rule to a program while preserving its semantics. In other words, a correctly unfolded rule is always redundant (but, of course, is expected to improve efficiency). In this paper we specialize the definition to the case of CHR simplification rules instead of arbitrary CHR rules. This simplifies the definition and is sufficient for our purposes.

For a goal \( A \), let \( \text{vars}(A) \) denote the set of variables in \( A \). A substitution is based on a mapping function from variables to terms \( \theta : V \rightarrow T \), written in postfix notation, such that domain of \( \theta \), the set \( \text{dom}(\theta) = \{ X \mid X\theta \neq X \} \) is finite. When a substitution is applied to a goal, it is applied to all variables in
the goal. If $A = B\theta$, where $B$ is a goal, we say that $A$ is an instance of $B$, $A$ matches $B$, and that $B$ is instantiated.

Set difference $C_1 = C_2 \setminus C_3$ for built-in constraints is defined as $C_1 = \{c \in C_2 \mid \not\models C_3 \rightarrow c\}$. In words, $C_1$ does not contain the constraints from $C_2$ that are implied by $C_3$.

**Definition 4 (Unfolding).** (Def. 8 [GMTW15]) Let $\mathcal{P}$ be a CHR program and let $r, v \in \mathcal{P}$ be two rules whose variables have been renamed apart

\[
\begin{align*}
  r &: H \leftrightarrow C \mid D \land B \land S \\
  v &: H' \leftrightarrow C' \mid B',
\end{align*}
\]

where $D$ is the conjunction of the built-in constraints in the body of $r$. Then we define

\[
\text{unfold}(r, v) = r'
\]

as follows. Let $\theta$ be a substitution such that $\text{dom}(\theta) \subseteq \text{vars}(H')$ and $\mathcal{C} \models (C \land D) \rightarrow S = H'\theta$, then the unfolded rule $r'$ is:

\[
\begin{align*}
  r' &: H \leftrightarrow C \land C'' \theta \mid D \land B \land S = H' \land B',
\end{align*}
\]

where $C'' \theta = C' \theta \setminus (C \land D)$ with $\text{vars}(C'' \theta) \cap \text{vars}(H' \theta) \subseteq \text{vars}(H)$ and $\mathcal{C} \models \exists(C \land C'' \theta)$.

If a goal $S$ in the body of rule $r$ matches the head $H'$ of a rule $v$, unfolding replaces $S$ in the body of rule $r$ by the body of rule $v$ together with $S = H'$ to obtain a new rule $r'$. In the resulting rule $r'$ we also add to its guard $C$ an instance of a part of the guard of rule $v$. This part $C''$ contains the non-redundant built-in constraints that are not already implied by the built-in constraints in the guard and body of the rule $r$.

Note that for a correct unfolding according to the above definition, three conditions have to be satisfied. First,

\[
\mathcal{C} \models (C \land D) \rightarrow S = H' \theta,
\]

means that goal $S$ must match the head $H'$, i.e. be an instance of $H'$, in the context of the built-in constraints of the rule $r$. This condition reflects the fact that the rule $v$ should be applicable to $S$.

Second, the non-obvious condition

\[
\text{vars}(C'' \theta) \cap \text{vars}(H' \theta) \subseteq \text{vars}(H)
\]

means that for correctness the variables shared between the instantiated head $H' \theta$ and instantiated simplified guard $C'' \theta$ of the rule $v$ must also occur in $H$. Note that using $C''$ instead of $C$ can make the set of variables smaller that have to occur in $H$.

Third, the satisfiability of the guard of the unfolded rule $r'$, $\mathcal{C} \models \exists(C \land C'' \theta)$, ensures that the rule is nontrivial in that it has a satisfiable guard. Otherwise the rule would never be applicable.
Example 2 (Summation, contd.). We unfold the recursive rule for summation with (a copy of) itself:

\[
\begin{align*}
  r & : \text{sum}(N, S) \Leftrightarrow N > 1 \mid S := N + S1, \text{sum}(N-1, S1) \\
  v & : \text{sum}(N', S') \Leftrightarrow N' > 1 \mid S' := N' + S1', \text{sum}(N'-1, S1').
\end{align*}
\]

Then the unfolded rule is

\[
\text{sum}(N, S) \Leftrightarrow N > 1, N-1 > 1 \mid S := N + S1, \text{sum}(N-1, S1) = \text{sum}(N', S'),
\]

\[
S' := N' + S1', \text{sum}(N'-1, S1').
\]

Unfolding is correct since its three conditions are satisfied. First, \(\text{sum}(N-1, S1)\) is an instance of \(\text{sum}(N', S')\), i.e.

\[
(N > 1, S := N + S1) \rightarrow \text{sum}(N-1, S1) = \text{sum}(N', S')\theta,
\]

since \(\theta\) can map \(N'\) to \(N-1\) and \(S'\) to \(S1\). Second,

\[
\text{vars}(N-1 > 1) \cap \text{vars}(\text{sum}(N-1, S1)) \subseteq \text{vars}(\text{sum}(N, S))
\]

holds since \(\{N\} \cap \{N, S1\} \subseteq \{N, S\}\). Third, the new guard \(N > 1, N-1 > 1\) is satisfiable.

Obviously we can simplify the built-in constraints of the guard and the body of this rule, and we will define this kind of simplification next.

### 3.2 Rule Simplification

Speedup crucially depends on the amount of constraint simplification that is possible in the unfolded rules. The goal is to replace constraints by semantically equivalent ones that can be executed more efficiently.

**Definition 5 (Simplification).** Given a rule \(r\) of the form

\[
r : H \Leftrightarrow C \mid D \land B,
\]

where \(D\) are the built-in constraints and \(B\) are the user-defined constraints in the body of the rule. We define

\[
\text{simplify}(r) = (H' \Leftrightarrow C' \mid D' \setminus C' \land B') \text{ such that}
\]

\[
(H' \land C) \equiv V (H' \land C') \text{ and } (C \land D \land B) \equiv V (D' \land B'),
\]

where \(D'\) are the built-in constraints and \(B'\) are the user-defined constraints in the body of the rule and where \(V = \text{vars}(H) \cup \text{vars}(H')\).

In the given rule, we replace head and guard, and the body, respectively, by simpler yet state equivalent goals. The choice of \(V\) allows us to remove local variables if possible, i.e. those that occur only in the guard or body of the rule. We temporarily add the guard \(C\) when we simplify the body to ensure correctness and improve the simplification.

For correctness we have to show that the same transitions \(S \mapsto T\) are possible with rule \(r\) and rule \(\text{simplify}(r)\).
The repeated unfolding of a recursive rule \( r \) with simplification is a sequence of rules \( r_0, r_1, \ldots, r_i, \ldots \) where
\[
 r_0 = r \in \mathcal{P} \\
 r_{i+1} = \text{simplify}(\text{unfold}(r_i))
\]

Let \( n \) be an upper bound on the number of recursive steps (recursion depth) for rule \( r \). The recursively unfolded program \( \mathcal{P}_{r,n} \) of rule \( r \) is defined as
\[
 \mathcal{P}_{r,n} = \mathcal{P} \cup \lfloor \log_2(n) \rfloor \bigcup_{i=1}^n r_i
\]

Example 4 (Summation, contd.). Recall the unfolded simplified rule
\[
\text{sum}(N,S) \iff N > 2 \land S := 2 \times N - 1 + S_1, \text{sum}(N-2,S_1)
\]
We repeat the unfolding:
\[
\text{sum}(N,S) \iff N > 2 \land N-2 > 2 \land S := 2 \times N - 1 + S_1, \text{sum}(N-4,S_1)
\]

The unfolded rule can be simplified into the rule
\[
\text{sum}(N,S) \iff N > 4 \land S := 4 \times N - 6 + S_1, \text{sum}(N-4,S_1)
\]

The complete program for repeated recursion unfolding sum three times is:
\[
\text{sum}(N,S) \iff N > 8 \land S := 8 \times N - 28 + S_1, \text{sum}(N-8,S_1)
\]

Note that for the query \( \text{sum}(9,R) \) we could use any of the recursive rules with the same result. Of course the most efficient way is to use the first rule. We will discuss optimal rule applications in the next section.

4 Time Complexity and Super-Linear Speedup

We first show that we can save on rule applications with our recursive unfolding scheme: we will always apply the most unfolded recursive rule to perform a maximum number of recursive steps with a minimum number of rule applications. Significant speedup then crucially depends on the amount of constraint simplification that is possible in the unfolded rules. Then we show that simplification leads to super-linear speedup in the best case, i.e. a change into a lower time complexity class.

Theorem 1 (Correctness of Rule Simplification). Let \( r = (H \iff C \land D \land B) \) be a rule and let \( s = (H' \iff C' \land D' \land B') \) be the simplified rule simplify\((r)\). For any state \( S \) and variables \( V, S \rightarrow_r T \text{ iff } S \rightarrow_s T \).

Proof. According to the definition of a CHR transition and simplify\((r)\), we know that
\[
 S \rightarrow_r T \text{ iff } S \equiv_V (H \land C \land G) \neq \text{false} \quad \text{and} \quad (C \land D \land B \land G) \equiv_V T \\\n S \rightarrow_s T \text{ iff } S \equiv_V (H' \land C' \land G') \neq \text{false} \quad \text{and} \quad (C' \land D' \land C' \land B' \land G') \equiv_V T \\
 (H \land C) \equiv_V (H' \land C') \\\n (C \land D \land B) \equiv_V (D' \land B')
\]

It suffices to show that \( S \rightarrow_r T \) implies \( S \rightarrow_s T \), since the implication in the other direction is symmetric and can be shown in the same way. Hence we have to show that there exists a goal \( G' \) such that
\[
 S \equiv (H \land C \land G) \equiv_V (H' \land C' \land G') \iff (H \land C) \equiv_V (H' \land C') \quad \text{and} \quad (C \land D \land B \land G) \equiv_V (C' \land D' \land C' \land B' \land G') \iff (C \land D \land B) \equiv_V (D' \land B').
\]
We choose \( G' = C \land G \). The full proof can be found in the full version of the paper that is online.

Example 3 (Summation, contd.). Recall the unfolded rule
\[
\text{sum}(N,S) \iff N > 1, N-1 > 1 \land S := N+S_1, \text{sum}(N-1,S_1) \equiv (N',S') \equiv (N'+S_1', \text{sum}(N'-1,S_1')).
\]
For the head and guard we have that
\[
\text{sum}(N,S), N > 1, N-1 > 1 \equiv_{\{S,N\}} \text{sum}(N,S), N > 2.
\]
For the body we have that
\[
N > 1, N-1 > 1, S := N+S_1, \text{sum}(N-1,S_1) \equiv (N',S'), S' := N'+S_1', \text{sum}(N'-1,S_1') \equiv_{\{S,N\}} N > 2, S := 2 \times N-1 + S_1', \text{sum}(N-2,S_1').
\]
Thus the rule can be simplified into the rule
\[
\text{sum}(N,S) \iff N > 2 \land S := 2 \times N-1 + S_1', \text{sum}(N-2,S_1').
\]

3.3 Repeated Recursion Unfolding

We now define what it means to unfold a given recursive rule with itself, to simplify it, to repeat this process, and to add the resulting rules up to a given bound to the original program.

Definition 6 (Repeated Recursion Unfolding). Let \( r \) be a recursive rule in a given program \( \mathcal{P} \). The unfolding of a recursive rule \( r \) is defined as
\[
\text{unfold}(r) = \text{unfold}(r,r)
\]
4.1 Optimal Rule Applications

By definition of correct unfolding, the unfolded rule is redundant with regard to the original rule. The other direction is not necessarily true. Even if the number of recursive steps should admit the application of an unfolded rule instead of the original rule, it may not be possible because the guard of the unfolded rule may be somewhat too strict.

Lemma 1 (Optimal Rule Applications).
Given a program \( P \) with a recursive rule \( r \). Assume that the unfolded rules in the unfolded program \( P_{r,n} \) satisfy the following condition: If a rule \( r_i \) with \( i < \log_2(n) \) can perform two recursive computation steps in a state, then rule \( r_{i+1} \) can perform one computation step.

Given a computation with the rule \( r \) that takes \( n \) recursion steps. Then there exists a computation with at most \( \log_2(n) \) recursive rule applications in the unfolded program \( P_{r,n} \) that results in an equivalent state. We call these optimal rule applications.

Proof.
The original recursive rule \( r \) performs one recursive step when applied. With each unfolding, the number of steps covered by the new unfolded rule is doubled. For rule \( r_i \), \( 2^i \) recursive steps are covered.

From the condition in the claim it follows that for a query with recursion depth \( n \), any unfolded rule \( r_i \) is applicable with \( n \geq 2^i \). Since the unfolding is correct, the resulting states will be equivalent. In one application step of rule \( r_i \), the recursion depth will be reduced to \( n-2^i \).

Then the optimal rule application strategy is to apply rule \( r_i \) of \( P_{r,n} \) such that \( 2^i+1 > n \geq 2^i \). We continue likewise with the resulting recursive goal until \( n = 0 \) and we have hit the base case of the recursion. As a consequence of this strategy, each of the rules \( r_j \) with \( 0 \leq j \leq i \) is applicable at most once because from \( 2^{j+1} > n \geq 2^j \) it follows that \( 2^j > n-j \geq 0 \). Since at most \( \log_2(n) \) rules have been generated by repeated unfolding, the above claim holds.

4.2 Super-Linear Speedup

With repeated unfolding up to \( \log_2(n) \) and optimal rule applications we are replacing \( n \) by \( \log_2(n) \) recursive calls. For a significant speedup, we also need simplification. To formalize the speedup, we need the following definition as a starting point.

Definition 7 (Worst-Case Runtime Bound).
For a given recursive rule \( r \), let \( c \) be a computable unary arithmetic function such that for a given goal with a recursion depth \( n \), \( c(n) \) is an upper bound on the runtime of the first recursive step with the original rule \( r \).

Note that by this definition, the runtime bound of the next recursive step will be \( c(n-1) \) and so on till \( c(0) \) which refers to the base case. We will ignore the runtime of base cases since their complexity is usually constant and since they are not affected by our transformations.

Example 4 (Summation, contd.). Recall the unfolded simplified rule \( \text{sum}(N, S) \Leftrightarrow N > 2 \mid S := 2N-1+S1, \text{sum}(N-2, S1) \).

We repeat the unfolding:

\[
\begin{align*}
\text{sum}(N, S) & \Leftrightarrow N > 2, N-2 > 2 \mid S := 2N-1+S1, \\
\text{sum}(N-2, S1) & = \text{sum}(N', S'), S' := 2N'-1+S1', \text{sum}(N'-2, S1').
\end{align*}
\]

The unfolded rule can be simplified into the rule

\[
\text{sum}(N, S) \Leftrightarrow N > 4 \mid S := 4N-6+S1', \text{sum}(N-4, S1').
\]

The complete program for repeated recursion unfolding \( \text{sum} \) three times is:

\[
\begin{align*}
\text{sum}(N, S) & \Leftrightarrow N > 8 \mid S := 8N-28+S1, \text{sum}(N-8, S1) \\
\text{sum}(N, S) & \Leftrightarrow N > 4 \mid S := 4N-6+S1, \text{sum}(N-4, S1) \\
\text{sum}(N, S) & \Leftrightarrow N > 2 \mid S := 2N-1+S1, \text{sum}(N-2, S1) \\
\text{sum}(N, S) & \Leftrightarrow N > 1 \mid S := N+S1, \text{sum}(N-1, S1) \\
\text{sum}(N, S) & \Leftrightarrow N = 1 \mid S = 1.
\end{align*}
\]

Note that for the query \( \text{sum}(9, R) \) we could use any of the recursive rules with the same result. Of course the most efficient way is to use the first rule. We will discuss optimal rule applications in the next section.

4 Time Complexity and Super-Linear Speedup

We first show that we can save on rule applications with our recursive unfolding scheme: we will always apply the most unfolded recursive rule to perform a maximum number of recursive steps with a minimum number of rule applications. Significant speedup then crucially depends on the amount of constraint simplification that is possible in the unfolded rules. Then we show that simplification leads to super-linear speedup in the best case, i.e. a change into a lower time complexity class.
4.1 Optimal Rule Applications

By definition of correct unfolding, the unfolded rule is redundant with regard to the original rule. The other direction is not necessarily true. Even if the number of recursive steps should admit the application of an unfolded rule instead of the original rule, it may not be possible because the guard of the unfolded rule may be somewhat too strict.

**Lemma 1 (Optimal Rule Applications).** Given a program \( P \) with a recursive rule \( r \). Assume that the unfolded rules in the unfolded program \( P^{r,n} \) satisfy the following condition: If a rule \( r_i \) with \( i < \log_2(n) \) can perform two recursive computation steps in a state, then rule \( r_{i+1} \) can perform one computation step.

Given a computation with the rule \( r \) that takes \( n \) recursion steps. Then there exists a computation with at most \( \log_2(n) \) recursive rule applications in the unfolded program \( P^{r,n} \) that results in an equivalent state. We call these optimal rule applications.

**Proof.** The original recursive rule \( r \) performs one recursive step when applied. With each unfolding, the number of steps covered by the new unfolded rule is doubled. For rule \( r_i \), \( 2^i \) recursive steps are covered.

From the condition in the claim it follows that for a query with recursion depth \( n \), any unfolded rule \( r_i \) is applicable with \( n \geq 2^i \). Since the unfolding is correct, the resulting states will be equivalent. In one application step of rule \( r_i \), the recursion depth will be reduced to \( n - 2^i \).

Then the optimal rule application strategy is to apply rule \( r_i \) of \( P^{r,n} \) such that \( 2^{i+1} > n \geq 2^i \). We continue likewise with the resulting recursive goal until \( n = 0 \) and we have hit the base case of the recursion. As a consequence of this strategy, each of the rules \( r_j \) with \( 0 \leq j \leq i \) is applicable at most once because from \( 2^{i+1} > n_j \geq 2^j \) it follows that \( 2^j > n_j - 2^j \geq 0 \). Since at most \( \log_2(n) \) rules have been generated by repeated unfolding, the above claim holds. \( \square \)

4.2 Super-Linear Speedup

With repeated unfolding up to \( \log_2(n) \) and optimal rule applications we are replacing \( n \) by \( \log_2(n) \) recursive calls. For a significant speedup, we also need simplification. To formalize the speedup, we need the following definition as a starting point.

**Definition 7 (Worst-Case Runtime Bound).** For a given recursive rule \( r \), let \( c \) be a computable unary arithmetic function such that for a given goal with a recursion depth \( n \), \( c(n) \) is an upper bound on the runtime of the first recursive step with the original rule \( r \).

Note that by this definition, the runtime bound of the next recursive step will be \( c(n-1) \) and so on till \( c(0) \) which refers to the base case. We will ignore the runtime of base cases since their complexity is usually constant and since they are not affected by our transformations.
A significant speedup can be achieved if we can simplify the built-in constraints in the unfolded rules so that their runtime does not double with each unfolding. In the best case, unfolding does not increase the runtime bound of a recursive step: instead of a time bound of $c(n) + c(n)$ in the unfolded rule, we still have a bound of $c(n)$ with simplified constraints. We call this best-case simplification. In the following theorem we show how this speedup can change the complexity class into a lower one.

**Theorem 2 (Super-Linear Speedup of repeated recursion Unfolding).**
Given an unfolded program $P^{r,n}$ with optimal rule applications. We assume best-case simplification. Let the function $c(n)$ compute a time bound for the first recursive step with any recursive rule $r_i$ with $i \leq \log_2(n)$ for any query with recursion depth $n$. Then the time complexity classes\(^1\) for the original and unfolded recursion are according to Table 1.

<table>
<thead>
<tr>
<th>Time Complexity Class</th>
<th>Recursive Step $c(n)$</th>
<th>Recursion $r(n)$</th>
<th>Unfolded $r'(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(poly)logarithmic, constant $k \geq 0$</td>
<td>$\log_2(n)^k$</td>
<td>$n \log_2(n)^k$</td>
<td>$\log_2(n)^{k+1}$</td>
</tr>
<tr>
<td>polynomial, linear $k \geq 1$</td>
<td>$n^k$</td>
<td>$n^{k+1}$</td>
<td>$2n^k$</td>
</tr>
<tr>
<td>exponential</td>
<td>$2^n$</td>
<td>$2^{n+1}$</td>
<td>$2^{n+1}$</td>
</tr>
</tbody>
</table>

Fig. 1. Speedup in terms of time complexity classes with recursion length $n$

**Proof.** Then the runtime for the recursive part of the computation $r(n)$ with $n \geq 1$ is clearly bounded by $nc(n)$. The time bound can be more precisely modeled by a recurrence relation\(^2\) of the form

$$r(n) = c(n) + r(n - 1).$$

With optimal rule applications in the unfolded recursion according to Lemma 1, the remaining number of recursive steps to be performed is at least halved going from $n$ to $n - 2^i$: from the condition for optimal rule application $2^{i+1} > n \geq 2^i$ it follows that $2^i > n/2$, $n - 2^i < n - n/2$ and thus $n - 2^i < n/2$. We can therefore model the time bound by the recurrence relation

$$r'(n) = c(n) + r'(n/2).$$

Now the results in Table 1 can be proven by solving the recurrences. They can be verified by inserting the solutions. For upper bounds it suffices to show that the left hand side is at least as large as the right hand side of the recurrence\(^3\).

\(^1\) We use constant factors in the complexity classes to emphasize that the actual runtime is expected to adhere to the constant factor.

\(^2\) A recurrence (relation) is an equation that recursively defines a function. A closed-form solution of a recurrence is a finitary mathematical expression.
relation for \( n \geq 2 \). For example, let \( c(n) = \log_2(n) \) then

\[
\begin{align*}
\log_2(n) &= n \log_2(n) = \log_2(n) + (n-1)\log_2(n) \\
\log_2(n) + (n-1)\log_2(n-1) &= c(n) + r(n-1) \quad \text{and} \\
\log_2(n) + (\log_2(n-1))^2 &= \log_2(n) + \log_2(n/2)^2 = c(n) + r'(n/2). \quad \square
\end{align*}
\]

For linear and polynomial worst-case time complexity classes a super-linear speedup by the factor \( O(n/2) \) is possible, for (poly)logarithmic or constant complexity classes a speedup of \( O(n/\log_2(n)) \), while for exponential complexity classes no improvement of the complexity class is possible (but the unfolded recursion will still run faster).

5 Recursionless Recursion

We show how we can guarantee optimal rule applications in practice in an implementation of a rule-based language. Besides relying on rule order, we present a semi-naive approach in this paper that has some overhead\(^3\). One advantage is that this transformation completely eliminates recursion up to a chosen recursion depth. It is therefore well-suited for hardware synthesis [TORF12].

5.1 Enabling Optimal Rule Applications

We transform the recursion away completely based on the observation that in an optimal rule application, each unfolded recursive rule \( r_i \) will be applied at most once only. We assume that rules are tried in the order in which they appear in the program as stipulated by the refined semantics of CHR.

**Definition 8 (Recursionless Recursion).** For each \( i \leq \log_2(n) \) we replace the rule

\[
r_i = H \leftrightarrow C | D, B, R,
\]

where \( R \) denotes the recursive call in the body, by the pair of rules

\[
r_i : H_i \leftrightarrow C | D, B, R_{i-1}
\]

\[
r'_i : H_i^X \leftrightarrow R_{i-1}^X
\]

where for \( H, H_i \) denotes the recursive constraint \( H \) whose constraint symbol \( c \) has been renamed to \( c_i \), except for \( H_{-1} \) which is just \( H \), and where \( H^X \) denotes a user-defined constraint whose arguments are the same sequence of distinct variables given in \( X \). The same syntax applies to \( R \).

\(^3\) Optimal complexity can be achieved with indexing on the recursion depth.
Note that by this construction, each rule is applicable at most once, because there is no recursion left. For each recursion level \( i \), relying on rule order, first rule \( r_i \) will be tried. It is either applied or otherwise rule \( r'_i \) is applicable. In \( r'_i \) we just pass the argument parameters down to the next lower rule level. So we either we apply rule \( r_i \) with no additional overhead or we apply the very simple rule \( r'_i \) which has constant runtime plus the overhead of trying rule \( r_i \) before.

**Example 5 (Summation, contd.).** The recursionless version of our running example is

\[
\begin{align*}
    r_2 & : \text{sum}_2(N, S) \iff N > 4 \mid S := 4 \cdot N - 6 + S1, \text{sum}_1(N-4, S1) \\
    r'_2 & : \text{sum}_2(N, S) \iff \text{sum}_1(N, S) \\
    r_1 & : \text{sum}_1(N, S) \iff N > 2 \mid S := 2N - 1 + S1, \text{sum}_0(N-2, S1) \\
    r'_1 & : \text{sum}_1(N, S) \iff \text{sum}_0(N, S) \\
    r_0 & : \text{sum}_0(N, S) \iff N > 1 \mid S := N + S1, \text{sum}(N-1, S1) \\
    r'_0 & : \text{sum}_0(N, S) \iff \text{sum}(N, S) \\
    \text{sum}(1, S) & \iff S = 1.
\end{align*}
\]

**5.2 Speedup with Worst-Case Overhead**

Rule application attempts that do not lead to rule applications cause an overhead. The overhead depends on the number of unfolded rules and the cost of trying them. In the best case, the cost for head matching attempts and guard checking is constant and the resulting overhead is neglectable. For the worst case we can assume that rule application attempts cost as much time as actual rule applications. This leads to the following lemma. It shows that even in the worst case super-linear speedup is still obtained for most recursion depths.

**Lemma 2 (Speedup with Worst-Case Overhead).** Let \( N > n \). Assume we have repeatedly unfolded for recursion depth \( N \), i.e. up to \( \log_2(N) \). We have a query with recursion depth \( n \) and complexities \( c(n) \), \( r(n) \) and \( r'(N) \) as calculated in Theorem 2. Then Table 2 shows the upper bounds on the worst-case overhead for the corresponding complexity classes.

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Recursion</th>
<th>Unfolded</th>
<th>Just Rule Order</th>
<th>Recursionless</th>
</tr>
</thead>
<tbody>
<tr>
<td>(poly)logarithmic, ( c. k \geq 0 )</td>
<td>( n \log_2(n)^k )</td>
<td>( \log_2(n)^{k+1} )</td>
<td>( \log_2(n) \log_2(n)^{k+1} )</td>
<td>( 2 \log_2(N) \log_2(n)^{k+1} )</td>
</tr>
<tr>
<td>polynomial, linear ( k \geq 1 )</td>
<td>( \log_2(n)^{k+1} )</td>
<td>( 2n^k )</td>
<td>( 2 \log_2(N) n^k )</td>
<td>( (\log_2(N) + 2)n^k )</td>
</tr>
<tr>
<td>exponential</td>
<td>( \log_2(n)^{k+1} )</td>
<td>( 2^{n+k} )</td>
<td>( \log_2(N) 2^{n+k} )</td>
<td>( (\log_2(N) + 2)2^n )</td>
</tr>
</tbody>
</table>

**Fig. 2.** Speedup with worst-case overhead when unfolded up to \( \log_2(N) \)

**Proof.** When we use rule order alone, up to \( \log_2(N) \) rules will be tried in vain before a rule will be applied. This happens for each recursive call. For the
first recursive call this incurs a runtime of \( c(n) \) with each attempt in the worst case or each application, for the second of \( c(n/2) \) and so on. We already know that the sum of these runtimes is \( r'(n) \). So instead of the runtime \( r'(n) \) we have a worst-case time bound of

\[
\log_2(N)r'(n).
\]

With recursionless recursion we try each rule at most once. With rule order the handling of the additional rules \( r'_i \) at each recursion level takes constant time and we can ignore them in the following. First we try \( \log_2(N) - \log_2(n) \) unfolded rules \( r_i \) in vain before we reach the rule corresponding to level \( \log_2(n) \). From then on, the worst case is the application of all rules below. So overall, \( \log_2(N) - \log_2(n) \) rules are tried once each with runtime \( c(n) \) and then a runtime of \( r'(n) \) is incurred, resulting in a bound

\[
(\log_2(N) - \log_2(n))c(n) + r'(n).
\]

Using these formulas with the complexities for \( c(n) \), \( r(n) \) and \( r'(N) \) from Theorem 2 gives the upper bounds that are listed in somewhat simplified form in Table 2.

From Table 2 we can see that the worst case overhead is a factor of \( O(\log_2(N)) \) for rule order alone. For recursionless recursion, it is a worst case factor of \( O(2\log_2(N)/\log_2(n)) \) for polylogarithmic complexity classes, and a factor of \( O(\log_2(N)/2) \) for polynomial and exponential complexity.

In this worst case of recursionless recursion, all complexity classes but exponential complexity are still lower than the complexity classes of the original recursions as long as \( n \) is larger than \( \log_2(N) \).

**Unbounded Recursion.** With recursionless recursion, the recursion depth is bounded by \( N \). We can simply eliminate this bound by re-introducing recursion for the most unfolded rule \( r_k \) with \( k = \lfloor \log_2(N) \rfloor \):

\[
r_k = H_k \iff C \mid D, B, R_k.
\]

For \( N \leq n \) we now apply rule \( r_k \) as long as possible. Of course, recursionless recursion is now a misnomer.

## 6 Examples with Benchmarks

We have implemented a simple prototype for performing repeated recursion unfolding in CHR and Prolog relying on rule order and recursionless recursion. The source code of the resulting example programs is listed in the full version of the paper that is online. In our experiments, we used the CHR system in SWI Prolog Version 6.2.1 running on an Apple Mac mini 2018 with Intel Core i5 8GB RAM and OS-X 10.14.6. In our benchmarks, unfolding is up to the level given by \( \log_2(N) \) with \( N > n \).
6.1 Benchmarks for Summation Example

All recursive rules of \textit{sum} have a constant time for matching the head, checking the guard, computing the sum and doing the recursive call including the subtraction. The recursion depth is determined by the input number \(n\).

\textbf{Super-Linear Speedup.} Table 3 lists our benchmark results for repeated unfolding up to \(\log_2(N) = 25\) and input numbers \(n\) from 1024 to 8192. Times are in milliseconds. Summation has been performed once for all numbers in the given range. The given timings are the sum of these execution times. The unfolded versions showed constant timings for the number ranges over the complete set of numbers, with recursionless recursion performing about 200 summations per millisecond.

The row \textit{Original} shows the linear runtime of the given recursive summation. Row \textit{Rule Order} refers to the unfolded recursion of \textit{sum} relying on rule order alone, which already shows constant time and at least a ten-fold speedup. \textit{Recursionless} refers to the recursionless recursion, it is a factor of four faster than the unfolded recursion, again showing constant time. By comparison, the Prolog implementation of the original rules using the cut operator for efficiency shows again the linear time behavior.

According to the Speedup Theorem 2 and Lemma 2, for the original recursion we expect linear time complexity \(O(n)\) and for the unfolded program with rule order complexity \(O(\log_2(N)\log_2(n))\). With recursionless recursion, we expect a complexity of \(O(\log_2(N))\). Since \(N\) is fixed in our benchmarks, this amounts to a constant time.

We attribute the better-than-estimated constant time behavior of the unfolded recursion with rule order to the strong variation in rule attempts and applications with subsequent numbers. For example, while the number 1025 requires just one recursive step with rule \(r_{10}\), the previous number 1024 needs all smaller recursive steps from \(r_9\) down to \(r_0\). This behavior leads to constant runtime when we sum over ranges of numbers as in Table 3.

<table>
<thead>
<tr>
<th>Input n</th>
<th>1024-2047</th>
<th>2048-3071</th>
<th>3072-4095</th>
<th>4096-5119</th>
<th>5120-6143</th>
<th>6144-7167</th>
<th>7168-8192</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>240</td>
<td>481</td>
<td>839</td>
<td>1118</td>
<td>1507</td>
<td>1832</td>
<td>2258</td>
</tr>
<tr>
<td>Rule Order</td>
<td>19</td>
<td>20</td>
<td>20</td>
<td>21</td>
<td>20</td>
<td>22</td>
<td>21</td>
</tr>
<tr>
<td>Recursion</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Prolog</td>
<td>185</td>
<td>369</td>
<td>663</td>
<td>873</td>
<td>1207</td>
<td>1457</td>
<td>1816</td>
</tr>
</tbody>
</table>

\textbf{Fig. 3.} Benchmarks of Summation
6.2 Complete Example List Reversal

This classical program reverses a given list in a naive way. The constraint \( r(A, B) \) holds if list \( B \) is the reversal of list \( A \). We use Prolog notation for lists.

\[
\begin{align*}
   r([C|A], D) & \Leftrightarrow r(A, B), a(B, [C], D) \\
   r([], D) & \Leftrightarrow D = [].
\end{align*}
\]

The built-in constraint \( a(X, Y, Z) \) appends (concatenates) two lists \( X \) and \( Y \) into a third list \( Z \).

**Repeated Recursion Unfolding.** We try to unfold the recursive rule with a copy of itself:

\[
\begin{align*}
   r([C|A], D) & \Leftrightarrow r(A, B), a(B, [C], D) \\
   r([C'|A'], D') & \Leftrightarrow r(A', B'), a(B', [C'], D').
\end{align*}
\]

However the recursive call in the original rule \( r(A, B) \) is not an instance of the head \( r([C'|A'], D') \) of the copy of the rule. This is a mere technicality. We move \([C'|A']\) into the guard and replace it by a new variable \( A''\):

\[
\begin{align*}
   r(A'', D') & \Leftrightarrow A'' = [C'|A'] | r(A', B'), a(B', [C'], D').
\end{align*}
\]

This does not change the semantics of the rule because

\[
\begin{align*}
   r([C'|A'], D') & \equiv_{\{C', A', D'\}} r(A'', D'), A'' = [C'|A'].
\end{align*}
\]

The unfolding is now correct with \( A'' \) substituted by \( A \) in the guard

\[
\begin{align*}
   r([C|A], D) & \Leftrightarrow A = [C'|A'] | r(A, B) = r(A'', D'), r(A', B'), \\
   & \quad a(B', [C'], D'), a(B, [C], D).
\end{align*}
\]

Next we use the guard equality \( A = [C'|A'] \) to substitute the variable \( A \) in the head \( r([C|A], D) \) back to \([C'|A']\). This is correct since

\[
\begin{align*}
   r([C|A], D), A = [C'|A'] \equiv_{\{C, A, D, C', A'\}} r([C, C'|A'], D), A = [C'|A'].
\end{align*}
\]

Now we proceed with the simplification for unfolded rules as defined. For the head and guard we simplify

\[
\begin{align*}
   r([C, C'|A'], D), A = [C'|A'] \equiv_{\{C, D, C', A'\}} r([C, C'|A'], D).
\end{align*}
\]

For the body we simplify

\[
\begin{align*}
   \quad A = [C'|A'], r(A, B) = r(A'', D'), r(A', B'), a(B', [C'], D'), a(B, [C], D) \equiv_{\{C, D, C', A'\}} r(A', B'), a(B', [C'], D'), a(B, [C], D) \equiv_{\{C, D, C', A'\}} r(A', B'), a(B', [C'], D'), a(B, [C], D).
\end{align*}
\]

The eureka moment for best-case simplification is that we can merge the two calls to constraint \( a \) into one if we concatenate their known second arguments.
As a result the unfolded rule is simplified into the rule
\[ r([C, C'|A'], D) \leftrightarrow r(A', B'), a(B', [C', C], D). \]

Repeated recursion unfolding thrice results in the following simplified rules
\[
\begin{align*}
  r([J, I, H, G, F, E, D, C|A], K) & \leftrightarrow r(A, B), a(B, [C, D, E, F, G, H, I, J], K) \\
  r([F, E, D, C|A], G) & \leftrightarrow r(A, B), a(B, [C, D, E, F], G) \\
  r([D, C|A], E) & \leftrightarrow r(A, B), a(B, [C, D], E) \\
  r([C|A], D) & \leftrightarrow r(A, B), a(B, [C], D) \\
  r([], D) & \leftrightarrow D = [].
\end{align*}
\]

Note that we see here a worst case of program size increase. With each unfolding, the rule size doubles. Still, this does not lead to code explosion since we unfold only up to \(\log(n)\) times and therefore the size of all rules added is proportional to \(n\).

**Time Complexity.** Consider the original recursive rule
\[ r([C|A], D) \leftrightarrow r(A, B), a(B, [C], D). \]
Its complexity is clearly related to the length of the list in the first argument of constraint \(r\), i.e. the number of elements in the list. Assume the length of the list is \(n\). The runtime of constraint \(a\) is proportional to the length of the list in the first argument, \(B\). The length of \(B\) is the same as the length of \(A\) due to \(r(A, B)\).

<table>
<thead>
<tr>
<th>List Lengths</th>
<th>Original</th>
<th>Hand-Optimized</th>
<th>Rule Order</th>
<th>Recursionless</th>
<th>Prolog Built-In</th>
</tr>
</thead>
<tbody>
<tr>
<td>128-383</td>
<td>496.0</td>
<td>7.6</td>
<td>5.7</td>
<td>3.6</td>
<td>4.2</td>
</tr>
<tr>
<td>384-639</td>
<td>1847.0</td>
<td>15.4</td>
<td>8.0</td>
<td>5.8</td>
<td>8.6</td>
</tr>
<tr>
<td>640-895</td>
<td>4442.0</td>
<td>22.8</td>
<td>11.0</td>
<td>8.6</td>
<td>12.7</td>
</tr>
<tr>
<td>896-1151</td>
<td>8630.0</td>
<td>30.2</td>
<td>12.4</td>
<td>9.3</td>
<td>16.6</td>
</tr>
<tr>
<td>1152-1407</td>
<td>14696.0</td>
<td>37.4</td>
<td>11.8</td>
<td>9.1</td>
<td>20.9</td>
</tr>
<tr>
<td>1408-1663</td>
<td>23445.0</td>
<td>45.1</td>
<td>16.8</td>
<td>13.2</td>
<td>24.7</td>
</tr>
<tr>
<td>1664-1929</td>
<td>35487.0</td>
<td>52.6</td>
<td>22.1</td>
<td>17.7</td>
<td>29.3</td>
</tr>
</tbody>
</table>

**Fig. 4.** Benchmarks of List Reversal

**Super-Linear Speedup.** Repeated recursion unfolding was performed up to \(\log_2(N) = 10\). The benchmarks in Table 4 cover all possible list lengths from 128 to 1929. Times are in milliseconds. They give the sum of runtimes for each of the 256 list lengths in the given range of numbers.

The row *Original* stands for the original recursion implementing naive list reversal. Row *Hand-Optimized* stands for the linear time non-naive list reversal written in CHR. Row *Rule Order* stands for the repeated unfolding of naive list reversal relying on rule order alone. Row *Recursionless* stands for the repeated
unfolding of naive list reversal were recursion is completely eliminated. Row Prolog Built-In stands for the built-in list reversal of SWI Prolog.

According to our Speedup Theorem 2 and Lemma 2, we expect quadratic behavior for the original recursion and linear behavior for the unfolded recursion, were the recursionless recursion should be twice as fast. This actually holds as the benchmarks show.

All timings show linear time behavior except the original list reversal without unfolding. The fastest program version with recursionless recursion reverses about 100 list elements per millisecond. The transformed code executed twice as fast as the hand-coded well-known linear version of list reversal that does not use list concatenation. Surprisingly, our recursionless recursion proved even faster than the built-in optimized reverse/2 of SWI Prolog.

Note that the runtimes around 1024 almost stay constant. Our most unfolded rule with \( i = N = 10 \) handles a list with \( 2^{10} = 1024 \) elements. Again we attribute this averaging effect of considering ranges of inputs. For example, a list of length 1024 needs one rule application with rule \( r_{10} \) at recursion level \( i = 10 \) before the base case is reached, while with 1023 we need to apply all rules for each level \( i < 10 \).

7 Related Work

Program transformation to improve efficiency is usually concerned with a strategy for the combination of unfolding and folding techniques to replace code starting with [BD77]. In the literature, recursion is usually unfolded several times together with the base case and then simplified. We rely solely on unfolding the recursive part and repeat the process. We ignore the base case. We add redundant rules this way but never remove any.

Program transformation as a tool for specific aims and applications is abundant in CHR, for an overview see [SVWSDK10,Frü15]. General methods for unfolding [GMTW15] exist (which we have used in this paper), for specializing rules with regard to a specific given query [Frü05], and for optimization induced by confluence [AF04].

In [Ant91] unfolding-based meta-level systems for Prolog consist of a hierarchy of additional meta-rules and a hierarchical execution scheme, i.e. a change in semantics. These rules are described as shortcuts obtained from unfold/fold operations. Presumably, when adapted to Prolog, the rules generated by repeated recursion unfolding could be considered as meta-rules in this generic framework.

Directly related literature is sparse. One reason could be that our approach is based on generating and keeping redundant rules which seems counter-intuitive at first. Moreover, super-linear speedups are a rare feat and mostly concern parallel programs, while our approach applies to sequential programs. An exception is work based on supercompilation. In this program transformation, generalisation while unfolding increases the chance for folding. Using advanced generalisation techniques for supercompilation, distillation [Ham09] and equality indices [GKN16] can achieve super-linear speedup on some examples. In contrast, our
approach is straightforward, as it does not involve generalisation or folding, and is applied to a programming language in practical use.

8 Conclusions and Future Work

Repeated recursion unfolding is a new approach that repeatedly unfolds a recursion with itself and simplifies it while keeping all unfolded rules. We proved its correctness.

We showed there exists an optimal rule application strategy such that significant speedups are possible. We proved a super-linear speedup theorem in case of best-case simplification. Our program transformation lowers the time complexity class of a recursion for polylogarithmic by a factor of \( O(n)/\log_2(n) \) and polynomial classes by a factor of \( O(n) \).

For a semi-naive implementation of repeated recursion unfolding, recursion-less recursion, we proved an overhead which is linear in the number of unfolded rules. Super-linear speedup is still obtained in most cases. The actual runtime improvement quickly reaches several orders of magnitude. Our approach improves the time complexity from linear to constant for the summation example and from quadratic to linear for list reversal. The latter runs faster than a built-in hand-optimized version.

While our speedups are super-linear, it is too early to tell which recursive programs allow for the necessary best-case simplification. A good simplification requires some insight and thus in general cannot be found automatically.

We defined and implemented repeated recursion unfolding using the rule-based language CHR (Constraint Handling Rules), but we think our approach can be applied to mainstream programming languages and hardware synthesis as well.

**Ongoing and future work.** To extend our approach to mutual recursion as well as multiple recursive rules should be straightforward. Indeed, we already could derive a novel double recursive linear-time algorithm from a naive exponential-time double recursion for Fibonacci numbers.

In this paper, the super-linear speedup is bounded in the sense that it holds up to a chosen upper bound on the number of recursive steps. In ongoing work, we have succeeded for the examples in this paper to extend our approach to run-time dynamic on-the-fly just-in-time repeated recursion unfolding so that the super-linear speedup is unbounded, i.e. independent of the recursion depth.

Optimal complexity can be achieved with indexing on the recursion depth which we are currently implementing. We already have implemented indexing for the examples in this paper.

One could also transfer our approach from recursion to loop constructs in other programming languages. We have already done examples for Java.

**Acknowledgements.** Part of this research work was performed during the sabbatical of the author in summer semester 2020. We thank the anonymous reviewers for their skepticism which helped to clarify the contribution of the paper.
A Formal Model for a Linear Time Correctness Condition of Proof Nets of Multiplicative Linear Logic

Satoshi Matsuoka
National Institute of Advanced Industrial Science and Technology (AIST),
1-1-1 Umezono, Tsukuba, Ibaraki, 305-8565 Japan
matsuoka@ni.aist.go.jp

Abstract.
In a previous paper, we have reported a new linear time correctness condition for proof nets of Multiplicative Linear Logic without units, where we gave a description of the algorithm in an informal way. In this paper, we give a formal model for the algorithm. Our formal model is based on a finite state transition system with queues as well as union-find trees as data structures. The model has been obtained by trial and error based on a concrete implementation of the algorithm. In addition, the algorithm has a subtle mechanism in order to avoid deadlock. We give an invariant property of the state transition system and it guarantees the deadlock-freedom.

1 Introduction

More than three decades ago, J.Y. Girard introduced the notion of proof nets of unit free Multiplicative Linear Logic (for short, MLL)[3]. It is a parallel syntax for MLL proofs, removing redundancy of sequent calculus proofs. In [3], he introduced MLL proof structures, which are graphs whose nodes are labeled by MLL formulas and then defined MLL proof nets as sequentializable MLL proof structures. Moreover he introduced a topological property called the long trip condition for MLL proof structures and showed that an MLL proof structure is an MLL proof net if and only if it satisfies the long trip condition. Such a characterization is called a correctness condition for MLL proof nets. Since then many other correctness conditions have been given for MLL and its variants or extensions by many researchers.

In [7], the author gives a new linear time correctness condition algorithm for MLL. This means that using the algorithm, we can check whether or not an MLL proof structure is an MLL proof net in linear time. The description of the algorithm in [7] is given in an informal way. Such an informal description is important because an algorithm supposed to be important should be understood by humans. In addition, an algorithm supposed to be important also should be understood by a machine and proved that it is correct easily: If an algorithm turned out to be difficult to implement, then its interest would be limited to theoretical one.

References

A Formal Model for a Linear Time Correctness Condition of Proof Nets of Multiplicative Linear Logic

Satoshi Matsuoka
National Institute of Advanced Industrial Science and Technology (AIST),
1-1-1 Umezono, Tsukuba, Ibaraki, 305-8565 Japan
matsuoka@ni.aist.go.jp

Abstract. In a previous paper, we have reported a new linear time correctness condition for proof nets of Multiplicative Linear Logic without units, where we gave a description of the algorithm in an informal way. In this paper, we give a formal model for the algorithm. Our formal model is based on a finite state transition system with queues as well as union-find trees as data structures. The model has been obtained by trial and error based on a concrete implementation of the algorithm. In addition, the algorithm has a subtle mechanism in order to avoid deadlock. We give an invariant property of the state transition system and it guarantees the deadlock-freedom.

1 Introduction

More than three decades ago, J.Y. Girard introduced the notion of proof nets of unit free Multiplicative Linear Logic (for short, MLL)[3]. It is a parallel syntax for MLL proofs, removing redundancy of sequent calculus proofs. In [3], he introduced MLL proof structures, which are graphs whose nodes are labeled by MLL formulas and then defined MLL proof nets as sequentializable MLL proof structures. Moreover he introduced a topological property called the long trip condition for MLL proof structures and showed that an MLL proof structure is an MLL proof net if and only if it satisfies the long trip condition. Such a characterization is called a correctness condition for MLL proof nets. Since then many other correctness conditions have been given for MLL and its variants or extensions by many researchers.

In [7], the author gives a new linear time correctness condition algorithm for MLL. This means that using the algorithm, we can check whether or not an MLL proof structure is an MLL proof net in linear time. The description of the algorithm in [7] is given in an informal way. Such an informal description is important because an algorithm supposed to be important should be understood by humans. In addition, an algorithm supposed to be important also should be understood by a machine and proved that it is correct easily: If an algorithm turned out to be difficult to implement, then its interest would be limited to theoretical one.
In this paper, we give a formal model for the algorithm. The formal model is obtained based on the implementation [8] of the algorithm by trial and error. Our model is presented as a finite state transition system. A notable point of the model is that a deadlock prevention mechanism in the algorithm is incorporated in order to guarantee the correctness of the algorithm and the deadlock-freedom is formalized as an invariant in the transition system.

Our implementation of the algorithm in [8] corresponding to the formal model is much faster than a naive quadratic algorithm, especially for bigger proof structures. It is an efficient implementation for a rewriting system and effectively exploits union-find data structures. Although so far several researchers have used union-find data structures in order to derive efficient algorithms in the context of logic-oriented computer science [11, 5, 9, 4], union-find data structures have not been used for concrete implementations in the community in many cases. We believe that there are other places in this research area that can exploit union-find data structures effectively because they provide a method that implements various equivalence relations. One instance is given in [6]. We hope that our work is helpful for promoting the use of union-find data structures.

2 Multiplicative Linear Logic, Proof Structures and Proof Nets

2.1 Multiplicative Linear Logic

We introduce the system of Multiplicative Linear Logic (for short MLL). We define MLL formulas, which are denoted by \( F, G, H, \ldots \), by the following grammar:

\[
F ::= p \mid p^\perp \mid F \otimes G \mid F \& G
\]

The negation of \( F \), which is denoted by \( F^\perp \) is defined as follows:

\[
(p)^\perp = p^\perp \\
(p^\perp)^\perp = p \\
(F \otimes G)^\perp = G^\perp \otimes F^\perp \\
(F \& G)^\perp = G^\perp \& F^\perp
\]

The formula \( p \) is called an atomic formula. In this paper, we only consider the logical system with only one atomic formula: We can reduce the correctness condition with many atomic formulas to this simplified case by forgetting the information. We denote multisets of MLL formulas by \( \Lambda, \Lambda_1, \Lambda_2, \ldots \). An MLL sequent is a multiset of MLL formulas \( \Lambda \). We write an MLL sequent \( \Lambda \) as \( \vdash \Lambda \). The inference rules of MLL are as follows:

\[
\begin{align*}
\text{ID} & \quad \vdash p^\perp, p \\
\otimes \quad \vdash A_1, F & \quad \vdash A_2, G & \quad \otimes \vdash A, F, G
\end{align*}
\]
We note that we restrict the ID-axiom to that with only atomic formula $p$ and its negation $p^\perp$. We omit the cut rule that has the form
\[
\text{Cut} \quad \vdash A_1, F \quad \vdash A_2, F^\perp \\
\Downarrow \quad \vdash A_1, A_2
\]
because it can be identified with the $\otimes$-rule for our purpose.

### 2.2 MLL Proof Nets

Next we introduce MLL proof nets. Figure 1 shows the MLL links we use. Each MLL link has a few MLL formulas. Such an MLL formula is a conclusion or a premise of the MLL link, which is specified as follows:

1. In an ID-link, each of $p$ and $p^\perp$ is called a conclusion of the link.
2. In a $\otimes$-link, each of $F$ and $G$ is called a premise of the link and $F \otimes G$ is called a conclusion of the link.
3. In a $\ominus$-link, each of $F$ and $G$ is called a premise of the link and $F \ominus G$ is called a conclusion of the link.

![Fig. 1. MLL Links](image)

An MLL proof structure $\Theta$ is a set of MLL links that satisfies the following conditions:

1. For each link $L$ in $\Theta$, each conclusion of $L$ can be a premise of at most one link other than $L$ in $\Theta$.
2. For each link $L$ in $\Theta$, each premise of $L$ must be a conclusion of exactly one link other than $L$ in $\Theta$.

A formula occurrence $F$ in an MLL proof structure $\Theta$ is a conclusion of $\Theta$ if $F$ is not a premise of any link in $\Theta$.

An MLL proof net is an MLL proof structure that is constructed by the rules in Figure 2. Note that each rule in Figure 2 has the corresponding inference rule in the MLL sequent calculus. All MLL proof structure are not necessarily an MLL proof net.

### 3 The Rewriting System over deNM-Trees

In this section we introduce our rewriting system. Then we give our correctness condition based on the system.
Fig. 2. Definition of MLL Proof Nets

3.1 deNM-Trees

First we define deNM-trees, which are inspired from de Naurois and Mogbil’s correctness condition [10]. In the following we fix an MLL proof structure \( \Theta \).

**Definition 1 (deNM-trees).** A deNM-tree is a finite tree consisting of labeled nodes and \( \otimes \)-nodes:

- A labeled node is labeled by a switch-label set \( S \) that is a subset of \( S_{\text{full}} = \{ l_1, r_1, \ldots, l_j, r_j \} \), where each \( L_j (1 \leq j \leq \ell) \) is a \( \otimes \)-link in \( \Theta \). The degree \( t \) of a labeled node is at most the number of nodes of the deNM-tree. See Figure 3.
- A \( \otimes \)-node is a labeled by a \( \otimes \)-link \( L \) in \( \Theta \). The degree of a \( \otimes \)-node is 1 or 2. See Figure 4. As shown symbolically, we distinguish the port above of a \( \otimes \)-node from the port below.

Next we give a translation from \( \Theta \) to a deNM-tree. The translation is slightly different from that given in [7]: it is suitable for a mechanical implementation. For that purpose, we have to make some preparations. We note that each formula occurrence in \( \Theta \) is (1) a literal \( p \) or \( p^\perp \), (2) a \( \otimes \)-formula \( A \otimes B \), or (3) a \( \otimes \)-formula \( A \otimes B \). We identify a \( \otimes \)-formula with the \( \otimes \)-node generated from it. Moreover, for each \( \otimes \)-link \( L \), two labeled nodes \( n_L^L \) and \( n_L^R \) are generated, corresponding to left and right premises respectively. In addition if such a premise is a conclusion of an ID-link or the conclusion of a \( \otimes \)-link, then the labeled node associated with the conclusion is also generated besides \( n_L^L \) or \( n_L^R \). When we say that a labeled
Then we define the undirected graph $T$. As to the incidence relation, each labeled node $n$ has the switch-label set $Lab^n$. Each labeled node $n$ in $T(\Theta)$ has the following labeled nodes and the $\ell$-link with conclusion $\ell$. Otherwise, $down_p$ is undefined.

We define a queue $Below(n)$ of labeled nodes for each labeled node $n$: If $n$ is $n'_L$ or $n''_L$, where $L$ is a $\otimes$-link or $n$ is a conclusion of $\Theta$, then $Below(n) = \emptyset$. Otherwise, $n$ must be a literal or a $\otimes$-formula:

$$Below(n) = \begin{cases} A \otimes B \text{ (if } n \text{ is a premise of a } \otimes \text{-link with conclusion } A \otimes B) \\ n'_L \text{ (if } n \text{ is a left premise of a } \otimes \text{-link } L) \\ n''_L \text{ (if } n \text{ is a right premise of a } \otimes \text{-link } L) \end{cases}$$

According to the type of each link $L$ in $\Theta$, the associated data for the labeled nodes and the $\otimes$-nodes for $T(\Theta)$ are defined as follows:
1. In the case where $L$ is an ID-link, let $c_0 = p$ and $c_1 = p^\perp$ be the conclusion formulas of $L$. For each $i \in \{0, 1\}$,
   (a) $Q^{c_i}_{\text{labeled}} = c_{i+1 \mod 2}$ Below($c_i$).
   (b) $Q^{c_0}_{\text{up}}$, $Q^{c_1}_{\text{down}}$, and $\text{Lab}^\circ$ are empty.

2. In the case where $L$ is a $\otimes$-link, let $c = A \otimes B$ be the conclusion and $p_0 = A$ and $p_1 = B$ be the left premise and the right premise respectively.
   (a) $Q^{c}_{\text{labeled}} = \text{Above}(c)$ Below($c$), where
   $$\text{Above}(c) = \begin{cases} 
   \text{empty} & \text{if both } p_0 \text{ and } p_1 \text{ are } \circ \text{-formulas} \\
   p_i & \text{if } p_i \text{ is a } \otimes \text{-formula or a literal and} \\
   p_{i+1 \mod 2} & \text{is a } \circ \text{-formula} \\
   p_0, p_1 & \text{if both } p_0 \text{ and } p_1 \text{ are a } \otimes \text{-formula or a literal}
   \end{cases}$$
   (b) $Q^{c}_{\text{up}}$ and $\text{Lab}^\circ$ is empty.
   (c) $Q^{c}_{\text{down}} = \begin{cases} 
   p_0, p_1 & \text{if both } p_0 \text{ and } p_1 \text{ are } \circ \text{-formulas} \\
   p_i & \text{if } p_i \text{ is a } \circ \text{-formula or} \\
   p_{i+1 \mod 2} & \text{is a } \otimes \text{-formula or a literal} \\
   \text{empty} & \text{if both } p_0 \text{ and } p_1 \text{ are a } \otimes \text{-formula or a literal}
   \end{cases}$

3. In the case where $L$ is a $\circ$-link, let $c = A \circ B$ be the conclusion and $p_0 = A$ and $p_1 = B$ be the left premise and the right premise respectively. Then we have the following labeled nodes $n_0 = n^c_L$ and $n_1 = n^r_L$ for $L$. For each $i \in \{0, 1\}$
   (a) $Q^{n_i}_{\text{labeled}} = p_i$ if $p_i$ is not a $\circ$-formula. Otherwise, $Q^{n_i}_{\text{labeled}}$ is empty.
   (b) If $i = 0$ then $Q^{n_0}_{\text{up}} = c$. Otherwise, $i$ must be 1. Then $Q^{n_1}_{\text{up}}$ is empty.
   (c) $Q^{n_i}_{\text{down}} = p_i$ if $p_i$ is a $\circ$-formula. Otherwise $Q^{n_i}_{\text{down}}$ is empty.
   (d) If $i = 0$ then $\text{Lab}^{n_0} = \{l_L\}$. Otherwise, $i$ must be 1. Then $\text{Lab}^{n_1} = \{r_L\}$.
   Moreover we have a $\circ$-node $p$ for the $\circ$-link $L$.
   (a) $\text{up}_p$ is $n_0 = n^c_L$.
   (b) $\text{right}_p$ is $n_1 = n^r_L$.
   (c) $\text{down}_p = \begin{cases} 
   A' \otimes B' & \text{if } c \text{ is a premise of a } \otimes \text{-link with conclusion } A' \otimes B' \\
   n^c_L & \text{if } c \text{ is a right premise of a } \circ \text{-link } L' \\
   n^r_L & \text{if } c \text{ is a right premise of a } \circ \text{-link } L' \\
   \text{undefined} & \text{Otherwise}
   \end{cases}$

Then we define the undirected graph $G_{\text{vleft}}(\Theta)$ as follows:

1. The set of nodes consists of the $m + 2\ell$ labeled nodes and the $\ell$ $\circ$-nodes as described above.
2. As to the incidence relation, each labeled node $n$ connects to each labeled node in $Q^{n}_{\text{labeled}}$. Each $\circ$-node in $Q^{n}_{\text{up}}$ connects to $n$ from the port above and each $\circ$-node in $Q^{n}_{\text{down}}$ connects to $n$ from the port below.
3. Each labeled node $n$ has the switch-label set $\text{Lab}^\circ$ on $n$.
4. Each $\circ$-node $p$ for $L$ has the label $L$ on $p$.

If $G_{\text{vleft}}(\Theta)$ is a tree, then the deNM-tree $T(\Theta)$ is $G_{\text{vleft}}(\Theta)$. Otherwise it is undefined. If $T(\Theta)$ is defined, then we can easily see that $T(\Theta)$ is a deNM-tree.

As an example, let us consider the MLL proof net $\Theta$ shown in Figure 5, where the symbol $\otimes$ means a $\circ$-link occurrence. Then $\Theta$ is translated to the deNM-tree $T(\Theta)$ shown in Figure 6.
3.2 The Rewriting System over deNM-Trees

Next we introduce our rewriting system over deNM-trees. In the rewriting system we must specify exactly one node in a deNM-tree that is about to be rewritten, which we call the active node in the deNM-tree. The active node must be a labeled node. Our rewriting system has only three rewrite rules.

- The rewrite rule of Figure 7 is called $O$-elimination: If the active node $n$ is connected to a $O$-node $p\_L$ labeled by $L$ through the port above and the switch-label set $S$ of $n$ contains switch-labels $l\_L$ and $r\_L$, then $p\_L$ is eliminated. Fig. 7.

- The rewrite rule of Figure 8 is called union: If the active node is connected to a labeled node, then these two nodes are merged. The switch-label set of the resulting node is the union of them of the merged two nodes. Fig. 8.

- The rewrite rule called local jump of Figure 9 does not change any nodes: It just changes which labeled node is active. Note that in this rewrite rule, the active node before the rewriting is connected to a $O$-node $p\_L$ through the port below and the active node after the rewriting is the labeled node whose switch-label set contains $r\_L$. We denote the rewriting system consisting of these three rewrite rules above by $R$.

Definition 2. Algorithm $A$ is defined as follows:

Fig. 5. An MLL proof net $\Theta$

Fig. 6. The deNM-tree $T(\Theta)$ obtained from $\Theta$
3.2 The Rewriting System over deNM-Trees

Next we introduce our rewriting system over deNM-trees. In the rewriting system we must specify exactly one node in a deNM-tree that is about to be rewritten, which we call the active node in the deNM-tree. The active node must be a labeled node. Our rewriting system has only three rewrite rules.

- The rewrite rule of Figure 7 is called $\exists$-elimination: If the active node $n$ is connected to a $\exists$-node $p_L$ labeled by $L$ through the port above and the switch-label set $S$ of $n$ contains switch-labels $l_L$ and $r_L$, then $p_L$ is eliminated.

- The rewrite rule of Figure 8 is called union: If the active node is connected to a labeled node, then these two nodes are merged. The switch-label set of the resulting node is the union of them of the merged two nodes.

- The rewrite rule called local jump of Figure 9 does not change any nodes: It just changes which labeled node is active. Note that in this rewrite rule, the active node before the rewriting is connected to a $\exists$-node $p_L$ through the port below and the active node after the rewriting is the labeled node whose switch-label set contains $r_L$.

We denote the rewriting system consisting of these three rewrite rules above by $\mathcal{R}$.

Definition 2. Algorithm $A$ is defined as follows:
data structures. In [7] we gave such data structures and the linear time algorithm based on them in an informal way. In this section we give a formal specification of our algorithm. The formal specification is extracted from the implementation in [8], which performed tests on dozens of instances successfully. In particular, we compared decision results of the new implementation with them of a naive implementation of the correctness condition by de Naurois and Mogbil [10]. Except for MLL proof nets to which the old implementation was not able to give an answer within a reasonable time, both implementations gave the same decision results for all MLL proof structures that we provided.

4.1 Union-Find Data Structures

In this section we give a brief overview of union-find data structures. For a more detailed treatment, the reader can consult [1].

A union-find data structure \( S \) represents a partition of a finite set \( \{1, \ldots, k\} \) but not statically: After operations defined below have been executed over \( S \), the resulting partition can be different from the initial partition. An element \( S \) in \( S \) has the representative element of the subset of \( \{1, \ldots, k\} \) to which \( S \) belongs. Initially each element \( S \) in \( S \) represents a singleton set \( \{S\} \). The union-find data structure \( S \) has two kinds of operations:

1. union(\( S; S' \)): When \( S \) and \( S' \) represents disjoint subsets \( \{u_1; \ldots; u_k\} \) and \( \{v_1; \ldots; v_k\} \) of \( \{1, \ldots, k\} \) respectively, after the operation union(\( S; S' \)) is executed, either \( S \) or \( S' \) becomes the representative element of the union \( \{u_1; \ldots; u_k; v_1; \ldots; v_k\} \) (both \( S \) and \( S' \) become elements of the union).

2. find(\( S \)): It is an element of \( \{1, \ldots, k\} \) that is also the representative of a subset of \( \{1, \ldots, k\} \) to which \( S \) belongs.

Note that after the execution of an operation union(\( S_1; S_2 \)), the element returned by find(\( S \)) may be a different element from an previously returned element by find(\( S \)).

Let \( S \) be a finite execution sequence of operations of a union-find data structure \( S \). Moreover if the base set \( \{1, \ldots, k\} \) of \( S \) has an additional structure as a finite tree \( T \) and each union operation in \( S \) respects the structure, that is, each subset \( S \) in \( S \) is a subtree of \( T \), then the time complexity of \( S \) is \( O(|S|) \) in amortized cost [2]. This means that each operation in \( S \) can be regarded as a constant time operation. Our formal model exploits this fact: Each transition step can be executed in constant time since each union-find data structure in the formal model satisfies the above condition. That is the reason why our correctness condition is linear.

4.2 Data Specification

We suppose \( T(\Theta) \) has \( k \) labeled nodes and \( \ell \) \( O \)-nodes. Our refined rewriting system will consist of rewrite rules that manipulate tuples having the form \( \langle \langle S_{\text{labeled}}; S_{\text{up}}; S_{\text{right}}; S_{\text{elim}} \rangle; \langle a; N; n; P; S_{\text{elim}}; \text{num\_labeled}; \text{num\_O} \rangle \rangle \).

**Theorem 1.** [7]

1. Algorithm A always terminates.
2. Let \( \Theta \) be an MLL proof structure. Then \( \Theta \) is an MLL proof net if and only if Algorithm A with input \( \Theta \) outputs yes.

Some examples can be found in [7] in order to understand the behavior of Algorithm A.

4 Linear Time Correctness Condition: a Formal Approach

Although our rewriting system \( R \) is surprisingly simple, it cannot establish linear time termination: Nodes in a deNM-tree \( T \) may have degrees depending on the number of nodes of \( T \) and then it may take linear time for each rewrite step in the case of a naive implementation. As a result, it may take quadratic total time for termination. Such an example can be seen in [7]. In order to establish linear time termination based on our rewriting system, we must restrict a way of application of rewrite rules using more sophisticated
data structures. In [7] we gave such data structures and the linear time algorithm based on them in an informal way. In this section we give a formal specification of our algorithm. The formal specification is extracted from the implementation in [8], which performed tests on dozens of instances successfully. In particular, we compared decision results of the new implementation with them of a naive implementation of the correctness condition by de Nauris and Mogbil [10]. Except for MLL proof nets to which the old implementation was not able to give an answer within a reasonable time, both implementations gave the same decision results for all MLL proof structures that we provided.

4.1 Union-Find Data Structures

In this section we give a brief overview of union-find data structures. For a more detailed treatment, the reader can consult [1].

A union-find data structure \( S \) represents a partition of a finite set \( \{1, \ldots, k\} \) but not statically: After operations defined below have been executed over \( S \), the resulting partition can be different from the initial partition. An element \( S \) in \( S \) has the representative element of the subset of \( \{1, \ldots, k\} \) to which \( S \) belongs. Initially each element \( S \) in \( S \) represents a singleton set \( \{S\} \). The union-find data structure \( S \) has two kinds of operations:

1. union\((S, S')\): When \( S \) and \( S' \) represents disjoint subsets \( \{u_1, \ldots, u_k\} \) and \( \{v_1, \ldots, v_{k'}\} \) of \( \{1, \ldots, k\} \) respectively, after the operation union\((S, S')\) is executed, either \( S \) or \( S' \) becomes the representative element of the union \( \{u_1, \ldots, u_k, v_1, \ldots, v_{k'}\} \) (both \( S \) and \( S' \) become elements of the union).

2. find\((S)\): It is an element of \( \{1, \ldots, k\} \) that is also the representative of a subset of \( \{1, \ldots, k\} \) to which \( S \) belongs.

Note that after the execution of an operation union\((S_1, S_2)\), the element returned by find\((S)\) may be a different element from an previously returned element by find\((S)\).

Let \( \pi \) be a finite execution sequence of operations of a union-find data structure \( S \). Moreover if the base set \( \{1, \ldots, k\} \) of \( S \) has an additional structure as a finite tree \( T \) and each union operation in \( \pi \) respects the structure, that is, each subset \( S \) in \( \pi \) is a subtree of \( T \), then the time complexity of \( \pi \) is \( O(|\pi|) \) in amortized cost [2]. This means that each operation in \( \pi \) can be regarded as a constant time operation. Our formal model exploits this fact: Each transition step can be executed in constant time since each union-find data structure in the formal model satisfies the above condition. That is the reason why our correctness condition is linear.

4.2 Data Specification

We suppose \( T(\Theta) \) has \( k \) labeled nodes and \( \ell \ \varphi\)-nodes. Our refined rewriting system will consist of rewrite rules that manipulate tuples having the form

\[
\langle \langle S_{\text{labeled}}, S_{\text{up}}, S_{\text{right}}, S_{\text{up}}, S_{\text{right}}, S_{\text{elim}} \rangle, \\
\langle a, N, n, P, S_{\text{elim}}, \text{num}_{\text{labeled}}, \text{num}_{\varphi} \rangle \rangle,
\]

where
4.3 Operational Semantics

We suppose that $a$ points to $N_i = \langle Q_i^{\text{labeled}}, Q_i^{\text{down}}, Q_i^{\text{up}}, S_i^{\text{up}}, Q_i^{\text{right}}, S_i^{\text{right}}, S_i^{\text{up\[up]}}, S_i^{\text{up\[right]}}, S_i^{\text{right\[up]}}, S_i^{\text{right\[right]}} \rangle$.

This means that $\text{find}(a) = i$. Then when given a state $\langle a; N; n; P; S_{\text{elim}}; \text{num\_labeled}; \text{num\_O} \rangle$ we specify the next state. In the following definition, we only describe the components to be changed by applying the $\text{next}(\cdot)$ operator to them. We do not describe the other components that are not changed.

1. **Union rule**: the case where $Q_i^{\text{labeled}} = i_0 Q_i'^{\text{labeled}}$:
   
   (a) The case where $i = \text{find}(i_0)$:
   
   $\text{next}(Q_i^{\text{labeled}}) = Q_i'^{\text{labeled}}$
   
   This case simply discards the index $i_0$, ignoring the redundant information.

   (b) The case where $i \neq \text{find}(i_0)$: Let $i' = \text{find}(i_0)$. Then,
   
   $\text{next}(a) = \text{next}(n_i) = \text{next}(n_i') = \text{find}(\text{union}(n_i; n_i'))$

   $\text{next}(N_{\text{next}}(a)) = \langle \text{union}(S_i^{\text{up}}; S_i'^{\text{up}}); Q_i^{\text{labeled\[+\]}}; Q_i'^{\text{labeled\[+\]}}, Q_i^{\text{down\[+\]}}; Q_i'^{\text{down\[+\]}}, Q_i^{\text{up\[+\]}}; Q_i'^{\text{up\[+\]}}, \text{union}(S_i^{\text{up\[\[up\]}}; S_i'^{\text{up\[\[up\]}}); \text{union}(S_i^{\text{up\[\[right\]}; S_i'^{\text{up\[\[right\]}}); \text{union}(S_i^{\text{up\[\[right\]}; S_i'^{\text{up\[\[right\]}}) \rangle$

   $\text{next}(\text{num\_labeled}) = \text{num\_labeled} + 1$,
   
   where $\text{union\[\[+]$ is the concatenation operation for queues. The labeled node with index $i$ is united to that with index $i'$.

2. **Local Jump rule**: the case where $Q_i^{\text{labeled}} = \emptyset$ and $Q_i^{\text{down}} = j Q_i'^{\text{down}}$:

   (a) The case where $j \notin S_{\text{elim}}$:
   
   $\text{next}(Q_i^{\text{down}}) = Q_i'^{\text{down}}$

   This case simply discards the index $j$. The $\text{O}$-link with index $j$ had been already eliminated.

   (b) The case where $j \notin S_{\text{elim}}$:
   
   $\text{next}(Q_i^{\text{down}}) = Q_i'^{\text{down}}$

   $i' = \text{find}(\text{right}(j))$

   $\text{next}(a) = i'$

   $\text{next}(S_i^{\text{up\[up\]}}) = S_i'^{\text{up\[up\]}}, S_i'^{\text{up\[right\]}}$

   $\text{next}(S_i^{\text{up\[right\]}}) = S_i'^{\text{up\[right\]}}, S_i'^{\text{right\[up\]}}, S_i'^{\text{right\[right\]}}$

   The active node has become the labeled node with index $\text{find}(\text{right}(j))$.

   It maintains the information of indices of $\ominus$-links eliminated.

   - the component $a$ points to the current active labeled node index $i$. This means that $a$ is an element of $S_{\text{labeled}}$.
   - each labeled node $i (1 \leq i \leq k)$ is represented by the following 8-tuple:
   
   $N_i = \langle Q_i^{\text{labeled}}, Q_i^{\text{down}}, Q_i^{\text{up}}, S_i^{\text{up}}, Q_i^{\text{right}}, S_i^{\text{right}}, S_i^{\text{up\[up\]}}, S_i^{\text{up\[right\]}, S_i^{\text{right\[up\]}, S_i^{\text{right\[right\]}}} \rangle$

   where
   
   - $Q_i^{\text{labeled}}$ is a queue data structure and includes a subset of $\{1, \ldots, k\}$.
   - $Q_i^{\text{down}}, Q_i^{\text{up}},$ and $Q_i^{\text{right}}$ are a queue data structure and include a subset of $\{1, \ldots, \ell\}$.
   - $S_i^{\text{up}}, S_i^{\text{right}}, S_i^{\text{up\[up\]}, S_i^{\text{up\[right\]}, S_i^{\text{right\[up\]}, S_i^{\text{right\[right\]}} are an element of $S_{\text{up}}, S_{\text{right}}, S_{\text{up\[up\]}, S_{\text{up\[right\], S_{\text{right\[up\]}, S_{\text{right\[right\]}}}}$ respectively.

   - In the case where the deNM-tree is $T(\Theta)$ initially,
   
   $S_i^{\text{up}} = S_i^{\text{right}} = S_i^{\text{up\[up\]} = S_i^{\text{up\[right\]} = \text{undefined}}$

   where $S$ is undefined, it is identified with $\emptyset$ in the union operation. In addition we have defined the initial values of $Q_i^{\text{labeled}}, Q_i^{\text{down}}, Q_i^{\text{up}}, Q_i^{\text{right}}$ already in the previous section. Then we have the properties

   $\bigcup_{1 \leq i \leq k} Q_i^{\text{labeled}} = \{1, \ldots, k\}$

   $\bigcup_{1 \leq i \leq k} Q_i^{\text{down}} = \bigcup_{1 \leq i \leq k} Q_i^{\text{up}} = \bigcup_{1 \leq i \leq k} Q_i^{\text{right}} = \{1, \ldots, \ell\}$.

   - For each $i (1 \leq i \leq k)$, $n_i$ is an element of $S_{\text{labeled}}$. Informally, $n_i$ is the representative element of the subset of $\{1, \ldots, k\}$ to which $i$ belongs. Initially $n_i$ is $i$.

   - Initially $S_{\text{elim}} = -1$

   - Initially
   
   $\text{num}_{\text{labeled}} = k - 1, \text{num}_{\ominus} = \ell$

   - Each $\ominus$-node $p_j (1 \leq j \leq \ell)$ has a triple
   
   $P_j = \langle \text{up}_j, \text{right}_j, \text{down}_j \rangle$

   where $\text{up}_j, \text{right}_j, \text{down}_j$ are an element of $S_j$ and we have defined the initial values of $\text{up}_j, \text{right}_j, \text{down}_j$ already in the previous section.
4.3 Operational Semantics

We suppose that a points to

\[ N_i = \langle Q_i^{\text{labeled}}, Q_i^{\text{down}}, Q_i^{\text{up}}, S_i^{\text{up}}, S_i^{\text{right}}, S_i^{\text{up} \cup \text{right}}, S_i^{\text{right}} \rangle. \]

This means that \( \text{find}(a) = i \). Then when given a state

\[ (a, N_i, n, P, S_{\text{elim}}, \text{num}_{\text{labeled}}, \text{num}_{\varphi}) \]

we specify the next state. In the following definition, we only describe the components to be changed by applying the next(−) operator to them. We do not describe the other components that are not changed.

1. **Union rule**: the case where \( Q_i^{\text{labeled}} = \emptyset \) and \( Q_i^{\text{down}} = \emptyset \):

   (a) The case where \( i = \text{find}(i_0) \):

   \[ \text{next}(Q_i^{\text{labeled}}) = Q_i'^{\text{labeled}} \]

   This case simply discards the index \( i_0 \), ignoring the redundant information.

   (b) The case where \( i \neq \text{find}(i_0) \): Let \( i' \) be \( \text{find}(i_0) \). Then,

   \[ \text{next}(a) = \text{next}(n_i) = \text{next}(n_{i'}) = \text{find}(\text{union}(n_i, n_{i'})) \]

   \[ \text{next}(N_{\text{next}(a)}) = \langle \text{union}(S_i, S_i'), Q_i^{\text{labeled}} + Q_i'^{\text{labeled}}, Q_i^{\text{down}} + Q_i'^{\text{down}}, Q_i^{\text{up}} + Q_i'^{\text{up}}, \text{union}(S_i^{\text{up}}, S_i'^{\text{up}}), Q_i^{\text{right}} + Q_i'^{\text{right}}, \text{union}(S_i^{\text{right}}, S_i'^{\text{right}}), \text{union}(S_i^{\text{up} \cup \text{right}}, S_i'^{\text{up} \cup \text{right}}) \rangle \]

   \[ \text{next}(\text{num}_{\text{labeled}}) = \text{num}_{\text{labeled}} - 1, \]

   where \( + \) is the concatenation operation for queues. The labeled node with index \( i \) is united to that with index \( i' \).

2. **Local Jump rule**: the case where \( Q_i^{\text{labeled}} = \emptyset \) and \( Q_i^{\text{down}} = \emptyset \):

   (a) The case where \( j \in S_{\text{elim}} \):

   \[ \text{next}(Q_i^{\text{down}}) = Q_i'^{\text{down}} \]

   This case simply discards the index \( j \). The \( \varphi \)-link with index \( j \) had been already eliminated.

   (b) The case where \( j \notin S_{\text{elim}} \):

   \[ \text{next}(Q_i^{\text{down}}) = Q_i'^{\text{down}} \]

   \[ i' = \text{find}(\text{right}_j) \]

   \[ \text{next}(a) = i' \]

   \[ \text{next}(S_i^{\text{up}'}) = S_i^{\text{up} \cup \text{up}'} \]

   \[ \text{next}(S_i^{\text{right}'}) = S_i^{\text{right} \cup \text{right}'} \]

   The active node has become the labeled node with index \( \text{find}(\text{right}_j) \).
The case where \( \text{find}(j) = \text{find}(S_{\text{right}}) \):

(a) The case where \( \text{find}(j) = \text{find}(S_{\text{right}}) \):

\[
\begin{align*}
\text{next}(Q_{\text{up}}^i) &= Q_{\text{up}}^i \\
\text{next}(Q_{\text{labeled}}^i) &= \{ Q_{\text{labeled}}^i \text{down}^j \} \text{ if } \text{down}^j \neq -1 \\
\text{next}(Q_{\text{right}}^i) &= \text{union}(j, S_{\text{up}}^i) \\
\text{next}(\text{num}_{\Sigma}) &= \text{num}_{\Sigma} - 1 \\
\text{next}(S_{\text{elim}}) &= \text{union}(j, S_{\text{elim}})
\end{align*}
\]

The \( \Sigma \)-link with index \( j \) is eliminated successfully.

(b) The case where \( \text{find}(j) \neq \text{find}(S_{\text{right}}) \) and \( \text{find}(j) = \text{find}(S_{\text{up}}) \):

\[
\begin{align*}
\text{next}(Q_{\text{up}}^i) &= Q_{\text{up}}^i \\
\text{next}(S_{\text{up}}^i) &= \text{union}(j, S_{\text{up}}^i) \\
\text{next}(S_{\text{right}}(j)) &= Q_{\text{right}}^i \\
\text{next}(\text{num}_{\Sigma}) &= \text{num}_{\Sigma} - 1 \\
\text{next}(S_{\text{elim}}) &= \text{union}(j, S_{\text{elim}})
\end{align*}
\]

The \( \Sigma \)-link with index \( j \) cannot be eliminated at this moment. But in order to eliminate it later, \( j \) is put in another queue \( Q_{\text{right}}^i \) for some \( i' \) (1 ≤ \( i' \) ≤ \( k \)).

(c) The case where \( \text{find}(j) \neq \text{find}(S_{\text{right}}) \) and \( \text{find}(j) \neq \text{find}(S_{\text{up}}) \):

\[
\begin{align*}
\text{next}(Q_{\text{up}}^i) &= Q_{\text{up}}^i \\
\text{next}(S_{\text{up}}^i) &= \text{union}(j, S_{\text{up}}^i) \\
\text{next}(S_{\text{up}}^i) &= \text{union}(j, S_{\text{up}}^i)
\end{align*}
\]

The \( \Sigma \)-link with index \( j \) cannot be eliminated at this moment. The next trial to eliminate the \( \Sigma \)-link will be done with \( j \) in \( Q_{\text{right}}^i \) for some \( i' \) (1 ≤ \( i' \) ≤ \( k \)) in \( \Sigma \)-elimination rule 2 below.

4. \( \Sigma \)-elimination rule 2: the case where \( Q_{\text{labeled}}^i = \emptyset, Q_{\text{down}}^i = \emptyset, Q_{\text{up}}^i = \emptyset \), and \( Q_{\text{right}}^i = jQ_{\text{right}}^i \) (this case is completely symmetrical to the above case):

(a) The case where \( \text{find}(j) = \text{find}(S_{\text{up}}) \):

\[
\begin{align*}
\text{next}(Q_{\text{right}}^i) &= Q_{\text{right}}^i \\
\text{next}(Q_{\text{labeled}}^i) &= \{ Q_{\text{labeled}}^i \text{down}^j \} \text{ if } \text{down}^j \neq -1 \\
\text{next}(\text{num}_{\Sigma}) &= \text{num}_{\Sigma} - 1 \\
\text{next}(S_{\text{elim}}) &= \text{union}(j, S_{\text{elim}})
\end{align*}
\]

The \( \Sigma \)-link with index \( j \) is eliminated successfully.

(b) The case where \( \text{find}(j) \neq \text{find}(S_{\text{up}}) \) and \( \text{find}(j) = \text{find}(S_{\text{up}}) \):

\[
\begin{align*}
\text{next}(Q_{\text{right}}^i) &= Q_{\text{right}}^i \\
\text{next}(S_{\text{right}}^i) &= \text{union}(j, S_{\text{right}}^i) \\
\text{next}(Q_{\text{right}}(j)) &= Q_{\text{right}}^i
\end{align*}
\]

The \( \Sigma \)-link with index \( j \) cannot be eliminated at this moment. But in order to eliminate it later, \( j \) is put in another queue \( Q_{\text{up}}^i \) for some \( i' \) (1 ≤ \( i' \) ≤ \( k \)).
(c) The case where $\text{find}(j) \neq \text{find}(S_{\text{up}})$ and $\text{find}(j) \neq \text{find}(S_{\text{up}})$:

\[
\begin{align*}
\text{next}(Q_{\text{right}}) &= Q_{\text{right}}' \\
\text{next}(S_{\text{right}}) &= \text{union}(j, S_{\text{right}}') \\
\text{next}(S_{\text{right}}') &= \text{union}(j, S_{\text{right}}')
\end{align*}
\]

The $\varphi$-link with index $j$ cannot be eliminated at this moment. The next trial to eliminate the $\varphi$-link will be done with $j$ in $Q_{\text{up}}'$ for some $j' (1 \leq j' \leq k)$ in $\varphi$-elimination rule 1 above.

5. Otherwise, i.e., the case where $Q'_{\text{labeled}} = \emptyset$, $Q'_{\text{down}} = \emptyset$, $Q'_{\text{up}} = \emptyset$, and $Q'_{\text{right}} = \emptyset$: It terminates. If num_labeled = 0 and num_\varphi = 0 then the output is yes. Otherwise, the output is no.

**Remark 1.** In order to establish the correctness of the algorithm, we need the case 3.(b): without the case, the algorithm would lead to a deadlock state and it would judge that correct proof nets are not. Figures 5 and 6 shows such an example: $\Theta$ is correct but it would not be able to eliminate $\varphi_1$.

Similarly, we need the case 4. (b): without the case, the algorithm would lead to a deadlock state and it would judge that correct proof nets are not. Figures 10 and 11 shows such an example: $\Theta'$ is correct but it also would not be able to eliminate $\varphi_1$.

Fig. 10. An MLL proof net $\Theta'$

**Theorem 2.** [?] Let $\Theta$ be an MLL proof structure such that the deNM-tree $T(\Theta)$ is well-defined. We suppose that the initial state for $T(\Theta)$ is

$\langle a, N, p, P, S_{\text{elim}}, k, \ell \rangle$

Then

1. The rewriting system described above terminates in linear time.
Fig. 11. The deNM-tree $T(\Theta')$ obtained from $\Theta'$

2. $\Theta$ is an MLL proof net if and only if it terminates in
$$(a', N', p', P', S_{\text{elim}}', 0, 0)$$
for some $a'$, $N'$, $p'$, $P'$, $S_{\text{elim}}'$.

Remark 2. In the proof of the only-if part of the second part, the following invariant, which claims deadlock-freedom, is essential: In each state of a transition process, for each $j$ ($1 \leq j \leq \ell$), if the $\&$-link with index $j$ is not eliminated at this moment, then for some $i$ ($1 \leq i \leq k$), $Q^i_{\text{up}}$ or $Q^i_{\text{right}}$ includes $j$.

5 Concluding Remarks

In this paper we have established a formal model of our linear time correctness algorithm for MLL proof nets based on a rewriting system over trees called deNM-trees, where in order to guarantee the correctness of the algorithm, we have introduced a deadlock prevention mechanism. Moreover we showed that the deadlock-freedom property can be formalized as an invariant of the formal model.

There are some future research directions.

- Extensions of our result to variants like noncommutative fragments or extensions like MALL or MELL.
- Implementation issues: In particular, to some extent it may be possible to have several active nodes in a deNM-tree and to exploit parallelism using one or many multi-core processors.
- Application to proof search: In [8] in order to search MLL proof nets for a given MLL formula, a backtracking mechanism and a naive implementation of de Naurois and Mogbil's correctness condition are combined. Our result may be used to obtain more elegant implementations for MLL proof search and its extensions. That was our original motivation for this work.
References


A An Example of the Transformation of MLL Proof Structures to deNM-Trees

The MLL proof net \( \Theta'' \) shown in Figure 12 induces the deNM-tree \( T(\Theta'') \) shown in Figure 13. Then \( T(\Theta'') \) has some associated data, for instance:

\[
\begin{align*}
Q_{\text{labeled}}^5 &= \text{empty} & Q_{\text{labeled}}^{n_1} &= 8 \\
Q_{\text{up}}^5 &= \text{empty} & Q_{\text{up}}^{n_1} &= 2 \\
Q_{\text{down}}^5 &= 1, 4 & Q_{\text{down}}^{n_1} &= \text{empty} \\
Q_{\text{labeled}}^7 &= 11, 12 & \text{up}_1 &= n_1 \\
Q_{\text{up}}^7 &= \text{empty} & \text{right}_1 &= n_1 \\
Q_{\text{down}}^7 &= \text{empty} & \text{down}_1 &= 5
\end{align*}
\]
We propose a mechanism for automating discovery of definitions, that, when added to a logic system for which we have a theorem prover, extends it to support an embedding of a new logic system into it. As a result, the synthesized definitions, when added to the prover, implement a prover for the new logic.

As an instance of the proposed mechanism, we derive a Prolog theorem prover for an interesting but unconventional epistemic Logic by starting from the sequent calculus $G4IP$ that we extend with operator definitions to obtain an embedding in intuitionistic propositional logic ($IPC$). With help of a candidate definition formula generator, we discover epistemic operators for which axioms and theorems of Artemov and Protopopescu's Intuitionistic Epistemic Logic ($IEL$) hold and formulas expected to be non-theorems fail. We compare the embedding of $IEL$ in $IPC$ with a similarly discovered successful embedding of Dosen's double negation modality, judged inadequate as an epistemic operator. Finally, we discuss the failure of the necessitation rule for an otherwise successful $S4$ embedding and share our thoughts about the intuitions explaining these differences between epistemic and alethic modalities in the context of the Brouwer-Heyting-Kolmogorov semantics of intuitionistic reasoning and knowledge acquisition.

Keywords: automatic synthesis of logic systems, deriving new theorem provers via program synthesis, epistemic intuitionistic logic, propositional intuitionistic logic, Prolog-based theorem provers, embedding of modal logics into intuitionistic logic.

1 Introduction

Deriving new logic systems and discovering relationships between them not only requires a knowledge-intensive understanding of the intricate connections between their axioms and inference rules but it is also a time-intensive trial and error process for the human logician. This is especially the case for logic systems that depart from the usual expectations coming from the prevalent use of classical logic in today's computational tools and methodologies, as well as from our familiarity with more commonly used forms of modal logic (e.g., alethic, temporal).

This motivates our effort to explore ways to automate this process, resulting not only in discovering some salient relationships between new and well-established logic systems, but also in developing automated tools that can generate these systems from scratch. The proposed mechanism has the potential to revolutionize the way we approach the development of new logical frameworks, making it possible to explore uncharted territories of logic without requiring a deep understanding of the underlying mathematics and syntax.

Fig. 12. An MLL proof net $\Theta''$

Fig. 13. The deNM-tree $T(\Theta'')$ obtained from $\Theta''$
Synthesis of Modality Definitions and a Theorem Prover for Epistemic Intuitionistic Logic

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
paul.tarau@unt.edu

Abstract. We propose a mechanism for automating discovery of definitions, that, when added to a logic system for which we have a theorem prover, extends it to support an embedding of a new logic system into it. As a result, the synthesized definitions, when added to the prover, implement a prover for the new logic. As an instance of the proposed mechanism, we derive a Prolog theorem prover for an interesting but unconventional epistemic Logic by starting from the sequent calculus G4IP that we extend with operator definitions to obtain an embedding in intuitionistic propositional logic (IPC). With help of a candidate definition formula generator, we discover epistemic operators for which axioms and theorems of Artemov and Protopopescu’s Intuitionistic Epistemic Logic (IEL) hold and formulas expected to be non-theorems fail. We compare the embedding of IEL in IPC with a similarly discovered successful embedding of Dosen’s double negation modality, judged inadequate as an epistemic operator. Finally, we discuss the failure of the necessitation rule for an otherwise successful S4 embedding and share our thoughts about the intuitions explaining these differences between epistemic and alethic modalities in the context of the Brouwer-Heyting-Kolmogorov semantics of intuitionistic reasoning and knowledge acquisition.

Keywords: automatic synthesis of logic systems, deriving new theorem provers via program synthesis, epistemic intuitionistic logic, propositional intuitionistic logic, Prolog-based theorem provers, embedding of modal logics into intuitionistic logic.

1 Introduction

Deriving new logic systems and discovering relationships between them not only requires a knowledge-intensive understanding of the intricate connections between their axioms and inference rules but it is also a time-intensive trial and error process for the human logician. This is especially the case for logic systems that depart from the usual expectations coming from the prevalent use of classical logic in today’s computational tools and methodologies, as well as from our familiarity with more commonly used forms of modal logic (e.g., alethic, temporal).

This motivates our effort to explore ways to automate this process, resulting not only in discovering some salient relationships between new and well-established logic
systems, but also in software artifacts (e.g., automated theorem provers) facilitating reasoning in these less explored new logics.

Epistemic Logic Systems have been derived often in parallel and sometime as afterthoughts of alethic Modal Logic Systems, in which modalities are defined by axioms and additional inference rules extending classical logic.

In the context of Answer Set Programming (ASP) epistemic logics hosted in this framework like e.g., [1–3] show that intermediate logics\(^1\) (e.g., equilibrium logic, [4]) be extended with definitions of epistemic operators. Steps\(^2\), further below classical logic or ASP, are taken in recent work [5], based on the Brouwer-Heyting-Kolmogorov (BHK) view of intuitionistic logic that takes into account the constructive nature of knowledge, modeling more accurately the connection between proof systems and the related mental processes. Along these lines, our inquiry into epistemic logic will focus on knowledge vs. truth seen as intuitionistic provability. Like in the case of embedding epistemic operators into ASP systems, but with automation in mind, we will design a synthesis mechanism for epistemic operators via embedding in IPC. For this purpose we will generate candidate formulas that verify axioms, theorems and rules and fail on expected non-theorems. For this purpose, we will use a lightweight IPC theorem prover and we will also show that this view generalizes to a mechanism for discovering, when possible, a simple embedding of a given logic into IPC and derivation of a theorem prover for it.

Our starting point is Artemov and Protopopescu’s Intuitionistic Epistemic Logic (IEL) [5] that will provide the axioms, theorems and non-theorems stating the requirements that must hold for the definitions extending IPC. The discovery mechanism will also bring up Dosen’s interpretation of double negation [6] as a potential epistemic operator and we will look into applying the same discovery mechanisms to find an embedding of modal logic S4 in IPC, with special focus on the impact of the necessitation rule, which requires that all theorems of the logic are necessarily true.

The rest of the paper is organized as follows. Section 2 overviews Artemov and Protopopescu’s Intuitionistic Epistemic Logic (IEL). Section 3 introduces the G4IP sequent calculus prover for Intuitionistic Propositional Logic (IPC). Section 4 describes the generator for candidate formulas extending IPC with modal operator definitions. Section 5 explains the discovering of the definitions that ensure the embedding of IEL into IPC and the embedding of Dosen’s double negation as a modality operator. It also discusses the intuitions behind the embedding of IEL, including the epistemic equivalent of the necessity rule, in IPC and the adequacy of this embedding as a constructive mechanism for reasoning about knowledge. Section 6 studies the case of the S4 modal logic and the failure of the necessity rule, indicating the difficulty of embedding it in IPC by contrast to IEL. Section 7 overviews some related work and section 8 concludes the paper.

The paper is written as a literate SWI-Prolog program with its extracted code at https://raw.githubusercontent.com/ptarau/TypesAndProofs/master/ieltp.pro.

---

\(^1\) Logics stronger than intuitionistic but weaker than classical.

\(^2\) Actually infinitely many, as there’s an infinite lattice of intermediate logics between classical and intuitionistic logic.
2 Overview of Artemov and Protopopescu’s IEL logic

In [5] a system for Intuitionistic Epistemic Logic is introduced that “maintains the original Brouwer-Heyting-Kolmogorov semantics for intuitionism and is consistent with the well-known approach that intuitionistic knowledge be regarded as the result of verification”. Instead of the classic, alethic-modalities inspired K operator for which

\[ K\alpha \rightarrow \alpha \]

Artemov and Protopopescu argue that co-reflection expresses better the idea of constructivity of truth

\[ \alpha \rightarrow K\alpha \]

They also argue that this applies to both belief and knowledge i.e., that “The verification-based approach allows that justifications more general than proof can be adequate for belief and knowledge”.

On the other hand, they consider intuitionistic reflection acceptable, expressing the fact that “known propositions cannot be false”:

\[ K\alpha \rightarrow \neg\neg\alpha \]

Thus, they position intuitionistic knowledge of \( \alpha \) between \( \alpha \) and \( \neg\neg\alpha \) and given that (via Glivenko’s transformation [7]) applying double negation to a formula embeds classical propositional calculus into IPC, they express this view as:

\[ \text{Intuitionistic Truth} \Rightarrow \text{Intuitionistic Knowledge} \Rightarrow \text{Classical Truth}. \]

They axiomatize the system IEL as follows.

1. Axioms of propositional intuitionistic logic;
2. \( K(\alpha \rightarrow \beta) \rightarrow (K\alpha \rightarrow K\beta) \); (distribution)
3. \( \alpha \rightarrow K\alpha \). (co-reflection)
4. \( K\alpha \rightarrow \neg\neg\alpha \) (intuitionistic reflection)

Rule Modus Ponens.

They also argue that a weaker logic of belief (IEL \( \neg \)) is expressed by considering only axioms 1,2,3.

3 The G4ip prover for IPC

We will describe next our lightweight propositional intuitionistic theorem prover, that will be used to discover an embedding of IEL into IPC.

3.1 The LJT/G4ip calculus, (restricted here to the implicational fragment)

Motivated by problems related to loop avoidance in implementing Gentzen’s LJ calculus, Roy Dyckhoff [8] introduces the following rules for the G4ip calculus\(^3\).

\(^3\) Originally called the LJT calculus in [8]. Restricted here to its key implicational fragment.
We validate the prover by testing it on the implicational subset, derived via the Curry-Howard isomorphism \[11\], then against Roy Dyckhoff’s Prolog implementation \[4\], working on formulas up to size 12. Finally we run it on human-made tests \[5\], on which we get no errors, solving correctly 161 problems, with a 60 seconds timeout, compared with the 175 problems solved by Roy Dyckhoff’s more refined, heuristics-based \(400\) lines prover, with the same timeout \[6\]. We refer to \[11\] for the derivation steps of variants of this prover working on the implicational and nested Horn clause fragments of IPC. While more sophisticated tableau-based provers are available for IPC among which we mention the excellent Prolog-based \(f\)Cube \[12\], our prover’s compact size and adequate performance will suffice \[7\].

4 The definition formula generator

We start with a candidate formula generator that we will constrain further to be used for generating candidate definitions of our modal operators.

4.1 Generating Operator Trees

We generate all formulas of a given size by decreasing the available size parameter at each step when nodes are added to a tree representation of a formula. Prolog’s DCG mechanism is used to collect the leaves of the tree.

4 https://github.com/ptarau/TypesAndProofs/blob/master/third_party/dyckhoff_orig.pro
5 at http://iltp.de
6 https://github.com/ptarau/TypesAndProofs/blob/master/tester.pro
7 In fact, our prover is faster than both fCube and Dyckhoff’s prover on the set of formulas of small size on which our definition induction algorithm will run.

3.2 A Lightweight Theorem Prover for Intuitionistic Propositional Logic

Starting from the sequent calculus for the intuitionistic propositional logic in G4ip \[8\], to which we have also added rules for the “\(<=>\)” relation, we obtain the following lightweight IPC prover.

\[
\begin{align*}
\text{prove_in_ipc(T)} & :- \text{prove_in_ipc(T,[])}.
\text{prove_in_ipc(A,Vs)} & :- \text{memberchk(A,Vs)},!.
\text{prove_in_ipc(\_ ,Vs)} & :- \text{memberchk(false,Vs)},!.
\text{prove_in_ipc(A \&\& B,Vs)} & :- !, \text{prove_in_ipc(B,[A|Vs]),prove_in_ipc(A,[B|Vs])}.
\text{prove_in_ipc((A \rightarrow B),Vs)} & :- !, \text{prove_in_ipc(B,[A|Vs])}.
\text{prove_in_ipc(A \& B,Vs)} & :- !, \text{prove_in_ipc(A,Vs),prove_in_ipc(B,Vs)}.
\text{prove_in_ipc(G,Vs1)} & :- % atomic or disj or false
\end{align*}
\]

\[
\begin{align*}
\text{prove_in_ipc(T)} & :- \text{prove_in_ipc(T,[])}.
\text{prove_in_ipc(A,Vs)} & :- \text{memberchk(A,Vs)},!.
\text{prove_in_ipc(\_ ,Vs)} & :- \text{memberchk(false,Vs)},!.
\text{prove_in_ipc(A \&\& B,Vs)} & :- !, \text{prove_in_ipc(B,[A|Vs]),prove_in_ipc(A,[B|Vs])}.
\text{prove_in_ipc((A \rightarrow B),Vs)} & :- !, \text{prove_in_ipc(B,[A|Vs])}.
\text{prove_in_ipc(A \& B,Vs)} & :- !, \text{prove_in_ipc(A,Vs),prove_in_ipc(B,Vs)}.
\text{prove_in_ipc(G,Vs1)} & :- % atomic or disj or false
\end{align*}
\]
genOperatorTree(N,Ops,Tree,Leaves):-
    genOperatorTree(Ops,Tree,N,0,Leaves,[]).
    genOperatorTree(_,V,N,N)--->[V].
    genOperatorTree(Ops,OpAB,SN1,N3)-->{ SN1>0,N1 is SN1-1,
    member(Op,Ops),make_oper2(Op,A,B,OpAB) },
    genOperatorTree(Ops,A,N1,N2),
    genOperatorTree(Ops,B,N2,N3).


4.2 Synthesizing the definitions of modal operators

As we design a generic definition discovery mechanism, we will denote our modal operators as follows, generically.
– "#" for □ = necessary and K = known
– "*" for ♦ = possible and M = knowable

After the operator definitions:- op( 500, fy, #).
:- op( 500, fy, *).
we specify our generator as covering the usual binary operators and we constrain it to have at least one of the leaves of its generated trees to be a variable. Besides the false constant used in the definition of negation, we introduce also a new constant symbol "?" assumed not to occur in the language. Its role will be left unspecified until the possible synthesized definitions will be filtered. We will constrain candidate definitions to ensure that axioms and selected theorems hold and selected non-theorems fail.

genDef(M,Def):-genDef(M,[(->),(&),(v)],[false,?],Def).
    between(0,M,N),
    genOperatorTree(N,Ops,T,Vs),
    pick_leaves(Vs,[X|Cs]),
    term_variables(Vs,[X]).

Iteration over integers N between 0 and M is provided by the built-in between/3. Variables are extracted from a term using the built-in term_variables. Next, leaves of the generated trees will be picked from a given set.

pick_leaves([],_).
pick_leaves([V|Vs],Ls):-member(V,Ls),pick_leaves(Vs,Ls).

We first expand our operator definitions for the "~" negation and "*" modal operator while keeping atomic variables and the special constant false untouched.

Note that, with the exception of the !/0 and memberchk/2 built-ins, used only as performance enhancers, the code is actually a set of Horn-clauses as select/3 is a library predicate with a pure Horn clause definition.

We validate the prover by testing it on the implicational subset, derived via the Curry-Howard isomorphism [11], then against Roy Dyckhoff’s Prolog implementation⁴, working on formulas up to size 12. Finally we run it on human-made tests⁵, on which we get no errors, solving correctly 161 problems, with a 60 seconds timeout, compared with the 175 problems solved by Roy Dyckhoff’s more refined, heuristics-based 400 lines prover, with the same timeout⁶. We refer to [11] for the derivation steps of variants of this prover working on the implicational and nested Horn clause fragments of IPC. While more sophisticated tableau-based provers are available for IPC among which we mention the excellent Prolog-based fCube [12], our prover’s compact size and adequate performance will suffice ⁷.

4 The definition formula generator

We start with a candidate formula generator that we will constrain further to be used for generating candidate definitions of our modal operators.

4.1 Generating Operator Trees

We generate all formulas of a given size by decreasing the available size parameter at each step when nodes are added to a tree representation of a formula. Prolog’s DCG mechanism is used to collect the leaves of the tree.

Note: ⁴ https://github.com/ptarau/TypesAndProofs/blob/master/third_party/dyckhoff_orig.pro
⁵ at http://iltp.de
⁶ https://github.com/ptarau/TypesAndProofs/blob/master/tester.pro
⁷ In fact, our prover is faster than both fCube and Dyckhoff’s prover on the set of formulas of small size on which our definition induction algorithm will run.
The special case for expanding a candidate operator definition requires a fresh variable for each instance, ensured by Prolog's built-in `copy` term.

Other operators are traversed generically by using Prolog's "=.." built-in and by recursing with `expand_def_list` on their arguments.

The predicate `prove_with_def` refines our G4ip prover by first expanding the definitions extending IPC with a given candidate modality.

The definition synthesizer will filter the candidate definitions provided by `genDef` such that the predicate `prove_with_def` succeeds on all theorems and fails on all non-theorems, provided as names of the facts of arity 1 containing them. It iterates over the theorems and non-theorems using the built-in `forall`. The negation-as-failure built-in `+` is used to preempt success on non-theorems.

Example 1
Candidate definitions up to size 2
```prolog
?- forall(genDef(2,Def),println(Def)).
#A :- A
#A :- A -> A
#A :- A -> false
#A :- A -> ?
#A :- false -> A
#A :- ? -> A
```

4.2 Synthesizing the definitions of modal operators

As we design a generic definition discovery mechanism, we will denote our modal operators as follows, generically.

- "#" for “□”=necessary and “K”=known
- “*” for “◊”=possible and “M”=knowable

After the operator definitions
```prolog
:- op( 500, fy, #).
:- op( 500, fy, *).
```

we specify our generator as covering the usual binary operators and we constrain it to have at least one of the leaves of its generated trees to be a variable. Besides the false constant used in the definition of negation, we introduce also a new constant symbol “?” assumed not to occur in the language. Its role will be left unspecified until the possible synthesized definitions will be filtered. We will constrain candidate definitions to ensure that axioms and selected theorems hold and selected non-theorems fail.

```prolog
def_synth(M,D):-def_synth(M,iel_th,iel_nth,D).
def_synth(M,Th,NTh,D):-
genDef(M,D),
forall(call(Th,T),prove_with_def(D,T)),
forall(call(NTh,NT), +prove_with_def(D,NT)).
```

Note that the generator first builds smaller formulas and then larger ones up the specified maximum size.

Example 1
Candidate definitions up to size 2
```prolog
?- forall(genDef(2,Def),println(Def)).
#A :- A
#A :- A -> A
#A :- A -> false
#A :- A -> ?
#A :- false -> A
#A :- ? -> A
```

Iteration over integers N between 0 and M is provided by the built-in `between/3`. Variables are extracted from a term using the built-in `term_variables`. Next, leaves of the generated trees will be picked from a given set.

```prolog
pick_leaves([],[]).
pick_leaves([V|Vs],Ls):-member(V,Ls),pick_leaves(Vs,Ls).
```

We first expand our operator definitions for the “¬” negation and “*” modal operator while keeping atomic variables and the special constant `false` untouched.
exp_ndef(_,-,false,R):-!,R=false.
exp_ndef(_,-,false,R):-atomic(A),!,R= A.
exp_ndef(D,"(A),(B→false))):-!,exp_ndef(D,A,B).
exp_ndef(D,"(A),R):-!,exp_ndef(D,"(¬(¬(A))),R).

The special case for expanding a candidate operator definition D requires a fresh variable for each instance, ensured by Prolog’s built-in copy_term.

exp_ndef(D,"(X),R):-!,copy_term(D,"(#(¬(¬(X))),R),exp_ndef(D,T,R).

Other operators are traversed generically by using Prolog’s “..” built-in and by recursing with exp_ndef_list on their arguments.

exp_ndef(D,A,B):-
   A=..[F|Xs],
   exp_ndef_list(D,Xs,Ys),
   B=..[F|Ys].

exp_ndef_list(_,[],[]).
exp_ndef_list(D,[X|Xs],[Y|Ys]):-
   exp_ndef(D,X,Y),
   exp_ndef_list(D,Xs,Ys).

The predicate prove_with_def refines our G4jip prover by first expanding the definitions extending IPC with a given candidate modality.

prove_with_def(Def,T0):-exp_ndef_def(Def,T0,T1),prove_in_ipc(T1,[]).

The definition synthesizer will filter the candidate definitions provided by genDef such that the predicate prove_with_def succeeds on all theorems and fails on all non-theorems, provided as names of the facts of arity 1 containing them. It iterates over theorems and non-theorems using the built-in forall. The negation-as-failure built-in \+ is used to preempt success on non-theorems.

def_synth(M,D):-def_synth(M,iel_th,iel_nth,D).

def_synth(M,Th,NT,D):-
   genDef(M,D),
   forall(call(Th,T),prove_with_def(D,T)),
   forall(call(NT,NT), \+prove_with_def(D,NT)).

Note that the generator first builds smaller formulas and then larger ones up the specified maximum size.

**Example 1** Candidate definitions up to size 2

?- forall(genDef(2,Def),println(Def)).

#A :- A
#A :- A -> A
#A :- A -> false
#A :- A -> ?
#A :- false -> A
#A :- ? -> A
The necessitation rule in a modal logic requires that if $T$ is a theorem than $\# T$ is also a theorem. This expresses the fact that the theorems of the logic are necessarily true, or in an epistemic context, that if $T$ is an (intuitionistically proven) theorem, then the agent knows $T$. Thus, we define (implicit) facts via a Prolog rule that states that the (generic) necessity operator "#" applied to proven theorems or axioms generates new theorems.

\[
ie_l\text{\_}nec\_th(T) :\text{-} iel\_th(T).
\]
\[
ie_l\text{\_}nec\_th(\# T) :\text{-} iel\_th(T).
\]

Finally, we obtain the discovery algorithm for IEL formula definitions and for IEL extended with the necessitation rule.

\[
ie_l\text{\_}discover :\text{-} \\
(\text{backtrack\_over}((\text{def\_synth}(2,iel\_th,iel\_nth,D),println(D)))).
\]
\[
ie_l\text{\_}nec\_discover :\text{-} \\
(\text{backtrack\_over}((\text{def\_synth}(2,iel\_nec\_th,iel\_nth,D),println(D)))).
\]
\[
\text{backtrack\_over}(\text{Goal}) :\text{-} \text{call}(\text{Goal}),\text{fail};\text{true}.
\]
\[
\text{println}(T) :\text{-} \text{numbervars}(T,0,\_),\text{writeln}(T).
\]

Note the use of backtrack/1 to backtrace over all answers to a given goal. We run iel\_discover, ready to see the surviving definition candidates.

Example 2
Definition discovery without the necessitation rule.

```
?\- iel\_discover.
#A :-(A -> false) -> A
#A :-(A -> false) -> false
#A :-(A -> ?) -> A
#A :-(? v A) v ?
#A :-(? v false) v A
#A :-(? v ?) v A
```

Example 3
Definition discovery with the necessitation rule.

```
?\- iel\_nec\_discover.
#A :-(A -> false) -> A
#A :-(A -> false) -> false
#A :-(A -> ?) -> A
true.
```

Unsurprisingly, the results are the same, as a consequence of axiom $A \rightarrow #A$. This final list of candidates will need to be evaluated based on their relevance to the intended semantics of IEL.

Clearly, the formula $#A :-(A -> false) -> A$ is not interesting as it would define knowing something as a contradiction that implies itself. This brings us to the second definition formula candidate.

5 Discovering the embedding of IEL and Dosen’s double negation modality in IPC

We specify a given logic (e.g., IEL or S4) by stating theorems on which the prover extended with the synthetic definition should succeed and non-theorems on which it should fail.

5.1 The discovery mechanism for IEL

We start with the axioms of Artemov and Protopopescu’s IEL system:

\[
ie_l\text{\_}th(a -> # a).
ie_l\text{\_}th(# (a->b)->(# a-> # b)).
ie_l\text{\_}th(# a -> ¬ ¬ a).
\]

Note that the axioms would be enough to specify the logic, but we also add some theorems when intuitively relevant and/or mentioned in [5], as an empirical check of their consistency with the axioms. Our Prolog code, running in less than a second, is not slowed down by this in any significant way.

\[
ie_l\text{\_}th(# (a & b) <-# (# a & # b)).
ie_l\text{\_}th(¬ # false).
ie_l\text{\_}th(¬ (# a & ¬ a)).
ie_l\text{\_}th(¬ a -> ¬ # a).
ie_l\text{\_}th(¬ ¬ (¬ a -> a)).
ie_l\text{\_}th(# a & # (a->b) -> # b).
ie_l\text{\_}th(* (a & b) <-* (* a & * b)).
ie_l\text{\_}th(# a -> * a).
ie_l\text{\_}th(# a v # b -> # (a v b)).
ie_l\text{\_}th(# p <-# # p).
ie_l\text{\_}th(* a <-* * a).
ie_l\text{\_}th(a -> * a).
\]

Again, following [5], we add our non-theorems. They act as a filtering mechanism rejecting candidate definitions that would contradict the system’s intended semantics.

\[
ie_l\text{\_}nth(# a -> a).
ie_l\text{\_}nth(# (a v b) -> # a v # b).
ie_l\text{\_}nth(# a).
ie_l\text{\_}nth(¬ (# a)).
\]
5.2 Eliminating Dosen’s double negation modality

In [2] double negation in IPC is interpreted as a "□" modality. This corresponds to our second synthetic definition, #A :- (A→false)→false, that is equivalent in IPC to #A :- ~~A. It is argued in [5] that it does not make sense as an epistemic modality, mostly because it would entail that all classical theorems are known intuitionistically.

We eliminate it by requiring the collapsing of "*" into "#" to be a non-theorem:

iel_nth(* a <-> # a).

iel_nth(# false).
iel_nth(# a).
iel_nth(~ (a)).
iel_nth(* false).

The necessitation rule in a modal logic requires that if T is a theorem then #T is also a theorem. This expresses the fact that the theorems of the logic are necessarily true, or in an epistemic context, that if T is an (intuitionistically proven) theorem, then the agent knows T. Thus, we define (implicit) facts via a Prolog rule that states that the (generic) necessity operator “#” applied to proven theorems or axioms generates new theorems.

iel_nec_th(T):-iel_th(T).
iel_nec_th(# T):-iel_th(T).

Finally, we obtain the discovery algorithm for IEL formula definitions and for IEL extended with the necessitation rule.

iel_discover:-
   backtrack_over((def synth(2,iel_th,iel_nth,D),println(D))).

iel_nec_discover:-
   backtrack_over((def synth(2,iel_nec_th,iel_nth,D),println(D))).

backtrack_over(Goal):-call(Goal),fail;true.
println(T):-numbervars(T,0,_),writeln(T).

Note the use of backtrack_over/1 to backtrack over all answers to a given goal. We run iel_discover, ready to see the surviving definition candidates.

**Example 2** Definition discovery without the necessitation rule.

?- iel_discover.
#A:- (A→false)→A
#A:- (A→false)→false
#A:- (A→ ?)→A
true.

**Example 3** Definition discovery with the necessitation rule.

?- iel_nec_discover.
#A:- (A→false)→A
#A:- (A→false)→false
#A:- (A→ ?)→A
true.

Unsurprisingly, the results are the same, as a consequence of axiom A → #A. This final list of candidates will need to be evaluated based on their relevance to the intended semantics of IEL.

Clearly, the formula #A:- (A→false)→A is not interesting as it would define knowing something as a contradiction that implies itself.

This brings us to the second definition formula candidate.
5.2 Eliminating Dosen’s double negation modality

In [2] double negation in IPC is interpreted as a “□” modality. This corresponds our second synthetic definition, #A := (A -> false) -> false, that is equivalent in IPC to #A := ¬¬A. It is argued in [5] that it does not make sense as an epistemic modality, mostly because it would entail that all classical theorems are known intuitionistically.

We eliminate it by requiring the collapsing of “*” into “#” to be a non-theorem:

iel_nth(* a <-> # a).

In fact, while known (#) implies knowable (“#” = *), it is reasonable to think, as in most modal logics, that the inverse implication does not hold.

After that, we have:

Example 4 The double negation modality is eliminated, as it collapses # and *.

?- iel_discover.
#A:- (? -> A) -> A
true.

?- iel_nec_discover.
#A:- (? -> A) -> A
true.

5.3 Knowledge as awareness?

This leaves us with the #A := (A -> ?) -> A.

Among the consequences of the fact that intuitionistic provability strictly implies classical, is that there’s plenty of room left between p and ¬¬p, where both # and * find their place, given that the following implication chain holds.

p -> #p -> *p -> ¬¬p

Let us now find an (arguably) intuitive meaning for the “?” constant in the definition. The interpretation of knowledge as awareness about truth goes back to [13]. Our final definition of intuitionistic epistemic modality as “#A := (A -> ?) -> A” suggests interpreting “?” as awareness of an agent entailed by (a proof of) A. With this in mind, one obtains an embedding of IEL in IPC via the extension

\[ KA \equiv (A \rightarrow eureka) \rightarrow A \]

where eureka is a new symbol not occurring in the language.

In line with the Brouwer-Heyting-Kolmogorov (BHK) interpretation of intuitionistic proof, we may say that an agent knows A if and only if A is validated by a proof of A that induces awareness of the agent about it.

\[ ^{8} \text{Not totally accidentally named, given the way Archimedes expressed his sudden awareness about the volume of water displaced by his immersed body.} \]
Thus knowledge of an agent, in this sense, collects facts that are proven constructively in a way that is “understood” by the agent. The consequence

\[ KA \rightarrow \neg \neg A \]

would then simply say that intuitionistic truths, that the agent is aware of, are also classically valid.

Thus, we can define our newly synthesized prover for \textbf{IEL} as follows.

\begin{verbatim}
iel_prove(P) :- prove_with_def((#A :- (A -> eureka) -> A), P).
\end{verbatim}

Interestingly, if one allows eureka to occur in the formulas of the language given as input to the prover, then it becomes (the unique) value for which we have equivalence between being known and having a proof.

?- iel_prove(#eureka <-> eureka).
true.

Similarly, it would also follow that

?- iel_prove(#eureka <-> ` ` eureka).
true.

Thus, one would need to forbid accepting it as part of the prover’s language to closely follow the intended semantics of \textbf{IEL}.

5.4 Discussion

\textit{The most significant consequence of the successful embedding of \textbf{IEL} into \textbf{IPC} via the epistemic modality definition \#A :- (A -> eureka) -> A) is that we have actually derived a theorem prover for \textbf{IEL}.} The theorem prover is implemented by the predicate \texttt{iel_prove/1} by extending a theorem prover for \textbf{IPC} with the induced definition.

As the \textbf{IPC} fragment with two variables, implication and negation has exactly 518 equivalence classes of formulas [14, 15], one would expect the construction deriving “∗” from “#” to reach a fixpoint. We can use our prover to find out when that happens.

?- iel_prove(#p <-> ` # (#p))).
false.
iel_prove(*p <-> `(#p))).
true.

Thus the fixpoint of the construction is “∗”, that we have interpreted as meaning that a proposition is knowable. Therefore, the equivalence reads reasonably as “something is knowable if and only if its negation is not knowable”. Note also that

?- iel_prove(`(*p)) -> #p).
false.

fails, by contrast to the equivalence $\Box p \equiv \neg \neg \neg p$ usual in classical modal logics.
6 Discovering an embedding of S4 without the necessitation rule

The fact that both IPC and S4 are known to be PSPACE-complete [16] means that polynomial-time translations exist between them.

In fact, Gödel’s translation from IPC to S4 (by prefixing each subformula with the □ operator) shows that the embedding of IPC into S4 can be achieved quite easily, by using purely syntactic means. However, the (very) few papers attempting the inverse translation [17, 18] rely on methods often involving intricate semantic constructions.

We will use our definition generator to identify the problem that precludes a simple embedding of S4 into IPC.

We start with the axioms of S4.

\[
\begin{align*}
s4\_th(# a \rightarrow a) . \\
s4\_th(# (a\rightarrow b) \rightarrow (# a \rightarrow # b)). \\
s4\_th(# a \rightarrow # # a). 
\end{align*}
\]

We add a few theorems.

\[
\begin{align*}
s4\_th(* a \leftrightarrow * a). \\
s4\_th(a \rightarrow * a). \\
s4\_th(# a \rightarrow * a). \\
s4\_th(# a \lor # b \rightarrow # (a \lor b)). \\
s4\_th(# (a \lor b) \rightarrow # # a # b). 
\end{align*}
\]

We add some non-theorems that ensure additional filtering.

\[
\begin{align*}
s4\_nth(# a). \\
s4\_nth(\neg (# a)). \\
s4\_nth(# false). \\
s4\_nth(* false). \\
s4\_nth(* a \rightarrow # * a). \ % \text{true only in S5} \\
s4\_nth(a \rightarrow # a). \\
s4\_nth(* a \rightarrow a). \\
s4\_nth(# a \leftrightarrow ?). \\
s4\_nth(* a \leftrightarrow ?). 
\end{align*}
\]

Like in the case of IEL we define implicit facts stating that the necessitation rule holds.

\[
\begin{align*}
s4\_nec\_th(T):=\neg s4\_th(T). \\
s4\_nec\_th(\# T):=\neg s4\_th(T). 
\end{align*}
\]

Finally we implement the definition discovery predicates and run them.

\[
\begin{align*}
s4\_discover:- \\
\quad \text{backtrack\_over}((\text{def\_synth}(2, s4\_th, s4\_nth, D), \text{println}(D))). \\
s4\_nec\_discover:- \\
\quad \text{backtrack\_over}((\text{def\_synth}(2, s4\_nec\_th, s4\_nth, D), \text{println}(D))). 
\end{align*}
\]

Example 5 The necessitation rule eliminates all simple embeddings of S4 into IPC, while a lot of definition formulas pass without it.
While we have eliminated Dosen’s double negation modality $K_p \equiv \neg \neg p$, it is significant that it came out as the only other meaningful candidate produced by our definition synthesizer. This suggests that it might be worth investigating further how a similar definition discovery mechanism as the one we have used for IEL and S4 would work for logics with multiple negation operators like equilibrium logic.

Besides the $K_p \rightarrow p$ vs. $p \rightarrow K_p$ problem a more general question is the choice of the logic supporting the epistemic operators, among logics with finite truth-value models (e.g., classical logic or equilibrium logic) or, at the limit, intuitionistic logic itself, with no such models. Arguably, this could be application dependent, as epistemic operators built on top of IPC are likely to fit better the landscape with intricate nuances of a richer set of epistemic and doxastic operators, while such operators built on top of finite-valued intermediate logics would benefit from simpler decision procedures and faster evaluation mechanisms.

8 Conclusions

We have devised a general mechanism for synthesizing definitions that extend a given logic system endowed with a theorem prover. The set of theorems on which the extended prover should succeed and the set of non-theorems on which it should fail can be seen as a declarative specification of the extended system. Success of the approach on embedding the IEL system in IPC and failure on trying to embed S4 has revealed the individual role of the axioms, theorems and rules that specify a given logic system and their interaction with the necessitation rule.

Given its generality, our definition generation technique can be applied also to epistemic or modal logic axiom systems to find out if they have interesting embeddings in ASP and superintuitionistic logics for which high quality solvers or theorem provers exist. Our program synthesis process, when the embedding succeeds, provides a way to automate the exploration of a new logic system with help of its derived theorem prover and facilitates the work of the human logician to validate or invalidate the intuitions behind it.

Acknowledgement

We thank the participants to the EELP’2019 workshop for their constructive suggestions and comments and the anonymous reviewers of LOPSTR’2020 for their constructive comments and suggestions.

References


Among the definitions succeeding without passing the necessity rule test, one might want to pick $\#A := ? \& A$ as an approximation of the S4 “□” operator. In this case “?” would simply state that “the IPC prover is sound and complete”. Still, given the failure of the necessitation rule, the resulting logic is missing a key aspect of the intended meaning of S4-provability.

7 Related work

Program synthesis techniques have been around in logic programming with the advent of Inductive Logic Programming [19], but the idea of learning Prolog programs from positive and negative examples goes back to [20]. Our definition synthesizer fits in this paradigm, with focus on the use of a theorem prover of a decidable logic (IPC) filtering formulas provided by a definition generator through theorems as positive examples and non-theorems as negative examples. The means we use for our definition synthesis are in fact as simple as those described in [20]. The strength of our approach comes from the use of a theorem prover that efficiently validates or rejects definition candidates. The idea to use the new constant “?” in our synthesizer is inspired by proofs that some fragments of IPC reduced to two variables have a (small) finite number of equivalence classes [14, 15] as well as by the introduction of new variables, in work on polynomial embeddings of S4 into IPC [17, 18].

We refer to [5] for a thorough discussion of the merits of IEL compared to epistemic logics following closely classical modal logic, but the central idea about using intuitionistic logic is that of belief and knowledge as the product of verification. Our embedding of IEL in IPC can be seen as a simplified view of this process through a generic “awareness of an agent” concept in line with [13].

In [1] the concept of epistemic specifications is introduced that support expressing knowledge and belief in an Answer Set Programming framework. Interestingly, refinements of this work like [21] and [3] discuss difficulties related to expressing an assumption like $p \rightarrow Kp$ in terms of ASP-based epistemic operators.

Equilibrium logic [4] gives a semantics to Answer Set programs by extending the 3-valued intermediate logic of here-and-there HT with Nelson’s constructive strong negation. In [22] a 5-valued truth-table semantics for equilibrium logic is given. In [23] (and several other papers) epistemic extensions of equilibrium logic [4] are proposed, in which $Kp \rightarrow p$. By contrast to “alethic inspired” epistemic logics postulating $Kp \rightarrow p$ we closely follow the $p \rightarrow Kp$ view on which [5] is centered.
While we have eliminated Dosen’s double negation modality [6] as an epistemic operator $Kp \equiv \neg\neg p$, it is significant that it came out as the only other meaningful candidate produced by our definition synthesizer.

This suggests that it might be worth investigating further how a similar definition discovery mechanism as the one we have used for IEL and S4 would work for logics with multiple negation operators like equilibrium logic.

Besides the $Kp \rightarrow p$ vs. $p \rightarrow KP$ problem a more general question is the choice of the logic supporting the epistemic operators, among logics with finite truth-value models (e.g., classical logic or equilibrium logic) or, at the limit, intuitionistic logic itself, with no such models. Arguably, this could be application dependent, as epistemic operators built on top of IPC are likely to fit better the landscape with intricate nuances of a richer set of epistemic and doxastic operators, while such operators built on top of finite-valued intermediate logics would benefit from simpler decision procedures and faster evaluation mechanisms.

8 Conclusions

We have devised a general mechanism for synthesizing definitions that extend a given logic system endowed with a theorem prover. The set of theorems on which the extended prover should succeed and the set of non-theorems on which it should fail can be seen as a declarative specification of the extended system. Success of the approach on embedding the IEL system in IPC and failure on trying to embed S4 has revealed the individual role of the axioms, theorems and rules that specify a given logic system and their interaction with the necessitation rule.

Given its generality, our definition generation technique can be applied also to epistemic or modal logic axiom systems to find out if they have interesting embeddings in ASP and superintuitionistic logics for which high quality solvers or theorem provers exist. Our program synthesis process, when the embedding succeeds, provides a way to automate the exploration of a new logic system with help of its derived theorem prover and facilitates the work of the human logician to validate or invalidate the intuitions behind it.

Acknowledgement

We thank the participants to the EELP’2019 workshop\(^9\) for their constructive suggestions and comments and the anonymous reviewers of LOPSTR’2020 for their constructive comments and suggestions.

References


\(^9\) A forum with no formal proceedings but insightful presentations and lively discussions on epistemic extensions of logic programming systems.


