16th International Workshop on Termination

WST 2018, July 18–19, 2018, Oxford, UK

Edited by

Salvador Lucas

WST 2018 Proceedings

Contents

16th International Workshop on Termination (WST 2018).
Editor: Salvador Lucas

Tool papers

16th International Workshop on Termination (WST 2018).
Editor: Salvador Lucas

Preface

This report contains the proceedings of the *16th International Workshop on Termination (WST 2018)*, which was held in Oxford, United Kingdom, during July 18–19, 2018. The termination workshops traditionally bring together, in an informal setting, researchers interested in all aspects of termination, whether this interest be practical or theoretical, primary or derived. The workshop also provides a ground for cross-fertilization of ideas from the different communities interested in termination (e.g., working on computational mechanisms, programming languages, software engineering, constraint solving, etc.). The friendly atmosphere enables fruitful exchanges leading to joint research and subsequent publications.

Previous termination workshops were organized in St. Andrews (1993), La Bresse (1995), Ede (1997), Dagstuhl (1999), Utrecht (2001), Valencia (2003), Aachen (2004), Seattle (2006), Paris (2007), Leipzig (2009), Edinburgh (2010), Obergurgl (2012), Bertinoro (2013), Vienna (2014), and Obergurgl (2016). This time, as in the previous editions of the Federated Logic Conference (FLoC) since 2006, WST 2018 is part of FLoC 2018. In particular, WST was affiliated to several FLoC conferences:

- 9th International Joint Conference on Automated Reasoning (IJCAR), as *primary hosting conference*,
- 30th International Conference on Computer Aided Verification (CAV),
- 23rd International Symposium on Formal Methods (FM),
- 3rd International Conference on Formal Structures for Computation and Deduction (FSCD),
- 34th International Conference on Logic Programming (ICLP),
- 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), and
- 21st International Conference on Theory and Applications of Satisfiability Testing (SAT).

I'm grateful to the PC chairs and Workshop chairs of these FLoC 2018 conferences for supporting WST 2018 affiliation.

The WST 2018 program included an invited talk by *James Worrell* on *Termination Checking and Invariant Synthesis for Affine Programs* and another invited talk by *Akihisa Yamada* entitled *Towards a unified method for termination*. The corresponding papers are included in the proceedings. WST 2018 received 14 submissions. After light reviewing and careful deliberations the program committee decided to accept all submissions. The 14 contributions are contained in the proceedings. Furthermore, the proceedings also contain short descriptions of several tools that participated in the 2018 Termination and Complexity Competition (TERCOMP). This competition ran live during FLoC 2018 as part of the FLoC Olympic Games and the results are available at `http://www.termination-portal.org/`.

Several persons helped to make WST 2018 a success. I'm specially grateful to the members of the program committee, the external reviewers, the members of the TERMCOMP Steering Committee, and also to the organizing committee of FLoC 2018 for their support.

Valencia, July 2018 *Salvador Lucas*

16th International Workshop on Termination (WST 2018).
Editor: Salvador Lucas

Organization

WST Program Committee

Cristina Borralleras	U. de Vic	
Ugo Dal Lago	U. degli Studi di Bologna	
Carsten Fuhs	Birkbeck, U. of London	
Samir Genaim	U. Complutense de Madrid	
Jürgen Giesl	RWTH Aachen	
Raúl Gutiérrez	U. Politècnica de València	
Keiichirou Kusakari	Gifu University	
Salvador Lucas	U. Politècnica de València	(chair)
Fred Mesnard	U. de La Réunion	
Aart Middeldorp	U. of Innsbruck	
Albert Rubio	U. Politècnica de Catalunya	
René Thiemann	U. of Innsbruck	
Caterina Urban	ETH Zürich	

External reviewers

Ralph Bottesch	Maximilian Haslbeck

TermComp Steering Committee

Jürgen Giesl	RWTH Aachen, Germany	
Albert Rubio	U. Politècnica de Catalunya, Spain	(chair)
Christian Sternagel	U. of Innsbruck, Austria	
Johannes Waldmann	HTWK Leipzig, Germany	
Akihisa Yamada	National Institute of Informatics, Japan	

16th International Workshop on Termination (WST 2018).
Editor: Salvador Lucas

Invited papers

16th International Workshop on Termination (WST 2018).
Editor: Salvador Lucas

Termination Checking and Invariant Synthesis for Affine Programs

James Worrell

Department of Computer Science, University of Oxford, UK
james.worrell@cs.ox.ac.uk

Abstract

Invariants are one of the most fundamental and useful notions in automated verification, dynamical systems, and control theory. With regard to program termination, invariants can be used both as certificates of non-termination and to support proofs of termination. In light of this, automated invariant synthesis has long been (and remains) a lively topic of research in program analysis. Current research on invariant generation employs an eclectic array of techniques, including abductive inference, abstract interpretation, constraint solving, interpolation, and machine learning.

In this talk we give a select overview of previous work on invariant synthesis, focussing on a simple class of programs with affine updates. Our starting point is a classical algorithm of Michael Karr for discovering affine invariants [4]. We proceed to mention a large number of variations and generalisations of this work, much of it stemming from [2, 6, 5], to the case of polyhedral and algebraic invariants. In the main body of the talk we describe two recent procedures to compute the strongest algebraic invariant of an affine program [3] and to synthesise semi-algebraic invariants for proving non-termination of linear loops [1]. We conclude by surveying a number of open problems and challenges for future work.

This is joint work with Shaull Almagor, Dmitry Chistikov, Ehud Hrushovski, Amaury Pouly, and Joël Ouaknine.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Linear Loops, Affine Programs, Polyhedra, Algebraic Sets, Semi-Algebraic Sets

References

1 S. Almagor, D. Chistikov, J. Ouaknine, and J. Worrell. O-minimal invariants for linear loops. In *Proceedings of ICALP 2018*, volume 107 of *LIPIcs*, pages 114:1–114:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

2 P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of POPL 1978*, pages 84–96. ACM, 1978.

3 E. Hrushovski, J. Ouaknine, A. Pouly, and J. Worrell. Polynomial invariants for affine programs. In *Proceedings of LICS 2018*, pages 530–539. ACM, 2018.

4 M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

5 M. Müller-Olm and H. Seidl. A note on karr's algorithm. In *Proceedings of ICALP 2004*, volume 3142 of *LNCS*, pages 1016–1028. Springer, 2004.

6 S. Sankaranarayanan, H. Sipma, and Z. Manna. Non-linear loop invariant generation using gröbner bases. In *Proceedings of POPL 2004*, pages 318–329. ACM, 2004.

© James Worrell;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

Towards a Unified Method for Termination*

Akihisa Yamada

National Institute of Informatics, Tokyo, Japan

Abstract

The question of how to ensure programs terminate has been for decades attracting remarkable attention of computer scientists, resulting in a great number of techniques for proving termination of term rewriting and other models of computation. Nowadays it has become hard for new-comers to come up with new termination techniques/tools, since there are so many to learn/implement before inventing a new one. In this talk, I present my past and on-going work towards unified method for termination, that allow one to learn/implement a single idea and obtain many well-known techniques as instances.

1 Preliminaries

An *abstract reduction system (ARS)*, following Klop [10], consists of a set T and a family $\{\to_\rho\}_{\rho\in\mathcal{R}}$ of binary relations over T. Our interest is proving that $\to_\mathcal{R} := \bigcup_{\rho\in\mathcal{R}} \to_\rho$ is *terminating*, i.e., there is no infinite sequence of form $s_1 \to_\mathcal{R} s_2 \to_\mathcal{R} \cdots$.

Termination can be incrementally proved by a function $[\![\cdot]\!] : T \to A$ to a well-founded ordered set $\langle A, \succsim, \succ\rangle$. Let us define $[_{(\succsim)}] := \{\rho \mid s \to_\rho t \implies [\![s]\!] \ {}_{(\succsim)}\ [\![t]\!]\}$.

▶ Proposition 1. If $\mathcal{R} \subseteq [\succsim]$, then $\to_\mathcal{R}$ is terminating if $\to_{\mathcal{R}\setminus[\succ]}$ is. ◀

In term rewriting, *reduction orders* are a famous approach for termination, which use identity $[\![\cdot]\!]$ and impose conditions on orderings so that $[\succ] = \succ$. To minimize definitions, let us formulate only *interpretation*-based approach.

▶ **Definition 2** (sorted terms and term rewriting). A *sorted signature* $\mathcal{F}$ consists of a set $\mathcal{S}_\mathcal{F}$ of sorts and a family $\{\mathcal{F}_\tau\}_{\tau\in\mathcal{S}_\mathcal{F}^*\times\mathcal{S}_\mathcal{F}}$ of function symbols. $\mathcal{F}$ is *single sorted* if $\mathcal{S}_\mathcal{F}$ is singleton. The *arity* of $f \in \mathcal{F}_{\vec\sigma,\sigma}$ is the length of $\vec\sigma$. Given a family $\{\mathcal{V}_\sigma\}_{\sigma\in\mathcal{S}_\mathcal{F}}$ of variables, the set $\mathcal{T}_\sigma(\mathcal{F},\mathcal{V})$ of *terms* of sort σ are defined as usual. A *term rewrite system (TRS)* is a set $\mathcal{R}$, where each $\rho \in \mathcal{R}$ is a pair $\langle l, r\rangle$ of (single-sorted) terms with $l \notin \mathcal{V}$ and $\mathsf{Var}(l) \supseteq \mathsf{Var}(r)$. The ARSs $\to_\rho$, $\xrightarrow[\rho]{\epsilon}$ and $\xrightarrow[\rho]{>\epsilon}$, are defined as usual. ◀

▶ **Definition 3** (algebras). For a sorted signature $\mathcal{F}$, an *$\mathcal{F}$-algebra* $[\![\cdot]\!]$ assigns each sort $\sigma \in \mathcal{S}_\mathcal{F}$ a set $[\![\sigma]\!]$ and each symbol $f \in \mathcal{F}_{[\sigma_1,\ldots,\sigma_n],\sigma}$ a mapping $[\![f]\!] : [\![\sigma_1]\!] \times \cdots \times [\![\sigma_n]\!] \to [\![\sigma]\!]$. The interpretation $[\![s]\!]\alpha \in [\![\sigma]\!]$ of term $s \in \mathcal{T}_\sigma(\mathcal{F},\mathcal{V})$ under assignment α is defined as usual. An *$\mathcal{F}$-logic* is an $\mathcal{F}$-algebra with a special sort $\mathtt{bool} \in \mathcal{S}_\mathcal{F}$ and standard logic symbols $\wedge, \vee, \Rightarrow \in \mathcal{F}_{[\mathtt{bool},\mathtt{bool}],\mathtt{bool}}$ etc. with expected interpretations. We say $\phi \in \mathcal{T}_\mathtt{bool}(\mathcal{F},\mathcal{V})$ is *valid*, written $[\![\phi]\!]$, if $[\![\phi]\!]\alpha = \textsc{True}$ for any assignment α. ◀

▶ **Definition 4** (ordered algebras). An *ordered $\mathcal{F}$-algebra* is a logic $[\![\cdot]\!]$, where the domain is quasi-ordered and the signature is $\mathcal{F}$ extended with logic symbols $\geq, > \in \mathcal{F}_{[\sigma,\sigma'],\mathtt{bool}}$, all interpreted as expected. We say $[\![\cdot]\!]$ is

- *well-founded* if $>$ is well-founded,
- *(weakly) monotone* if $[\![f]\!]$ is monotone w.r.t. ${}_{(\geq)}$ in every argument for every $f \in \mathcal{F}$, and

* The author is supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST.

© Akihisa Yamada;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

- *(weakly) simple* if $[\![f]\!](a_1,\dots,a_n)\ {}_{(}\!\geq_{)}\ a_i$ for every $f \in \mathcal{F}$ and i. ◀

Let us write $s\ [{}_{(}\!\geq_{)}]\ t :\iff [\![s\ {}_{(}\!\geq_{)}\ t]\!]$. A reduction order can be characterized by $[>]$ of a well-founded monotone algebra, a *simplifiaction order* [2] by a simple monotone algebra, and *reduction pair* [1] by a well-founded weakly monotone algebra in the same manner.

2 Monotone WPO

Here we present the basic version of the *weighted path order (WPO)* [14].

▶ **Definition 5** (monotone WPO)**.** Let $[\![\cdot]\!]$ be a well-founded $\mathcal{F}$-algebra, and $\succsim$ a well-founded quasi-order on $\mathcal{F}$. We define relations ${}_{(}\!\succsim_{)}\mathsf{WPO}$ as follows: $s = f(s_1,\dots,s_n)\ {}_{(}\!\succsim_{)}\mathsf{WPO}\ t$ iff

1. $[\![s > t]\!]$, or
2. $[\![s \geq t]\!]$ and
   - **a.** $\exists i \in \{1,\dots,n\}.\ s_i \succsim\!\!\!\succsim_{\mathsf{WPO}} t$, or
   - **b.** $t = g(t_1,\dots,t_m)$, $\forall j \in \{1,\dots,m\}.\ s \succ_{\mathsf{WPO}} t_j$ and either
     - **i.** $f \succ g$ or
     - **ii.** $f \sim g$ and $\langle s_1,\dots,s_n\rangle\ {}_{(}\!\succsim_{)}^{\mathsf{lex}}\mathsf{WPO}\ \langle t_1,\dots,t_m\rangle$. ◀

▶ **Theorem 6.** *WPO is a simplification order if $[\![\cdot]\!]$ is weakly monotone and weakly simple.* ◀

- Dropping lines **1** and **2** results in LPO. The same effect can be achieved by choosing a trivial (singleton-carrier) algebra as $[\![\cdot]\!]$ or by interpretation $[\![f]\!](a_1,\dots,a_n) = \max\{a_1,\dots,a_n\}$ on a usual carrier like $\mathbb{N}$. Hence, WPO subsumes LPO.
- Dropping lines **a** and **b** results in GKBO, but for the resulting order to be well-founded, it is required to strengthen the weak simplicity condition to strict simplicity. When this condition is satisfied, WPO coincides with GKBO.
- The definition of KBO has a similar structure as Definition 5, with a particular condition in **a** (which can in fact be simplified). Similar to GKBO, KBO requires the "admissibility" condition on $[\![\cdot]\!]$, and under this condition, WPO coincides with KBO. For a general and detailed account, see [13, Chapter 3].
- As a reduction (simplification) order, WPO also subsumes monotone interpretations over totally ordered carrier, in the sense that $[>] \subseteq \succ_{\mathsf{WPO}}$. The side condition of Theorem 6 is known to be satisfied [15]. Further, WPO can be seen as a stretch of [16, Proposition 12], which indicates that a weakly simple and weakly monotone well-founded algebra can be extended to a simplification order.

3 Weakly Monotone WPO

The *dependency pair method* [1] reduces the termination of $\xrightarrow[\mathcal{R}]{}$ to the *finiteness* of *DP problem* $\langle \mathsf{DP}(\mathcal{R}), \mathcal{R}\rangle$, i.e., the termination of ARS[1] $\left\{\xrightarrow[\rho]{\epsilon} \circ \xrightarrow[\mathcal{R}]{>\epsilon\,*}\right\}_{\rho\in\mathsf{DP}(\mathcal{R})}$, where $\mathsf{DP}(\mathcal{R}) := \{\langle l, r\rangle \mid \langle l, C[r]\rangle \in \mathcal{R}, \mathsf{root}(r) \in \mathcal{D}\}$ for $\mathcal{D}$ consisting of the root symbols of the left-hand sides of $\mathcal{R}$. A big merit of this reduction is that the monotonicity of reduction orders can be relaxed:

▶ **Theorem 7** ([7, 5])**.** *Let $\langle \succsim, \succ\rangle$ be a reduction pair such that $\mathcal{R} \cup \mathcal{P} \subseteq \succsim$. Then $\xrightarrow[\mathcal{P}]{\epsilon} \circ \xrightarrow[\mathcal{R}]{>\epsilon\,*}$ is terminating if $\xrightarrow[\mathcal{P}\setminus\succ]{\epsilon} \circ \xrightarrow[\mathcal{R}]{>\epsilon\,*}$ is.* ◀

[1] More precisely, $\xrightarrow[\rho]{\epsilon}$ here is restricted to $\xrightarrow[\mathcal{R}]{>\epsilon}$-terminating terms.

Although WPO is a simplification order, its well-foundedness is directly proved by an inductive argument inspired by Jouannaud and Rubio [9]. There, the key is to ensure $[\![s \geq s_i]\!]$ for those s_i's that are used in recursive comparison in lines **a** and **ii**. In fact, WPO is still well-founded if we restrict these recursively compared arguments to those which $[\![s \geq s_i]\!]$ is ensured. This has a similar effect as *argument filtering*, but we can additionally take the weights of dropped arguments into account.

▶ **Definition 8** (weakly monotone WPO). Let π be a mapping that assigns each n-ary symbol $f \in \mathcal{F}$ a subset $\pi(f) \subseteq \{1, \dots, n\}$ of its argument positions. Abusing notation, we see $\pi(f)$ also as an index-filtering operation over lists. We refine WPO as follows: $s = f(s_1, \dots, s_n) \mathrel{(\gtrsim)_{\mathsf{WPO}}} t$ iff

1. $[\![s > t]\!]$, or
2. $[\![s \geq t]\!]$ and
   - **a.** $\exists i \in \pi(f).\ s_i \gtrsim_{\mathsf{WPO}} t$, or
   - **b.** $t = g(t_1, \dots, t_m)$, $\forall j \in \pi(g).\ s \succ_{\mathsf{WPO}} t_j$ and either
     - **i.** $f \succ g$ or
     - **ii.** $f \sim g$ and $\pi(f)[s_1, \dots, s_n] \mathrel{(\gtrsim)^{\mathsf{lex}}_{\mathsf{WPO}}} \pi(g)[t_1, \dots, t_m]$. ◀

Since now some arguments may be dropped in the comparison of line **ii**, $\succ_{\mathsf{WPO}}$ is not closed under context anymore, but it is no problem in the DP framework.

▶ **Theorem 9** ([14]). *WPO forms a reduction pair if $[\![\cdot]\!]$ is weakly monotone and π-simple: $[\![f]\!](a_1, \dots, a_n) \geq a_i$ whenever $i \in \pi(f)$.* ◀

With some small refinements [14, Section 4.2], one can get $[\geq] \subseteq \gtrsim_{\mathsf{WPO}}$ and $[>] \subseteq \succ_{\mathsf{WPO}}$ by setting $\pi(f) := \emptyset$ and $\gtrsim := \mathcal{F} \times \mathcal{F}$. In this sense, weakly monotone WPO subsumes weakly monotone interpretations.

4 Non-Monotone WPO

Now we further generalize WPO so that non-monotone algebras can be used. Such algebras are still useful for proving innermost termination [6], and probably full termination [3].

▶ **Definition 10.** Let μ be a mapping that assigns each n-ary symbol f and $i \in \{1, \dots, n\}$ a subset of $\{\geq, \leq\}$. We say an ordered algebra $[\![\cdot]\!]$ is *μ-monotone* if $a_i \geq a'_i$ implies $[\![f]\!](\dots, a_i, \dots) \sqsupseteq [\![f]\!](\dots, a'_i, \dots)$ whenever $\sqsupseteq \in \mu(f, i)$. For a TRS $\mathcal{R}$, we define set $U_{\mathcal{R},\mu}(s)$ of pairs of terms so that $U_{\mathcal{R},\mu}(f(s_1, \dots, s_n))$ is a superset of

1. $\{\langle l, r\rangle\} \cup U_{\mathcal{R},\mu}(r)$ for every $\langle l, r\rangle \in \mathcal{R}$ with $\mathsf{root}(l) = f$,
2. $U_{\mathcal{R},\mu}(s_i)$ if $\sqsubset \notin \mu(f, i)$, and
3. $U_{\mathcal{R},\mu}(s_i)^{-1}$ if $\sqsupset \notin \mu(f, i)$.

▶ **Theorem 11** ([6]). *Let $[\![\cdot]\!]$ be a well-founded μ-monotone algebra such that $\mathcal{P} \cup \left(\bigcup_{\langle l,r\rangle \in \mathcal{P}} U_{\mathcal{R},\mu}(r)\right) \subseteq [\geq]$. Then*[2] $\xrightarrow[\mathcal{P}]{\epsilon} \circ \xrightarrow[\mathcal{R}]{>\epsilon}!$ *is terminating if* $\xrightarrow[\mathcal{P}\setminus[>]]{\epsilon} \circ \xrightarrow[\mathcal{R}]{>\epsilon}!$ *is.* ◀

WPO is already applicable to μ-monotone algebras, in the following sense:

▶ **Theorem 12.** *WPO forms a well-founded μ-monotone term algebra, if*

1. *$[\![\cdot]\!]$ is μ-monotone and π-simple, and*

[2] More precisely, $\xrightarrow[\rho]{\epsilon}$ here is restricted to $\xrightarrow[\mathcal{R}]{>\epsilon}$-normal forms.

2. $\mu(f,i) = \{\geq\}$ *for every* $f \in \mathcal{F}$ *and* $i \in \pi(f)$.

Proof Sketch. Only μ-monotonicity has to be proved. The interesting case is $\leq \in \mu(f,i)$. Then $s_i \succsim_{\mathsf{WPO}} s'_i$ implies $[\![f(\dots,s_i,\dots) \leq f(\dots,s'_i,\dots)]\!]$. If this is not "strict", then it is easy to see that case **ii** is applied, and compared argument lists are identical. ◀

It is tempting to exploit anti-monotonicity by comparing some arguments in line **ii** in reverse direction. Then μ-monotonicity will still be preserved, but unfortunately, it turns out that well-foundedness and even *non-inifinitesimality* [6] will be broken.

5 Constrained WPO

WPO combines the syntactic termination argument of path orders and the semantic termination argument of algebraic interpretations. Hence, we expect WPO to be useful for term rewriting combined with algebraic semantics [4, 12].

▶ **Definition 13.** Let $\mathcal{F}$ be a signature, partitioned into $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$. We fix the semantics of $\mathcal{B}$ by a $\mathcal{B}$-logic $[\![\cdot]\!]_{\mathcal{B}}$, and assume a terminating $\mathcal{B}$-TRS $\mathcal{S}$ such that[3] $\langle l,r\rangle \in \mathcal{S}$ implies $[\![l]\!]_{\mathcal{B}} = [\![r]\!]_{\mathcal{B}}$. A *constrained TRS* $\mathcal{R}$ is a set where each $\rho \in \mathcal{R}$ is a triple $\langle l,\phi,r\rangle$, such that[4] $l,r \in \mathcal{T}(\mathcal{F},\mathcal{V})$, $\phi \in \mathcal{T}_{\mathsf{bool}}(\mathcal{B},\mathcal{V})$, $\mathsf{root}(l) \in \mathcal{D}$ and $\mathsf{Var}(l) \cup \mathsf{Var}(\phi) \supseteq \mathsf{Var}(r)$. The relation $\xrightarrow[\rho]{\epsilon}$ is characterized by $l\theta \xrightarrow[\rho]{\epsilon} r\theta$ where $\phi\theta \in \mathcal{T}_{\mathsf{bool}}(\mathcal{B},\emptyset)$ is valid, and extended to $\xrightarrow[\rho]{>\epsilon}$ and $\xrightarrow[\rho]{}$ as usual. The termination problem is a relative termination problem: $\xrightarrow[\mathcal{R}]{}/\xrightarrow[\mathcal{S}]{}$. ◀

Kop [11] imported the dependency pair method to (logically) constrained TRSs.

▶ **Theorem 14.** *Constrained TRS* $\mathcal{R}$ *is terminating if* $\mathcal{S}$ *is non-duplicating and* $\xrightarrow[\mathsf{DP}(\mathcal{R})]{\epsilon} \circ \xrightarrow[\mathcal{R}\cup\mathcal{S}]{>\epsilon}{}^*$ *is terminating, where* $\mathsf{DP}(\mathcal{R}) := \{\, \langle l,\phi,r\rangle \mid \langle l,\phi,C[r]\rangle \in \mathcal{R}, \mathsf{root}(r) \in \mathcal{D} \,\}$.

Proof. Let $\overline{\mathcal{R}} := \{\langle l\theta, r\theta\rangle \mid \langle l,\phi,r\rangle \in \mathcal{R}, [\![\phi\theta]\!]_{\mathcal{B}}\}$. We have $\xrightarrow[\overline{\mathcal{R}}]{>\epsilon} = \xrightarrow[\mathcal{R}]{>\epsilon}$ and $\xrightarrow[\mathsf{DP}(\overline{\mathcal{R}})]{\epsilon} = \xrightarrow[\mathsf{DP}(\mathcal{R})]{\epsilon}$. Iborra et al. [8] shows that $\xrightarrow[\overline{\mathcal{R}}]{}/\xrightarrow[\mathcal{S}]{}$ is terminating if $\xrightarrow[\mathsf{DP}(\overline{\mathcal{R}})]{\epsilon} \circ \xrightarrow[\mathcal{S}\cup\overline{\mathcal{R}}]{>\epsilon}{}^*$ is. ◀

The key ingredient of constrained TRSs is, obviously, constraints. Hence it is of essential importance to exploit information from constraints.

▶ **Definition 15** (constrained WPO). For $\phi \in \mathcal{T}_{\mathsf{bool}}(\mathcal{B},\mathcal{V})$, we define relations ${}_{(\sim)}\succsim_{\mathsf{WPO}[\phi]}$ as follows: $s = f(s_1,\dots,s_n)$ ${}_{(\sim)}\succsim_{\mathsf{WPO}[\phi]}$ t iff

1. $[\![\phi \Rightarrow s > t]\!]$, or
2. $[\![\phi \Rightarrow s \geq t]\!]$ and
   - **a.** $\exists i \in \pi(f).\ s_i \succsim_{\mathsf{WPO}[\phi]} t$, or
   - **b.** $t = g(t_1,\dots,t_m)$, $\forall j \in \pi(g).\ s \succ_{\mathsf{WPO}[\phi]} t_j$ and either
     - **i.** $f \succ g$ or
     - **ii.** $f \sim g$ and $\pi(f)[s_1,\dots,s_n]$ ${}_{(\sim)}\succsim^{\mathsf{lex}}_{\mathsf{WPO}[\phi]}$ $\pi(g)[t_1,\dots,t_m]$. ◀

As one naturally expects, $[\![\cdot]\!]$ used above should *respect* $[\![\cdot]\!]_{\mathcal{B}}$ used for inducing rewrite relation, i.e., $[\![s]\!] = [\![s]\!]_{\mathcal{B}}$ for any $s \in \mathcal{T}(\mathcal{B},\mathcal{V})$. Under this assumption, we may write $[{}_{(\sim)}\succsim_{\mathsf{WPO}}] := \{\langle l,\phi,r\rangle \mid l\ {}_{(\sim)}\succsim_{\mathsf{WPO}[\phi]}\ r\}$; this notation, corresponding to the notation in Proposition 1, is justified by the following fact:

[3] Kop and Nishida [12] assumes $\mathcal{B}$ to contain all values of $[\![\cdot]\!]_{\mathcal{B}}$, and fixes $\mathcal{S}$ to be the *calculation* step. For the purpose of this talk, we do not need these assumptions.

[4] Here we ignore the sensible assumption that l and r should be of the same sort.

▶ **Lemma 16.** *If* $[\![\cdot]\!]$ *respects* $[\![\cdot]\!]_{\mathcal{B}}$*, then* $\langle l, \phi, r\rangle \in [{}_{(}\succsim_{)}\mathsf{WPO}]$ *implies* $\xrightarrow[\langle l,\phi,r\rangle]{\epsilon} \subseteq {}_{(}\succsim_{)}\mathsf{WPO}$.

Proof Sketch. Consider $l\theta \xrightarrow[\langle l,\phi,r\rangle]{\epsilon} r\theta$, so $[\![\phi\theta]\!]_{\mathcal{B}}$. Let us assume that $l \succ_{\mathsf{WPO}\,[\phi]} r$ is derived from case **1**. Then we have $[\![\phi \Rightarrow l > r]\!]$, hence $[\![\phi\theta \Rightarrow l\theta > r\theta]\!]$. Since $[\![\phi\theta]\!] = [\![\phi\theta]\!]_{\mathcal{B}} = \textsc{True}$, we get $[\![l\theta > r\theta]\!]$, deriving $l\theta \succ_{\mathsf{WPO}} r\theta$. Other cases go as well. ◀

Since $[\![\cdot]\!]_{\mathcal{B}}$ is often the integer arithmetic, which is not monotone, we cannot assume (weak) monotonicity on $[\![\cdot]\!]$. So we now restrict our interest to innermost termination. We extend $U_{\mathcal{R},\mu}(\mathcal{P})$ for constrained TRSs by replacing item **1** of Definition 10 with

1. $\{\langle l, \phi, r\rangle\} \cup U_{\mathcal{R},\mu}(r)$ for every $\langle l, \phi, r\rangle \in \mathcal{R}$ with $\mathsf{root}(l) = f$.

▶ **Theorem 17.** *Let* $[\![\cdot]\!]$ *respect* $[\![\cdot]\!]_{\mathcal{B}}$*, and* $\mathcal{P} \cup \left(\bigcup_{\langle l,\phi,r\rangle\in\mathcal{P}} U_{\mathcal{R}\cup\mathcal{S},\mu}(r)\right) \subseteq [\succsim_{\mathsf{WPO}}]$*. Then* $\xrightarrow[\mathcal{P}]{\epsilon} \circ \xrightarrow[\mathcal{R}\cup\mathcal{S}]{>\epsilon}{}^{!}$ *is terminating if* $\xrightarrow[\mathcal{P}\setminus[\succ_{\mathsf{WPO}}]]{\epsilon} \circ \xrightarrow[\mathcal{R}\cup\mathcal{S}]{>\epsilon}{}^{!}$ *is.*

Proof Sketch. We take the same approach as Theorem 14 to reduce to Theorem 11. ◀

References

1 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theor. Compt. Sci.*, 236(1-2):133–178, 2000.
2 N. Dershowitz. Orderings for term-rewriting systems. *Theor. Compt. Sci.*, 17(3):279–301, 1982.
3 C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. Maximal termination. In *RTA 2008*, volume 5117 of *LNCS*, pages 110–125, 2008.
4 Y. Furuichi, N. Nishida, M. Sakai, K. Kusakari, and T. Sakabe. Approach to procedural-program verification based on implicit induction of constrained term rewriting systems. *IPSJ Transactions on Programming*, 1(2):100–121, 2008.
5 J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *FroCoS 2005*, volume 3717 of *LNAI*, pages 216–231, 2005.
6 J. Giesl, R. Thiemann, S. Swiderski, and P. Schneider-Kamp. Proving termination by bounded increase. In *CADE-21*, pages 443–459, 2007.
7 N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Inf. Comput.*, 199(1,2):172–199, 2005.
8 J. Iborra, N. Nishida, G. Vidal, and A. Yamada. Relative termination via dependency pairs. *J. Autom. Reasoning*, 58(3):391–411, 2017.
9 J.-P. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *LICS 1999*, pages 402–411, 1999.
10 J.W. Klop. *Term rewriting systems*, volume 2 of *Handbook of Logic in Computer Science*. Oxford University Press, 1992.
11 C. Kop. Termination of LCTRSs. In *WST 2013*, pages 59–63, 2013.
12 C. Kop and N. Nishida. Term rewriting with logical constraints. In *FroCoS 2013*, pages 343–358, 2013.
13 A. Yamada. *The weighted path order for termination of term rewriting*. PhD thesis, Nagoya University, 2014.
14 A. Yamada, K. Kusakari, and T. Sakabe. A unified order for termination proving. *Sci. Comput. Program.*, 111:110–134, 2015.
15 H. Zantema. Termination of term rewriting: interpretation and type elimination. *J. Symb. Comput.*, 17(1):23–50, 1994.
16 H. Zantema. The termination hierarchy for term rewriting. *Appl. Algebr. Eng. Comm. Compt.*, 12:3–19, 2001.

Regular papers

16th International Workshop on Termination (WST 2018).
Editor: Salvador Lucas

Objective and Subjective Specifications

Eric C.R. Hehner

Department of Computer Science, University of Toronto,
hehner@cs.utoronto.ca

Abstract

We examine specifications for dependence on the agent that performs them. We look at the consequences for the Church-Turing Thesis and for the Halting Problem. The full paper is at www.cs.utoronto.ca/~hehner/halting.html.

Keywords and phrases Church-Turing Thesis, Halting problem

© Eric Hehner;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

Termination of $\lambda\Pi$ modulo rewriting using the size-change principle (work in progress)

Frédéric Blanqui[1,2] and Guillaume Genestier[1,2,3]

1 **LSV, ENS Paris-Saclay, CNRS, Université Paris-Saclay**
2 **Inria**
3 **MINES ParisTech, PSL University**

Abstract

The Size-Change Termination principle was first introduced to study the termination of first-order functional programs. In this work, we show that it can also be used to study the termination of higher-order rewriting in a system of dependent types extending LF.

Keywords and phrases Termination, Higher-Order Rewriting, Dependent Types, Lambda-Calculus.

1 Introduction

The Size-Change Termination principle (SCT) was first introduced by Lee, Jones and Ben Amram [8] to study the termination of first-order functional programs. It proved to be very effective, and a few extensions to typed λ-calculi and higher-order rewriting were proposed.

In his PhD thesis [13], Wahlstedt proposes one for proving the termination, in some presentation of Martin-Löf's type theory, of an higher-order rewrite system $\mathcal{R}$ together with the β-reduction of λ-calculus. He proceeds in two steps. First, he defines an order, the instantiated call relation, and proves that $\longrightarrow = \longrightarrow_{\mathcal{R}} \cup \longrightarrow_{\beta}$ terminates on well-typed terms whenever this order is well-founded. Then, he uses SCT to eventually show the latter.

However, Wahlstedt's work has some limitations. First, it only considers weak normalization, that is, the mere existence of a normal form. Second, it makes a strong distinction between "constructor" symbols, on which pattern matching is possible, and "defined" symbols, which are allowed to be defined by rewrite rules. Hence, it cannot handle all the systems that one can define in the $\lambda\Pi$-calculus modulo rewriting, the type system implemented in Dedukti [2].

Other works on higher-order rewriting do not have those restrictions, like [3] in which strong normalization (absence of infinite reductions) is proved in the calculus of constructions by requiring each right-hand side of rule to belong to the Computability Closure (CC) of its corresponding left-hand side.

In this paper, we present a combination and extension of both approaches.

2 The $\lambda\Pi$-calculus modulo rewriting

We consider the $\lambda\Pi$-calculus modulo rewriting [2]. This is an extension of Automath, Martin-Löf's type theory or LF, where functions and types can be defined by rewrite rules, and where types are identified modulo those rules and the β-reduction of λ-calculus.

Assuming a signature made of a set $\mathcal{C}_T$ of type-level constants, a set $\mathcal{F}_T$ of type-level definable function symbols, and a set $\mathcal{F}_o$ of object-level function symbols, terms are inductively defined into three categories as follows:

kind-level terms	K	$::= \text{Type} \mid (x : U) \to K$	
type-level terms	T, U	$::= \lambda x : U.\, T \mid (x : U) \to T \mid U\, t \mid D \mid F$	where $D \in \mathcal{C}_T$ and $F \in \mathcal{F}_T$
object-level terms	t, u	$::= x \mid \lambda x : U.\, t \mid t\, u \mid f$	where $f \in \mathcal{F}_o$

© Frédéric Blanqui and Guillaume Genestier;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

By $\bar{t}$, we denote a sequence of terms $t_1 \dots t_n$ of length $|\bar{t}| = n$.

Next, we assume given a function τ associating a kind to every symbol of $\mathcal{C}_T$ and $\mathcal{F}_T$, and a type to every symbol of $\mathcal{F}_o$. If $\tau(f) = (x_1 : T_1) \to \dots \to (x_n : T_n) \to U$ with U not an arrow, then f is said of arity $\mathrm{ar}(f) = n$.

An object-level function symbol f of type $(x_1 : T_1) \to \dots \to (x_n : T_n) \to D\, u_1 \dots u_{\mathrm{ar}(D)}$ with $D \in \mathcal{C}_T$ and every T_i of the form $E\, v_1 \dots v_{\mathrm{ar}(E)}$ with $E \in \mathcal{C}_T$ is called a *constructor*. Let $\mathcal{C}_o$ be the set of constructors.

Terms built from variables and constructor application only are called *patterns*: $p ::= x \mid c\, p_1 \dots p_{\mathrm{ar}(c)}$ where $c \in \mathcal{C}_o$.

Next, we assume given a set $\mathcal{R}$ of rewrite rules of the form $f\, p_1 \dots p_{\mathrm{ar}(f)} \longrightarrow r$, where f is in $\mathcal{F}_o$ or $\mathcal{F}_T$, the p_i's are patterns and r is β-normal. Then, let $\longrightarrow = \longrightarrow_\beta \cup \longrightarrow_\mathcal{R}$ where $\longrightarrow_\mathcal{R}$ is the smallest rewrite relation containing $\mathcal{R}$.

Note that rewriting at type level is allowed. For instance, we can define a function taking a natural number n and returning Nat → Nat → ⋯ → Nat with as many arrows as n. In Dedukti syntax, this gives:

```
def F : Nat -> Type.
[]  F 0     --> Nat.
[n] F (S n) --> Nat -> (F n).
```

Well-typed terms are defined as in LF, except that types are identified not only modulo β-equivalence but modulo $\mathcal{R}$-equivalence also, by adding the following type conversion rule:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash t : B} \quad \text{if } A \longleftrightarrow^* B \text{ and } s \in \{\mathrm{Type}, \mathrm{Kind}\}$$

Convertibility of A and B, $A \longleftrightarrow^* B$, is undecidable in general. However, it is decidable if $\longrightarrow$ is confluent and terminating. So, a type-checker for the $\lambda\Pi$-calculus modulo $\longleftrightarrow^*$, like Dedukti, needs a criterion to decide termination of $\longrightarrow$. This is the reason of this work.

To this end, we assume that $\longrightarrow$ is confluent and preserves typing.

There exist tools to check confluence, even for higher-order rewrite systems, like CSIˆho or ACPH. The difficulty in presence of type-level rewrite rules, is that we cannot assume termination to show confluence since we need confluence to prove termination. Still, there is a simple criterion in this case: orthogonality [12].

Checking that $\longrightarrow$ preserves typing is undecidable too (for $\longrightarrow_\beta$ alone already), and often relies on confluence except when type-level rewrite rules are restricted in some way [3]. Saillard designed and implemented an heuristic in Dedukti [10].

Finally, note that constructors can themselves be defined by rewrite rules. This allows us to define, for instance, the type of integers with two constructors for the predecessor and successor, together with the rules stating that they are inverse of each other.

3 The Size-Change Termination principle

Introduced for first-order functional programming languages by Lee, Jones and Ben Amram [8], the SCT is a simple but powerful criterion to check termination. We recall hereafter the matrix-based presentation of SCT by Lepigre and Raffalli [9].

▸ **Definition 1** (Size-Change Termination principle)**.** The (strict) *constructor subterm relation* $\triangleleft$ is the smallest transitive relation such that $t_i \triangleleft c\, t_1 \dots t_n$ when $c \in \mathcal{C}_o$.

We define the *formal call relation* by $f\, \bar{p} >_{call} g\, \bar{t}$ if there is a rewrite rule $f\, \bar{p} \longrightarrow r \in \mathcal{R}$ such that $g \in \mathcal{F}_T \cup \mathcal{F}_o$ and $g\, \bar{t}$ is a subterm of r with $|\bar{t}| = \mathrm{ar}(g)$.

From this relation, we construct a *call graph* whose nodes are labeled with the defined symbols. For every call $f\,\bar{p} >_{call} g\,\bar{t}$, an edge labeled with the *call matrix* $(a_{i,j})_{i\leqslant \mathrm{ar}(f), j\leqslant \mathrm{ar}(g)}$ links the nodes f and g, where $a_{i,j} = -1$ if $t_j \lhd p_i$, $a_{i,j} = 0$ if $t_j = p_i$, and $a_{i,j} = \infty$ otherwise.

A set of rewrite rules $\mathcal{R}$ satisfies the *size-change termination principle* if the transitive closure of the call graph (using the max-plus semi-ring to multiply the matrices) is such that all arrows linking a node with itself are labeled with a matrix having at least one -1 on the diagonal.

The formal call relation is also called the dependency pair relation [1].

4 Wahlstedt's extension of SCT to Martin-Löf's Type Theory

The proof of weak normalization in Wahlstedt's thesis uses an extension to rewriting of Girard's notion of reducibility candidate [7], called computability predicate here. This technique requires to define an interpretation of every type T as a set of normalizing terms $[\![T]\!]$ called the set of *computable* terms of type T. Once this interpretation is defined, one shows that every well-typed term $t : T$ is *computable*, that is, belongs to the interpretation of its type: $t \in [\![T]\!]$, ending the normalization proof. To do so, Wahlstedt proceeds in two steps. First, he shows that every well-typed term is computable whenever all symbols are computable. Then, he introduces the following relation which, roughly speaking, corresponds to the notion of minimal chain in the DP framework [1]:

▸ **Definition 2** (Instantiated call relation)**.** Let $f\,\bar{t} \precsim g\,\bar{v}$ if there exist $\bar{p}$, $\bar{u}$ and a substitution γ such that $\bar{t}$ is normalizing, $\bar{t} \longrightarrow^* \bar{p}\gamma$, $f\,\bar{p} >_{call} g\,\bar{u}$ and $\bar{u}\gamma = \bar{v}$.

and proves that all symbols are computable if $\precsim$ is well-founded:

▸ **Lemma 3** ([13, Lemma 3.6.6, p. 82])**.** *If $\precsim$ is well-founded, then all symbols are computable.*

Finally, to prove that $\precsim$ is well-founded, he uses SCT:

▸ **Lemma 4** ([13, Theorem 4.2.1, p. 91])**.** $\precsim$ *is well-founded whenever the set of rewrite rules satisfies* SCT.

Indeed, if $\precsim$ were not well-founded, there would be an infinite sequence $f_1\,\bar{t}_1 \precsim f_2\,\bar{t}_2 \precsim \ldots$, leading to an infinite path in the call graph which would visit infinitely often at least one node, say f. But the matrices labelling the looping edges in the transitive closure all contain at least one -1 on the diagonal, meaning that there is an argument of f which strictly decreases in the constructor subterm order at each cycle. This would contradict the well-foundedness of the constructor subterm order.

However, Wahlstedt only considers weak normalization of orthogonal systems, in which constructors are not definable. There exist techniques which do not suffer those restrictions, like the Computability Closure.

5 Computability Closure

The Computability Closure (CC) is also based on an extension of Girard's computability predicates [4], but for strong normalization. The gist of CC is, for every left-hand side of a rule $f\,\bar{l}$, to inductively define a set $\mathrm{CC}_\sqsupset(f\,\bar{l})$ of terms that are computable whenever the l_i's so are. Function applications are handled through the following rule:

$$\frac{f\,\bar{l} \sqsupset g\,\bar{u} \qquad \bar{u} \in \mathrm{CC}_\sqsupset(f\,\bar{l})}{g\,\bar{u} \in \mathrm{CC}_\sqsupset(f\,\bar{l})}$$

where $\sqsupset = (>_{\mathcal{F}}, (\triangleright \cup \longrightarrow)_{\text{stat}})_{\text{lex}}$ is a well-founded order on terms $f\,\bar{t}$ such that $\bar{t}$ are computable, with $>_{\mathcal{F}}$ a precedence on function symbols and stat either the multiset or the lexicographic order extension, depending on f.

Then, to get strong normalization, it suffices to check that, for every rule $f\,\bar{l} \longrightarrow r$, we have $r \in \text{CC}_{\sqsupset}(f\,\bar{l})$. This is justified by Lemma 6.38 [3, p.85] stating that all symbols are computable whenever the rules satisfy CC, which looks like Lemma 3. It is proved by induction on $\sqsupset$. By definition, $f\,\bar{t}$ is computable if, for every u such that $f\,\bar{t} \longrightarrow u$, u is computable. There are two cases. If $u = f\,\bar{t}'$ and $\bar{t} \longrightarrow \bar{t}'$, then we conclude by the induction hypothesis. Otherwise, $u = r\,\gamma$ where r is the right-hand side of a rule whose left-hand side is of the form $f\,\bar{l}$. This case is handled by induction on the proof that $r \in \text{CC}_{\sqsupset}(f\,\bar{l})$.

So, except for the order, the structures of the proofs are very similar in both works. This is an induction on the order, a case distinction and, in the case of a recursive call, another induction on a refinement of the typing relation, restricted to β-normal terms in Wahlstedt's work and to the Computability Closure membership in the other one.

6 Applying ideas of Computability Closure in Wahlstedt's criterion

We have seen that each method has its own weaknesses: Wahlstedt's SCT deals with weak normalization only and does not allow pattern-matching on defined symbols, while CC enforces mutually defined functions to perform a strict decrease in each call.

We can subsume both approaches by combining them and replacing in the definition of CC the order $\sqsupset$ by the formal call relation:

$$\frac{f\,\bar{l} >_{call} g\,\bar{u} \qquad \bar{u} \in \text{CC}_{>_{call}}(f\,\bar{l})}{g\,\bar{u} \in \text{CC}_{>_{call}}(f\,\bar{l})}$$

We must note here that, even if $>_{call}$ is defined from the constructor subterm order, this new definition of CC does not enforce an argument to be strictly smaller at each recursive call, but only smaller or equal, with the additional constraint that any looping sequence of recursive calls contains a step with a strict decrease, which is enforced by SCT.

▸ **Proposition 5.** *Let $\mathcal{R}$ be a rewrite system such that $\longrightarrow = \longrightarrow_{\beta} \cup \longrightarrow_{\mathcal{R}}$ is confluent and preserves typing. If $\mathcal{R}$ satisfies $\text{CC}_{>_{call}}$ and* SCT*, then $\longrightarrow$ terminates on every term typable in the $\lambda\Pi$-calculus modulo $\longleftrightarrow^*$.*

Note that $\text{CC}_{>_{call}}$ essentially reduces to checking that the right-hand sides of rules are well-typed which is a condition that is generally satisfied.

The main difficulty is to define an interpretation for types and type symbols that can be defined by rewrite rules. It requires to use induction-recursion [6]. Note that the well-foundedness of the call relation $\gtrdot$ is used not only to prove reducibility of defined symbols, but also to ensure that the interpretation of types is well-defined.

If we consider the example of integers mentioned earlier and define the function erasing every constructor using an auxiliary function, we get a system rejected both by Wahlstedt's criterion since `S` and `P` are defined, and by the $CC_{\sqsupset}$ criterion since there is no strict decrease in the first rule. On the other hand, it is accepted by our combined criterion.

```
Int : Type.                          0 : Int.
def S : Int -> Int.                  def P : Int -> Int.

[x] S (P x) --> x.                   [x] P (S x) --> x.
[x] returnZero x --> aux x.          []  aux 0     --> 0.
[x] aux (S x) --> returnZero x.      [x] aux (P x) --> returnZero x.
```

7 Conclusion

We have shown that Wahlstedt's thesis [13] and the first author's work [3] have strong similarities. Based on this observation, we developed a combination of both techniques that strictly subsumes both approaches.

This criterion has been implemented in the type-checker Dedukti [2] and gives promising results, even if automatically proving termination of expressive logic encodings remains a challenge. The code is available at https://github.com/Deducteam/Dedukti/tree/sizechange.

Many opportunities exist to enrich our new criterion. For instance, the use of an order leaner than the strict constructor subterm for SCT, like the one defined by Coquand [5] for handling data types with constructors taking functions as arguments. This question is studied in the first-order case by Thiemann and Giesl [11].

Finally, it is important to note the modularity of Wahlstedt's approach. Termination is obtained by proving 1) that all terms terminate whenever the instantiated call relation is well-founded, and 2) that the instantiated call relation is indeed well-founded. Wahlstedt and we use SCT to prove 2) but it should be noted that other techniques could be used as well. This opens the possibility of applying to type systems like the ones implemented in Dedukti, Coq or Agda, techniques and tools developed for proving the termination of DP problems.

Acknowledgments. The authors thank Olivier Hermant for his comments, as well as the anonymous referees.

References

1 T. Arts, J. Giesl. Termination of term rewriting using dependency pairs. *TCS* 236, 2000.
2 A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Dedukti: a Logical Framework based on the λΠ-Calculus Modulo Theory, 2016. Draft.
3 F. Blanqui. Definitions by rewriting in the calculus of constructions. *MSCS* 15(1), 2005.
4 F. Blanqui. Termination of rewrite relations on λ-terms based on Girard's notion of reducibility. *TCS*, 611:50–86, 2016.
5 T. Coquand. Pattern matching with dependent types. In *Proc. of TYPES'92*.
6 P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. of Symbolic Logic*, 65(2):525–549, 2000.
7 J.-Y. Girard, Y. Lafont, P. Taylor. *Proofs and types*. Cambridge University Press, 1988.
8 C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Proc. of POPL'01*.
9 R. Lepigre and C. Raffalli. Practical Subtyping for System F with Sized (Co-)Induction. https://arxiv.org/abs/1604.01990, 2017.
10 R. Saillard. *Type Checking in the Lambda-Pi-Calculus Modulo: Theory and Practice*. PhD thesis, Mines ParisTech, France, 2015.
11 R. Thiemann and J. Giesl. The size-change principle and dependency pairs for termination of term rewriting. *AAECC*, 16(4):229–270, 2005.
12 V. van Oostrom and F. van Raamsdonk. Weak orthogonality implies confluence: the higher-order case. In *Proc. of LFCS'94*, LNCS 813.
13 D. Wahlstedt. *Dependent type theory with first-order parameterized data types and well-founded recursion*. PhD thesis, Chalmers University of Technology, Sweden, 2007.

Complexity Analysis for Bitvector Programs*

Jera Hensel, Florian Frohn, and Jürgen Giesl

LuFG Informatik 2, RWTH Aachen University, Germany
{hensel,florian.frohn,giesl}@informatik.rwth-aachen.de

Abstract

In earlier work, we developed approaches for automated termination analysis of several different programming languages, based on back-end techniques for termination proofs of term rewrite systems and integer transition systems. In the last years, we started adapting these approaches in order to analyze the complexity of programs as well. However, up to now a severe drawback was that we assumed the program variables to range over mathematical integers instead of bitvectors. This eases mathematical reasoning but is unsound in general. While we showed in [8] how to handle fixed-width bitvector integers in termination analysis, we now present the first technique to analyze the runtime complexity of programs with bitvector arithmetic. We implemented our contributions in the tool AProVE and evaluate its power by extensive experiments.

Keywords and phrases Complexity Analysis, Bitvectors, C, Integer Transition Systems

1 Introduction

Our verifier AProVE [7] is one of the leading tools for termination analysis of languages like Java, C, Haskell, Prolog, and term rewrite systems, as witnessed by its success at the annual *Termination Competition* and the termination category of the *SV-COMP* competition.[1] However, often one is not only interested in termination, but in the runtime of a program. Thus, *automated complexity analysis* has become increasingly important and there exist several tools which analyze the complexity of programs in different languages and formalisms.

In [6], we adapted our approach for termination of Java to infer complexity bounds. Based on a symbolic execution of the program, we developed a transformation of (possibly heap-manipulating) Java programs to *integer transition systems* (*ITSs*). These ITSs are then analyzed by standard complexity tools for integer programs like CoFloCo [5] and KoAT [3].

However, similar to many other termination techniques, our approach for termination and complexity analysis of Java is restricted to mathematical integers. To see why this is unsound when analyzing languages like C or Java, consider the C functions `f` and `g` in Fig. 1, which increment a variable `j` as long as the loop condition holds. For `f`, one leaves the loop as soon as `j` exceeds the value of `x`. Thus, `f` does not terminate if `x` has the maximum value of its type.[2] But we can falsely prove termination if we treat `x` and `j` as mathematical integers. For `g`, the loop terminates as soon as the value of `j` becomes zero. So when considering mathematical integers, we would falsely conclude non-termination for positive initial values

* Supported by the DFG grant GI 274/6-1.

[1] See `http://www.termination-portal.org/wiki/Termination_Competition` and `http://sv-comp.sosy-lab.org`.

[2] In C, adding 1 to the maximal unsigned integer results in 0. In contrast, for signed integers, adding 1 to the maximal signed integer results in undefined behavior. However, most C implementations return the minimal signed integer as the result.

© Jera Hensel, Florian Frohn, and Jürgen Giesl;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

```
void f(unsigned int x)  {
    unsigned int j = 0;
    while (j <= x)
        j++;
}
```

```
void g(unsigned int j) {
    while (j > 0)
        j++;
}
```

Figure 1 C functions on bitvectors

of `j`, although `g` always terminates due to the wrap-around for unsigned overflows.

In [8], we showed how termination techniques can be extended from mathematical integers to bitvector integers and adapted our approach for termination analysis of C programs from [13] accordingly. In this way, we obtained the first technique for termination of C programs that covers both byte-accurate explicit pointer arithmetic and bit-precise modeling of integers.

In the current paper, we show that such an extension to bitvectors can also be used to analyze the runtime complexity. To this end, we extend the termination technique for bitvector C programs from [8] to analyze the complexity of programs. In a similar way, our complexity technique for Java [6] could also be adapted to treat integers as bitvectors.

To avoid dealing with the intricacies of C, we analyze programs in the intermediate representation of the LLVM compilation framework [11]. As an example, consider the LLVM code for the function `g` in Fig. 2. To ease readability, we wrote variables without "`%`" (i.e., we wrote "`j`" instead of "`%j`" as in proper LLVM) and added line numbers. Here, `j` is of type `i32`, where `i`n is the type of n-bit integers. So $0 \leq \texttt{j} \leq 2^{32} - 1 = \mathsf{umax}_{32}$, where $\mathsf{umax}_n = 2^n - 1$ is the maximum unsigned value of the type `i`n.[3] In the basic block `entry`, `j` is stored at the address `ad`. In the block `cmp`, one performs an integer comparison (`icmp`) to check whether the value at the address `ad` is unsigned-greater than 0 (`ugt`). In that case, this value is incremented by 1 in the block `body` and one branches (`br`) back to the block `cmp`.

Our approach for termination analysis works in two steps: First, it constructs a *symbolic execution graph* that represents an over-approximation of all possible program runs. This graph can also be used to prove that the program does not result in undefined behavior (so in particular, it is memory safe). In [8] we showed how to adapt the rules for symbolic execution of those LLVM instructions that are affected by the change from mathematical integers to bitvectors. In a second step, this graph is transformed into an ITS. If the resulting ITS is terminating, then the original C resp. LLVM program terminates as well. Note that we express relations between bitvectors by corresponding relations between mathematical integers $\mathbb{Z}$. In this way, we can use standard SMT solving over $\mathbb{Z}$ for all steps needed to construct the symbolic execution graph. Moreover, this allows us to obtain ITSs over mathematical integers from these graphs, and to use standard ITS tools to analyze their termination. In Sect. 2 we show that our transformation into ITSs can also be adapted in order to derive upper bounds on the program's runtime, i.e., our approach can be used for complexity analysis of bitvector programs as well. In Sect. 3, we evaluate our corresponding implementation in AProVE.

2 Finding Upper Runtime Complexity Bounds

To infer runtime bounds instead of proving termination, we keep the construction of the symbolic execution graph unchanged and only adapt our technique to transform the graph into an ITS. The resulting approach for complexity analysis mainly succeeds on arithmetic programs. To analyze programs whose runtime depends on the memory, one would have to extend the abstraction used in our symbolic execution, since then the abstract program

[3] LLVM does not distinguish between signed and unsigned integers. Instead, there are signed and unsigned versions of many arithmetical operations. We use a heuristic to guess if a variable is used signed or unsigned in the LLVM program and model all LLVM instructions correctly independently of our guess.

```
define i32 @g(i32 j) {
entry: 0: ad = alloca i32
       1: store i32 j, i32* ad
       2: br label cmp
cmp:   0: j1 = load i32* ad
       1: j1pos = icmp ugt i32 j1, 0
       2: br i1 j1pos, label body, label done
body:  0: j2 = load i32* ad
       1: inc = add i32 j2, 1
       2: store i32 inc, i32* ad
       3: br label cmp
done:  0: ret void }
```

Figure 2 LLVM code for the function g

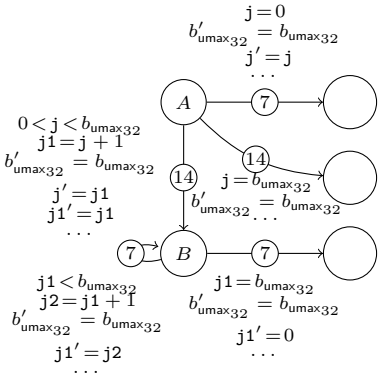


Figure 3 Weighted ITS for g

states would also have to contain information on the sizes of the allocated memory areas. Note that for a terminating arithmetic program with m instructions and k variables of types $\mathtt{i}n_1, \ldots, \mathtt{i}n_k$, the runtime is bounded by $m \cdot \prod_{j=1}^{k} 2^{n_j}$, which is the number of possible program states. The reason is that at each program position, every variable $\mathtt{x}_j$ may be assigned any value of its type (whose range is 2^{n_j}). Whenever a program state is visited twice, the program must be non-terminating. So since the state space is finite, every terminating arithmetic bitvector program has constant complexity. Thus, for terminating arithmetic programs on bitvectors, asymptotic complexity is meaningless as all programs have a runtime in $\mathcal{O}(1)$.

Therefore, our goal is to infer concrete (non-asymptotic) bounds which are smaller than the maximum bound $m \cdot \prod_{j=1}^{k} 2^{n_j}$. In particular, we aim to find bounds that depend on the program's input parameters, because such bounds are usually more interesting than a huge constant that depends on the sizes of the types $\mathtt{i}n$. We developed the following adaptions of our approach for termination analysis in order to find runtime bounds for bitvector programs. While (1)-(3) are used similarly in other approaches for termination or complexity analysis, we developed (4) and (5) specifically for our setting.

(1) For termination, one only has to consider the cycles of the symbolic execution graph. But for the runtime of a program, we have to count every execution step. Thus, the ITS must be extracted from the whole graph and not only from its cycles. Moreover, this is required to infer correct runtime bounds for subsequent cycles. The reason is that the first cycle might increase values which are used afterwards when entering the next cycle.

(2) The initial abstract state of the symbolic execution graph is also considered to be the initial location of the ITS. So only evaluation sequences of the ITS that start in this location have to be considered. Then the goal is to find a bound on the length of the ITS evaluations that depends only on the values of the variables in this location.

(3) For an efficient analysis, we simplify the transitions of the ITS by filtering away variables that do not influence the termination and by iteratively compressing several transitions into one, cf. [7]. This is unproblematic for termination analysis, but the compression of transitions would distort a concrete complexity result if several evaluation steps are counted as one. Therefore, we now assign a weight to each transition which over-approximates the number of evaluation steps that are represented by this transition. As shown in [6], such weights can also be used to modularize the analysis in order to increase scalability.

(4) Since all handling of bitvector arithmetic is done during the symbolic execution, we generate ITSs over mathematical integers. Hence, their complexity can be analyzed by existing complexity tools for such ITSs. Since such tools have not been used for bitvector programs yet, some of them are targeted towards the inference of small asymptotic bounds (i.e., for a program with constant runtime, they would rather infer a huge constant bound than a linear bound that depends on the program's input parameters).

To facilitate the deduction of a bound depending on the program's parameters and to obtain more informative bounds, we therefore perform the following modification of the ITS. During the graph construction, we now keep track of all constants like $\mathsf{umax}_{32} = 2^{32} - 1$ that originate from the size bounds of a variable's type. When the ITS is extracted from the graph, these constants are transformed to variables like $b_{\mathsf{umax}_{32}}$.

Thus, we obtain the weighted ITS in Fig. 3 from `g`'s symbolic execution graph. Here, 14 LLVM instructions are executed from `g`'s initial state A to the state B where the block `body` is reached for the second time. Thus, the transition from location A to B has weight 14 and it can be taken if the variables satisfy the conditions $0 < \mathtt{j} < b_{\mathsf{umax}_{32}}$, $\mathtt{j1} = \mathtt{j} + 1$, etc., where a primed variable denotes the value of the variable *after* the transition. The loop (i.e., the blocks `cmp` and `body`) contains 7 instructions. When evaluating State B symbolically, we consider the possible overflows and exit the loop if $\mathtt{j1} = \mathsf{umax}_{32}$ holds. In this case, 7 further instructions are executed until the function `g` ends with a `return`.

(5) To ensure that the ITS complexity tool prefers bounds that contain the program's parameters over bounds containing size bound variables like $b_{\mathsf{umax}_{32}}$, we first pass a modified ITS to the underlying complexity tool where the initial transitions do not impose any conditions on the size bound variables. So while all other transitions have requirements like $b'_{\mathsf{umax}_{32}} = b_{\mathsf{umax}_{32}}$ in their condition, the conditions of the initial transitions in this modified ITS do not contain variables like $b'_{\mathsf{umax}_{32}}$. Hence, now the size bound variables can change arbitrarily in the initial transitions and the runtime would be unbounded if it depends on one of these variables. Therefore, the complexity tool will try to find other runtime bounds that only depend on the program's parameters.

If the complexity analysis of this modified ITS fails, then instead we use the ITS as before, where the size bound variables like $b_{\mathsf{umax}_{32}}$ are considered to be input parameters. In other words, now the initial transitions also contain $b' = b$ for all size bound variables b and this ITS is now given to the complexity tool in the back-end.

For the function `g`, no upper bound is found if the size bound variables are treated as being unbounded (i.e., if one deletes $b'_{\mathsf{umax}_{32}} = b_{\mathsf{umax}_{32}}$ from the conditions of the initial transitions). On the other hand, if one calls the complexity tool CoFloCo with the ITS from Fig. 3 where $b_{\mathsf{umax}_{32}}$ is considered to be an input parameter, then we obtain the bound $\max(21, 7 \cdot b_{\mathsf{umax}_{32}} - 7 \cdot \mathtt{j} + 14)$. In fact, if $0 < \mathtt{j} < b_{\mathsf{umax}_{32}}$ holds at the beginning of the program, then the loop is executed $b_{\mathsf{umax}_{32}} - \mathtt{j} - 1$ times. Since the loop of the ITS consists of 7 instructions and the path of the loop has $14 + 7 = 21$ remaining instructions, in this case we obtain $7 \cdot (b_{\mathsf{umax}_{32}} - \mathtt{j} - 1) + 21 = 7 \cdot b_{\mathsf{umax}_{32}} - 7 \cdot \mathtt{j} + 14$ instructions for the path of the loop. CoFloCo combines this with the maximum weight of all paths that do not traverse loops, which results in the bound $\max(21, 7 \cdot b_{\mathsf{umax}_{32}} - 7 \cdot \mathtt{j} + 14)$.

Note that the replacement of constants by variables like $b_{\mathsf{umax}_{32}}$ yields a more informative bound than the corresponding term $30064771079 - 7 \cdot \mathtt{j}$: the bound $7 \cdot b_{\mathsf{umax}_{32}} - 7 \cdot \mathtt{j} + 14$ clearly shows that the runtime depends on the range of the type `i32`. For this program, there is indeed no reasonable upper bound that depends on `j` but not on umax_{32}.

3 Experiments and Conclusion

In [8], we adapted our approach for proving memory safety and termination of C (resp. LLVM) programs to bitvector semantics. While before, program variables were treated as mathematical integers and overflows were ignored, bitvector operations such as type conversions, overflow-sensitive operations, and bitwise operations are now modeled correctly. In the current paper, we showed how our adaption of symbolic execution can also be used for complexity analysis. We transform programs into ITSs over mathematical integers and thus, we can use standard complexity tools to infer upper runtime bounds for the resulting ITSs. While there exists a wealth of recent techniques and tools for complexity analysis of programs on mathematical integers (e.g., [1, 3, 4, 5, 10, 12]), to our knowledge our approach is the first

which allows us to use these tools to analyze the runtime of bitvector programs automatically.

We implemented our approach in the tool AProVE [7] and used KoAT [3] and CoFloCo [5] in the back-end. We always ran KoAT and CoFloCo in parallel and took the minimum of the bounds obtained by the two tools. To evaluate our implementation, we performed experiments on 118 C programs (which we mainly obtained from the collections used for the evaluations of other C termination tools). Out of the 95 programs where AProVE could show termination, it infers an upper bound for 60 programs, using a time-out of 300 seconds per example. For 7 of these programs, AProVE finds a small constant bound. Here, the runtime indeed does not depend on the input variables or on the sizes of the types. For 38 programs, an upper bound is found that depends linearly on the input variable(s) and for 3 more programs, a quadratic upper bound is obtained. Thus, the runtime of these 41 programs is independent of the sizes of the integer types. For 4 programs, AProVE generates an upper bound that only depends on size bound variables. For the remaining 8 programs, the inferred bound depends on both size bound variables and input variables of the function.

For details on our experiments (including the exact runtime bounds inferred by AProVE) and to access our implementation via a web interface, we refer to [2]. A full version of the current paper combined with [8] appeared in [9]. The full version also contains a comparison with the variant of AProVE where integers are treated as mathematical integers.

References

1 E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

2 `http://aprove.informatik.rwth-aachen.de/eval/BitvectorTerminationComplexity`.

3 M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Analyzing runtime and size complexity of integer programs. *ACM Transactions on Programming Languages and Systems*, 38(4):13:1–13:50, 2016.

4 Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional certified resource bounds. In *Proc. PLDI '15*, pages 467–478, 2015.

5 A. Flores-Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In *Proc. APLAS '14*, LNCS 8858, pages 275–295, 2014.

6 F. Frohn and J. Giesl. Complexity analysis for Java with AProVE. In *Proc. iFM '17*, LNCS 10510, pages 85–101, 2017.

7 J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning*, 58:3–31, 2017.

8 J. Hensel, J. Giesl, F. Frohn, and T. Ströder. Proving termination of programs with bitvector arithmetic by symbolic execution. In *Proc. SEFM '16*, LNCS 9763, pages 234–252, 2016.

9 J. Hensel, J. Giesl, F. Frohn, and T. Ströder. Termination and complexity analysis for programs with bitvector arithmetic by symbolic execution. *Journal of Logical and Algebraic Methods in Programming*, 97:105–130, 2018.

10 J. Hoffmann, A. Das, and S.-C. Weng. Towards automatic resource bound analysis for OCaml. In *Proc. POPL '17*, pages 359–373, 2017.

11 C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. CGO '04*, pages 75–88, 2004.

12 M. Sinn, F. Zuleger, and H. Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *Journal of Automated Reasoning*, 59(1):3–45, 2017.

13 T. Ströder, J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, P. Schneider-Kamp, and C. Aschermann. Automatically proving termination and memory safety for programs with pointer arithmetic. *Journal of Automated Reasoning*, 58:33–65, 2017.

Semantic Kachinuki Order

Alfons Geser[1], Dieter Hofbauer[2], and Johannes Waldmann[1]

1 **HTWK Leipzig, Germany**
2 **ASW Berufsakademie Saarland, Germany**

Abstract

We present an extension of the Kachinuki order on strings. The Kachinuki order transforms the problem of comparing strings to the problem of comparing their syllables length-lexicographically, where the syllables are defined via a precedence on the alphabet. Our extension allows the number of syllables to increase under rewriting, provided we bound it by a weakly compatible interpretation.

1 Introduction

The *Kachinuki* order [6] is an iterated syllable order on strings: given a precedence on the alphabet, words are split at occurrences of the highest letter, resulting in sequences of syllables over a smaller alphabet. These sequences are compared by the (length-)lexicographic extension of the Kachinuki order on the smaller alphabet (without the highest letter). For instance, to show termination of $ab \to bba$, we take precedence $a > b$, and split at occurrences of a, to obtain $[\epsilon, b] \to [bb, \epsilon]$. We note that the number of syllables is constant under rewriting, and that the word of syllables is lexicographically decreasing (looking from the right end) w.r.t. the number of b. The Kachinuki order is a simplification order.

The word *Kachinuki* means "the winner walks through" and describes the order of the bouts in team matches in Japanese judo. For ordering strings, the "winner" is the symbol of highest precedence. In its original definition, the Kachinuki order is equivalent to the recursive path order on strings, see also [7]. It is folklore knowledge (e.g., assumed in [3]) that the definition can be extended by assigning, to each symbol x, a status bit that tells whether the sequence of syllables w.r.t. x should be compared lexicographically from the left, or from the right. For example, termination of $\{ab \to bba, bc \to cbb\}$ can be shown by precedence $c > a > b$ and status Left for c, Right for a. This extension is specific to string rewriting, as there is no obvious way to reverse a term.

In the present note, we show a way to include semantic information (from an interpretation) in the choice of the letter(s) for splitting. For instance, we want to prove termination of $\{ca \to b^2c^3, bcb \to a\}$ by splitting at occurrences of a and b. This is allowed because the weight function $w : a \mapsto 2, b \mapsto 1, c \mapsto 0$ is non-decreasing, and a and b have positive weight.

2 Syllables

▶ **Definition 1.** The syllable word $\mathsf{split}(\Gamma, w)$ of a string $w \in \Sigma^*$ with respect to a subset $\Gamma \subseteq \Sigma$ of the alphabet Σ is the unique sequence $[s_0, s_1, \ldots, s_k] \in (\Sigma \setminus \Gamma)^{*+}$ such that $w = s_0 a_1 s_1 a_2 \ldots a_k s_k$ with each $a_i \in \Gamma$.

▶ **Example 2.** For $\Sigma = \{a, b, c, d\}$ and $w = abcadc$, we have $\mathsf{split}(\{b, c\}, w) = [a, \epsilon, ad, \epsilon]$, $\mathsf{split}(\Sigma, w) = \epsilon^7$, and $\mathsf{split}(\emptyset, w) = [w]$. For each Γ, we have $\mathsf{split}(\Gamma, \epsilon) = [\epsilon]$.

We sometimes write $\mathsf{split}(a, w)$ for $\mathsf{split}(\{a\}, w)$, and $\mathsf{split}_\Gamma(w)$ for $\mathsf{split}(\Gamma, w)$.

▶ **Definition 3.** The binary operation $\odot$ on Σ^{*+} is defined by $[x_1, \ldots, x_p] \odot [y_1, \ldots, y_q] = [x_1, \ldots, x_p \cdot y_1, \ldots, y_q]$.

© Alfons Geser, Dieter Hofbauer, and Johannes Waldmann;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

▶ **Example 4.** $[ab, c] \odot [a, bc] = [ab, ca, bc]$.

▶ **Lemma 5.** $(\Sigma^{*+}, \odot, [\epsilon])$ *is a monoid, and* split_Γ *is a monoid morphism from* $(\Sigma^*, \cdot, \epsilon)$.

▶ **Lemma 6.** *For each rewrite step* $x = plq \to prq = y$, *we have* $\mathit{split}_\Gamma(x) = \mathit{split}_\Gamma(p) \odot \mathit{split}_\Gamma(l) \odot \mathit{split}_\Gamma(q)$ *and* $\mathit{split}_\Gamma(y) = \mathit{split}_\Gamma(p) \odot \mathit{split}_\Gamma(r) \odot \mathit{split}_\Gamma(q)$.

For notational convenience, we extend the split operation from words to rules and sets of rules.

▶ **Example 7.** For $R = \{cab \to bcba, bcba \to abc\}$, let $\mathsf{split}_{\{a,b\}}(R)$ denote $\{[c, \epsilon, \epsilon] \to [\epsilon, c, \epsilon, \epsilon], [\epsilon, c, \epsilon, \epsilon] \to [\epsilon, \epsilon, c]\}$.

3 Stable Lexicographic Extension

▶ **Definition 8.** The *stable lexicographic* extensions $>^{\mathsf{L}}_{\mathsf{Lex}}, >^{\mathsf{R}}_{\mathsf{Lex}}$ of a relation $>$ on Σ are

$$[x_1, \ldots, x_p] >^{\mathsf{L}}_{\mathsf{Lex}} [y_1, \ldots, y_q] \iff \exists i : x_1 \geq y_1 \wedge \cdots \wedge x_{i-1} \geq y_{i-1} \wedge x_i > y_i,$$
$$[x_1, \ldots, x_p] >^{\mathsf{R}}_{\mathsf{Lex}} [y_1, \ldots, y_q] \iff \exists i : x_p \geq y_q \wedge \cdots \wedge x_{p-i+1} \geq y_{p-i+1} \wedge x_{p-i} > y_{q-i}.$$

We do not use the standard definition "... or y is a strict prefix (suffix, resp.) of x" because we need the following

▶ **Lemma 9.** $>^{L}_{Lex}$ *and* $>^{R}_{Lex}$ *are stable w.r.t. concatenation from both sides.*

Consider $(\mathbb{N}, >)$ as the domain, and $x = [1, 2, 3], y = [1, 2]$. Concatenate with $[4]$ on the right, obtain $x' = [1, 2, 3, 4], y' = [1, 2, 4]$. We do not want $x >^{\mathsf{L}}_{\mathsf{Lex}} y$, since certainly $x' \not>^{\mathsf{L}}_{\mathsf{Lex}} y'$.

Using stability and Lemma 6, we get

▶ **Lemma 10.** *If* $>$ *is a reduction order, and* $u \to_{(l,r)} v$ *with* $\mathit{split}_\Gamma(l) >^{L}_{Lex} \mathit{split}_\Gamma(r)$, *then* $\mathit{split}_\Gamma(u) >^{L}_{Lex} \mathit{split}_\Gamma(v)$. *Similarly, for* $>^{R}_{Lex}$.

▶ **Lemma 11.** *If* $(D, >)$ *is well-founded, then for each* n, $(D^{\leq n}, >^{L}_{Lex})$ *and* $(D^{\leq n}, >^{R}_{Lex})$ *are well-founded.*

Proof. Take any decreasing sequence in $(D^{\leq n}, >^{\mathsf{L}}_{\mathsf{Lex}})$. Obtain a decreasing sequence in $(D^n, >^{\mathsf{L}}_{\mathsf{Lex}})$ by padding (to the right) shorter strings with arbitrary elements of D. Well-foundedness of $(D^n, >^{\mathsf{L}}_{\mathsf{Lex}})$ is a standard result. ◀

4 Bounding Occurrences of Letters

The standard Kachinuki order requires that the number of occurrences of the highest symbol does not increase under rewriting. We will generalize this to allow some increase, provided that for each starting string x, the number of such occurrences in strings reachable from x is bounded. This concept can be seen as a generalisation of

▶ **Definition 12.** [1] A relation $\to$ is *quasi-terminating* if for each x, the set $\{y \mid x \to^* y\}$ is finite.

In that case, also the lengths of reachable words are bounded. Now we will just count occurrences of letters from a subset of the alphabet.

▶ **Definition 13.** For $\Gamma \subseteq \Sigma$, a relation $\to$ on Σ^* is called Γ-*quasi-terminating*, if for each x, the set $\{y_\Gamma \mid x \to^* y\}$ is finite, where y_Γ denotes the image of y under the morphism that deletes all letters not in Γ.

Equivalently (for finite Γ) there is a bound on the number of occurrences of letters from Γ in words reachable from x.

The name is justified by the following

▶ **Lemma 14.** *If $\to$ on Σ^* is Σ-quasi-terminating, then $\to$ is quasi-terminating.*

▶ **Theorem 15.** *For a string rewriting system R on Σ and $\Gamma \subseteq \Sigma$, if $\to_R$ is Γ-quasi-terminating, and there is some well-founded reduction order $>$ on $(\Sigma\backslash\Gamma)^*$ such that $\mathsf{split}_\Gamma(R)$ is contained in $>^{\mathsf{L}}_{\mathsf{Lex}}$ or in $>^{\mathsf{R}}_{\mathsf{Lex}}$, then R is terminating.*

5 Interpretations that prove Γ-Quasi-Termination

▶ **Definition 16.** A domain with relation $(D, >)$ is called *uniformly well-founded* if for each $x \in D$, there is $b \in \mathbb{N}$ such that each descending $>$-chain has length at most b.

▶ **Example 17.** We can take D as the set of square matrices $\mathbb{N}^{d\times d}$, and $A > B$ if $A_{1,d} > B_{1,d}$, disregarding relations between matrix entries elsewhere.

▶ **Example 18.** (Non-Example.) Take $D = (\mathbb{N}^2, >^{\mathsf{L}}_{\mathsf{Lex}})$. Each chain starting in $(1, 0)$ is finite, but the length of all these chains is not bounded, as we have $(1, 0) > (0, n) > \cdots > (0, 0)$.

▶ **Definition 19.** For $(D, >)$, an interpretation $i : \Sigma \to (D \to D)$, we write $i_a(x)$ as shorthand for $i(a)(x)$. We call i *weakly simple* if $\forall x \in D, a \in \Sigma : i_a(x) \geq x$. W.r.t. such an interpretation, letter a is called *strongly simple*, or just *strong*, if $\forall x \in D : i_a(x) > x$. We let $\mathsf{Strong}(i)$ denote the set of i-strong letters.

▶ **Example 20.** An interpretation into $E = \{A \mid A \in \mathbb{N}^{d\times d}, A_{1,1} \geq 1 \wedge A_{d,d} \geq 1\}$, cf. [4], is weakly simple w.r.t. the order from Example 17. Matrices in $E_+ = \{A \mid A \in \mathbb{N}^{d\times d}, A_{1,1} \geq 1 \wedge A_{d,d} \geq 1 \wedge A_{1,d} > 0\}$ are strong.

The above includes, as a special case, linear interpretations of slope 1, that is, weights. We represent weight $w \geq 0$ by matrix $\begin{pmatrix} 1 & w \\ 0 & 1 \end{pmatrix}$. This is always in E, and it is in E_+ if $w > 0$.

▶ **Example 21.** We can use interpretations into arctic matrices [5] as well. Denote by 0 the unit of arctic multiplication. We use domain $\{A \mid A_{1,1} \geq 0\}$ with order $l > r$ iff $A_{1,1} > B_{1,1}$. Strong matrices have $A_{1,1} > 0$.

▶ **Lemma 22.** *If a weakly simple interpretation i into uniformly well-founded $(D, >)$ is weakly compatible with a relation $\to$ and Γ is a subset of i-strong letters, then $\to$ is Γ-quasi-terminating.*

We note that to prove weak compatibility with $\to_R$, we might need a stronger order on D that is stable w.r.t. contexts. For example, matrix interpretations need $[l]_{i,j} \geq [r]_{i,j}$ elsewhere.

6 Applications

Combining results from previous sections, we obtain this method of proving termination:

- give an interpretation i into $(D, >)$ according to Lemma 22,
- give a well-founded reduction order $>$ on $\Sigma \setminus \Gamma$, and a status (L or R), according to Theorem 15.

In the second step, well-foundedness can be proved by applying the method again, on the smaller alphabet. As a special case, when the interpretation always counts occurrences of "top" letters, the standard Kachinuki order appears.

▶ **Example 23.** For $R = \{ca \to b^2c^3, bcb \to a\}$, we use interpretation $i_a(x) = x + 2, i_b(x) = x + 1, i_c(x) = 0$ on $(\mathbb{N}, >)$. Here, $\mathsf{Strong}(i) = \{a, b\}$.

We have $\mathsf{split}_{\{a,b\}}(R) = \{[c, \epsilon] \to [\epsilon, \epsilon, c^3], [\epsilon, c, \epsilon] \to [\epsilon, \epsilon]\}$. Both pairs are included in $>^{\mathsf{L}}_{\mathsf{Lex}}$ where $>$ compares lengths.

▶ **Example 24.** For $R = \{cab \to bcba, bcba \to abc\}$, we use interpretation

$$i : a \mapsto \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}, b \mapsto \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, c \mapsto \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},$$

Note that $i(a)$ is not in E_+, as defined in Example 20. We consider $D = \{(x_1, x_2) \mid x_1 \geq 1, x_2 \geq 1\}$, ordered by $(x_1, x_2) > (y_1, y_2)$ if $x_1 > y_1$. The given interpretation indeed maps D into D, and both a and b are strong.

We have $\mathsf{split}_{\{a,b\}}(R) = \{[c, \epsilon, \epsilon] \to [\epsilon, c, \epsilon, \epsilon], [\epsilon, c, \epsilon, \epsilon] \to [\epsilon, \epsilon, c]\}$ which is contained in $>^{\mathsf{L}}_{\mathsf{Lex}}$ where $>$ compares lengths.

This can be generalized: For $D = \mathbb{N}^n_{>0}$, ordered by $x > y \iff x_1 > y_1$, we can use any matrix interpretation where in each matrix, the top left entry is ≥ 1 and in each (other) line there is at least one entry ≥ 1. All these are weakly simple. If we additionally have that the top left entry is > 1, or some other entry in the top line is > 0, then the matrix is strong.

7 Extension

We mention here a variation of the method where we don't need to order syllables at all – in case we know that their number is strictly increasing.

▶ **Lemma 25.** *For a string rewriting system R on Σ and $\Gamma \subseteq \Sigma$, if $\to_R$ is Γ-quasi-terminating, and for each $(l, r) \in \mathsf{split}_\Gamma(R)$ we have that that $|l| < |r|$, then R is terminating.*

This holds true since Γ-quasi-termination bounds the number of syllables (in words reachable from any fixed starting word) from above. Given the conditions of the lemma, the distance to that bound will decrease strictly.

▶ **Example 26.** For SRS/Zantema/z050, $R = \{abbaab \to aabbaba\}$, we use the arctic interpretation

$$i : a \mapsto \begin{pmatrix} 1 & -\infty & 1 \\ -\infty & -\infty & 1 \\ 0 & -\infty & -\infty \end{pmatrix}, b \mapsto \begin{pmatrix} 0 & 0 & 1 \\ 0 & 2 & 4 \\ 1 & 0 & -\infty \end{pmatrix},$$

We verify that $i(l) \geq i(r)$ point-wise, so i is weakly compatible with rewriting, and that $i(a)$ is strong. We obtain $\mathsf{split}_a(R) = \{[\epsilon, bb, \epsilon, b] \to [\epsilon, \epsilon, bb, b, \epsilon]\}$ which is length-increasing. By Lemma 22 and Lemma 25, R is terminating.

8 Discussion

Related work. If all symbols have identical status (e.g., from the left), then the concept of semantic Kachinuki order presented here could be related to the concept of semantic path order [2], or to semantic labelling [9] w.r.t. a quasi-model, as the interpretation that bounds occurrences of strong letters is a quasi-model. Our precedence is still on the original alphabet, not on the semantically labelled one. Still, semantic path order or semantic labelling could probably not handle mixed status.

Implementation. The constraint system that describes the interpretation that determines strong symbols is easy to write. For matrices, we might use SMT over bit-vectors (BV). This will produce candidates for subsets Γ of strong letters (there may be several). We have a prototypical implementation, with obvious extensions for relative termination, as part of the Matchbox termination prover `https://gitlab.imn.htwk-leipzig.de/waldmann/pure-matchbox`. Performance on the Termination Problems Data Base is weak, but we expect this to improve (as with all methods that use interpretations) by enlarging the alphabet via pre-processing, e.g., root labelling [8]. Typically, this will result in larger systems, so the search for semantic (Kachinuki) orders will be challenging.

Extension. Splitting w.r.t. Γ reduces termination of R to a Boolean combination of termination sub-problems. For example, assume R is Γ-quasi-terminating, and $\mathsf{split}_\Gamma(R) = \{[l_1, l_2] \to [r_1, r_2, r_3]\}$. Using standard notation $\mathsf{SN}(R/S)$ for relative termination, we see that $\mathsf{SN}(l_1 \to r_1) \vee \mathsf{SN}(l_2 \to r_2 / l_1 \to r_1)$ (lexicographic decrease from the left) but also $\mathsf{SN}(l_2 \to r_3) \vee \mathsf{SN}(l_1 \to r_2 / l_2 \to r_3)$ (from the right) imply $\mathsf{SN}(R)$. With more rules in R, we obtain a conjunction of such formulas. We might handle this inside one SMT encoding (of the conjunction), or we can translate to disjunctive normal form, where clauses are standard (relative) termination problems that could be tried concurrently.

References

1 Nachum Dershowitz. Termination of Rewriting. *J. Symb. Comput.*, 3(1/2):69–116, 1987.

2 Alfons Geser. An Improved General Path Order. *Appl. Algebra Eng. Commun. Comput.*, 7(6):469–511, 1996.

3 Dieter Hofbauer. On the derivational complexity of Kachinuki orderings. In *Intl. Workshop on Termination*, 2014.

4 Dieter Hofbauer and Johannes Waldmann. Termination of String Rewriting with Matrix Interpretations. In Frank Pfenning, editor, *Term Rewriting and Applications, 17th International Conference, RTA 2006, Seattle, WA, USA, August 12-14, 2006, Proceedings*, volume 4098 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2006.

5 Adam Koprowski and Johannes Waldmann. Max/Plus Tree Automata for Termination of Term Rewriting. *Acta Cybern.*, 19(2):357–392, 2009.

6 Ko Sakai. Knuth-Bendix Algorithm for Thue System Based on Kachinuki Ordering. ICOT Technical Memorandum TM-0087, Institute for New Generation Computer Technology, December 1984.

7 Joachim Steinbach. Comparing on Strings: Iterated syllable ordering and recursive path orderings. SEKI Report SR-89-15, Universität Kaiserslautern, 1989.

8 Christian Sternagel and Aart Middeldorp. Root-Labeling. In Andrei Voronkov, editor, *Rewriting Techniques and Applications, 19th International Conference, RTA 2008, Hagenberg, Austria, July 15-17, 2008, Proceedings*, volume 5117 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2008.

9 Hans Zantema. Termination of Term Rewriting by Semantic Labelling. *Fundam. Inform.*, 24(1/2):89–105, 1995.

GPO: A Path Ordering for Graphs

Nachum Dershowitz[1] and Jean-Pierre Jouannaud[2]

1 **School of Computer Science, Tel Aviv University, Tel Aviv, Israel**
2 **LIX & Deducteam, École Polytechnique, Palaiseau, France**

Abstract

We define well-founded monotonic rewrite orderings on graphs inspired by the recursive path ordering on terms. Our graph path ordering applies to finite, directed, ordered multigraphs and provides a building block for rewriting with such graphs, which should impact the many areas in which computations take place on graphs.

1 Introduction

We are interested in well-founded orderings that can be used to show termination of rewriting on first-order terms having both sharing and back-arrows, which is another way of saying that we study rewriting of rooted (multi-) graphs, each vertex of which is labeled by a function symbol whose arity governs the number of vertices it points to. Different target applications require different properties of the ordering: totality is crucial for operaders, while monotonicity with respect to graph structure is important for rewriters.

We propose a generalization of the recursive path ordering to ordered, labeled, rooted graphs. The graph ordering we end up with, *GPO*, has the very same definition as the recursive path ordering (RPO), but the computation of the "head" and "tail" of a multigraph shares little resemblance with the case of trees, for which the head is the top function symbol labeling the root of the tree and the tail is the list of its subtrees. GPO has many of the properties that are important for its various potential users. It is well-founded, total on graph expressions up to isomorphism, and monotonicity is testable.

Graph rewriting has been richly studied. Path orderings of term graphs were developed in [3]. Graph decomposition and weight-based orderings for cyclic graphs are explored in [1]. By adding a root structure to cyclic graphs, we get a natural decomposition into head and tail and a concomitant path ordering.

2 Drags

We work with *drags*, which are finite ***d***irected multi-***r***ooted vertex-l***a***beled ***g***raphs with multiple edges. We begin with a brief presentation of the algebra of drags, developed more fully in [2].

We presuppose a set of function symbols Σ, whose elements are used as vertex labels and are equipped with a fixed arity, plus a denumerable set of (nullary) *variable* symbols Ξ disjoint from Σ. Outgoing edges are ordered (from left to right, say) and their quantity depends solely on the label of the vertex. An "open" drag will include variables, while a "closed" one won't. The successors of a vertex labeled $f \in \Sigma$ in a drag are understood as the arguments of the function symbol f – in order. In contrast with the common categorical approach to graph rewriting, this multigraph model is quite standard. The novelty is that we allow for arbitrarily many roots and arbitrarily complex cycles.

A *(closed) drag* is a tuple $\langle V, R, L, X\rangle$, where V is a finite set of *vertices*; R is a finite (ordered) list of vertices, called *roots*, with $R(n)$ denoting the nth root in the list; $L : V \to \Sigma$ is the *labeling* function, mapping vertices to labels from the vocabulary Σ; and $X : V \to V^*$

© N. Dershowitz and J.-P. Jouannaud;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

is the *successor* function, mapping each vertex $v \in V$ to a list of vertices in V whose length equals the appropriate arity. Given $b \in X(a)$, we say that there is an *edge* from *source* a to *target* b and write aXb. The reflexive-transitive closure X^* of relation X is *accessibility*. Vertex v is *accessible* if it is accessible from a root $r \in R$ (rX^*v). A drag is *clean* if all its vertices are accessible. A root r is *maximal* if it can access all roots its accessible from ($\forall r' \in R.\ r'X^*r$ implies rX^*r').

An *open drag* is a drag over $\Sigma \cup \Xi$. The vertices labeled by a function symbol in Σ are *internal*; those labeled by a variable are called *sprouts*. When nonempty, the set S of sprouts will be added at the end of the tuple: $\langle V, R, L, X, S\rangle$. An open drag is termed *linear* if different sprouts have different labels. It is *cyclic* if each internal vertex accesses a root ($\forall v \in V \setminus S.\ \exists r \in R.\ vX^*r$) and it has no variable roots ($R \cap S = \varnothing$).

Given drag D, $\mathcal{A}cc(D)$ is its set of accessible vertices; $\mathcal{R}(D)$, its list of roots; $\mathcal{R}^\bullet(D) \subseteq \mathcal{R}(D)$ are the *maximal* roots; $[R]$ are the numbers $[1\,..\,|R|]$ referring to the roots in order; $\mathcal{S}(D)$ are its sprouts; $\mathcal{V}ar(D) = L(\mathcal{S}(D))$ are the variables labeling its sprouts; and $D^\sharp$ is the *clean* drag obtained by removing inaccessible vertices and associated edges. Drag D is *(quasi-) isomorphic* to drag D', indicated $D \cong D'$ (or $D \simeq D'$ in the quasi case) if there is a graph isomorphism between them that respects roots (as multisets in the quasi case) and sprouts up to renaming of labels.

Given drag $D = \langle V, R, L, X, S\rangle$ and a subset $W \subseteq V$ of its vertices, the *subdrag* $D_{|W}$ of D generated by vertices W is the drag $\langle V', R', L', X', S'\rangle$ where V' is the least superset of W that is closed under X; L', X', S' are the restrictions of L, X, S to V'; and R' is $(R \cap V') \cup (X(V \setminus V') \cap V')$. A subdrag is clean by construction. Its roots are obtained by adding as new roots those vertices that have an incoming edge in D that is not in $D_{|W}$. The order of elements in this additional list does not really matter for now. In particular, restricting a drag D to its maximal roots, yields $D^\sharp$ ($= D_{|\mathcal{R}^\bullet(D)}$).

The key to working with drags is that we can equip them with a parameterized binary composition operator that connects sprouts of each of two drags with roots of the other. Denote by $\mathcal{D}om(\xi)$ and $\mathcal{I}m(\xi)$ the *domain (of definition)* and *image* of a (partial) function ξ, using $\xi_{A\to B}$ for its restriction going from $A \subseteq \mathcal{D}om(\xi)$ to $B \subseteq \mathcal{I}m(\xi)$, omitting $\to B$ when irrelevant. Let $D = \langle V, R, L, X, S\rangle$ and $D' = \langle V', R', L', X', S'\rangle$ be open drags. A *switchboard* $\xi : S \cup S' \to \mathbb{N}$ for D, D' splits into a pair $\langle \xi_D : S \to [R'], \xi_{D'} : S' \to [R]\rangle$ of partial injective functions such that (i) $\forall s, t \in S.\ s \in \mathcal{D}om(\xi)$ and $L(s) = L(t)$ imply $t \in \mathcal{D}om(\xi)$ and $D_{|R'(\xi(s))} \simeq D_{|R'(\xi(t))}$; (ii) $\forall s', t' \in S'.\ s' \in \mathcal{D}om(\xi)$ and $L'(s') = L'(t')$ imply $t' \in \mathcal{D}om(\xi)$ and $D'_{|R(\xi(s'))} \simeq D'_{|R(\xi(t'))}$. We also say that $D'\xi$ is an *extension* of D.

Each switchboard induces a binary composition operation on open drags, the precise definition of which we omit (but see [2]). The essence is that the (disjoint) union of the two drags is formed, with sprouts in the domain of the switchboards merged with the vertices referred to in its images. This necessitates renaming targets of edges that had pointed to sprouts.

A switchboard ξ for D, D' is *directed* if one of ξ_D and $\xi_{D'}$ has an empty domain. Directed switchboards correspond to the tree case, with all connections from one of the drags to the other.

▸ **Lemma 1** (Unique Decomposition). *Given a drag D and a subset of its vertices W, there exists a drag A, called its* antecedent*, and a directed switchboard ξ such that $D = A\,\xi\,D_{|W}$.*

The fact that the switchboard ξ is directed expresses the property that decomposing a drag into a subdrag and its antecedent does not break any of its cycles. If it's cyclic, it's its own antecedent.

Just as a tree can be decomposed into a head node f and subtrees, a drag has a head, viz. its smallest (nontrivial) antecedent, and one tail, possibly a list of several connected components. The *tail* $\triangledown D$ of a drag $D = \langle V, R, L, X, S\rangle$ is the subdrag generated by the set of vertices $V \setminus \{v \in V : vX^*\mathcal{R}^\bullet(D)\}$. Furthermore, for drag D, there exists a linear drag $\widehat{D}$, the *head* of D, and a directed switchboard ξ such that $D = \widehat{D}\,\xi\,\triangledown D$. The head of a drag is therefore the antecedent of its tail.

Isomorphic drags have isomorphic heads and tails. The crucial point is that decomposition into head and tail is canonical and faithful because drags are multi-rooted. Uni-rooted drags cannot represent horizontal sharing, in contrast to vertical sharing which can *sometimes* be preserved with uni-rooted drags.

3 Drag Rewriting

An extension $B\xi$ is a *context* of drag D when $\mathcal{D}om(\xi_D) = \varnothing$. It's a *substitution* when $\mathcal{D}om(\xi_B) = \varnothing$. It is a *rewriting* extension if ξ_B is linear and surjective and ξ_D is total. An extension $B\xi$ of a clean drag D is *cyclic* if B is a linear drag generated by $\mathcal{I}m(\xi_D)$, $B\,\xi\,D$ is a clean nonempty drag, and there exists $s \in \mathcal{D}om(\xi_B)$ such that tX^*s for all $t \in V$. The extension is *trivial* if B is an identity drag (a linear open drag all of whose vertices are sprouts and sans edges) and *total* if ξ_B is surjective.

The conditions for being a cyclic extension impose that ξ_D is surjective on $R \setminus S$ as a set so as to generate B. *Identity cyclic extensions* of D are such that its sprout variables are in one-to-one correspondence with the sprouts in D that are connected by the switchboard and connect them to some of the roots of D. Thus, cyclic extensions modify the structure of D by connecting some of its sprouts to some of its roots. Unfortunately, identity cyclic extensions suffice for that purpose only in special cases. Identity extensions allow one to change the structure of a drag without changing its internal nodes. If the drag has a single root, it is easy to see that identity extensions are enough to predict all forms that a drag may take under composition with an extension. This is not true for multirooted drags, since identity cyclic extensions cannot reach two different roots from the same sprout.

▸ **Theorem 2.** *Let D, E be clean nonempty drags and ξ be a switchboard for them. If ξ_E is surjective on maximal roots $\mathcal{R}^\bullet(D)$, then there exist drags A, B, C and switchboards ζ, θ such that (i) $B\langle\xi_B, \xi_{D\to B}\rangle$ is a cyclic extension of D; (ii) $C\theta$ is a substitution extension of $B\,\xi\,D$; (iii) $A\zeta$ is a context extension of $(B\,\theta\,D)\,\eta\,C$; (iv) $E\,\xi\,D = A\,\zeta\,((B\,\theta\,D)\,\eta\,C)$; (v) C is empty if all internal nodes of E reach one of its sprouts.*

Can a drag D be seen as the composition of a given drag G with some context C via some switchboard ξ? In this case, we say that D *matches* G, E and ξ being the *matching* context and switchboard. The idea is that E splits into three: vertices that are accessible from some sprout of G and can access some of its roots define a drag B; those accessible from some sprout of G but which cannot access any of its roots define a drag C that is a substitution extension of G; those which are not accessible from the sprouts of G form the remaining part A, which is a context extension of G.

A *graph rewrite rule* is a pair of open clean drags written $G \to G'$ such that $|\mathcal{R}(G)| = |\mathcal{R}(G')|$ and $\mathcal{V}ar(G') \subseteq \mathcal{V}ar(G)$. We say that a nonempty clean drag D *rewrites* to a clean drag D' via this rule, and write $D \longrightarrow D'$, if $D = E\,\xi\,G$ and $D' = (E\,\xi\,G')^\natural$ for some rewriting extension $E\xi$ of G, such that ξ_G is linear if G is. We have: (1) If $D \longrightarrow D'$, then $\mathcal{V}ar(D') \subseteq \mathcal{V}ar(D)$. (2) If $D \longrightarrow D'$ is a rewrite, C is an open drag, and ξ is a switchboard for C, D, then ξ (restricted to sprouts of D') is also a switchboard for C, D'. (3) If $D \longrightarrow D'$, then

$D = A\,\zeta\,((B\,\xi\,L)\,\theta\,C)$ and $D' \cong A\,\zeta\,((B\,\xi\,L)\,\theta\,C)$ for some A, B, C and ζ, ξ, θ such that $A\zeta$, $B\xi$ and $C\theta$ are context, cyclic and substitution extensions, respectively.

A *graph rewrite system* is a set of graph rewrite rules, each of which can be used to rewrite.

4 Graph Path Ordering

A *reduction ordering* is a well-founded ordering $\succ$ of the set of drags that is (i) *compatible* with drag isomorphism: for all D, D', E, E' such that $D \succ E$, $D \cong D'$ and $E \cong E'$, then $D' \succ E'$; (ii) *monotonic*: for all D, E such that $D \succ E$ and for all context extensions $A\xi$ of D, then $A\,\xi\,D \succ A\,\xi\,E$; and (iii) *stable*: for all D, E such that $D \succ E$ and for all substitution extensions $C\xi$ of D, then $D\,\xi\,C \succ E\,\xi\,C$. Apart from compatibility with isomorphism, the notion of reduction ordering is the same as the usual one. Monotonicity is the usual property since a directed switchboard turns the context A into a usual context. Stability corresponds to the usual stability property, but substitution extensions can introduce sharing.

▸ **Theorem 3** (Termination). *A graph rewrite system $\mathcal{R}$ terminates iff there's a graph reduction ordering $\succ$ such that, for all rules $G \to G' \in \mathcal{R}$ and cyclic extensions $B\xi$ of G, we have $B\,\xi\,G \succ B\,\xi\,R$.*

We need the analog for drags of the precedence on function symbols used by RPO: A *head order* for a drag rewrite system $\mathcal{R}$ is a quasi-order $\succcurlyeq$ on clean cyclic drags whose strict part $\succ$ is well-founded.

▸ **Definition 4** (Graph Path Order (GPO)). Given two drags s, t and a head order $\succcurlyeq$, we define $s \succ t$ (under GPO), iff any of the following three cases hold:

$$[\nabla]\ \nabla s \succcurlyeq t \qquad [\rhd]\ \widehat{s} \succ \widehat{t} \text{ and } s \succ \nabla t \qquad [\doteq]\ \widehat{s} \doteq \widehat{t},\ s \succ \nabla t \text{ and } \nabla s \succ \nabla t.$$

GPO is defined by induction on the pair $\langle s, t\rangle$ via the lexicographic extension of the subdrag order imposed by tails. Let $t \rhd u$ if $u = \nabla t$ and let $\rhd^+$ be its transitive closure.

▸ **Lemma 5** (Subdrag Properties). *(1) The subdrag order $\rhd^+$ is well-founded on nonempty open drags. (2) The empty drag is minimal in $\succ$. (3) $\rhd \subseteq \succ$. (4) If $D \succ E$, then $\mathcal{V}ar(D) \subseteq \mathcal{V}ar(E)$. (5) GPO ($\succ$) is transitive.*

GPO is a strict ordering compatible with drag isomorphism. This justifies our way of building compatibility into a path ordering via a careful definition of subdrags instead of using a normal-form representation of congruence classes of drags. If the drag ∇t is terminating with respect to $\succ$, then the drag $t = g(\overline{t})$ is.

A head order is *compatible* if two cyclic drags that are isomorphic up to their lists of roots are equivalent in the quasi-order. It is *total* if $\succcurlyeq$ is total and its equivalence $\doteq$ is exactly cyclic drag isomorphism up to their lists of roots. It is *subcyclic* if $D \succcurlyeq A$ whenever $D = A\,\xi\,B$ for a cyclic drag B that is not isomorphic to an identity. And it is *cyclic monotonic* if for all cyclic drags D, E and for all cyclic extensions $C\xi$ of D, $D \succcurlyeq E$ implies $C\,\xi\,D \succcurlyeq C\,\xi\,E$. Cyclic monotonicity tells us that the order between cycles is preserved by growing them. Monotonicity/stability of drag orders and cyclic monotonicity of head orders are complementary: the former extends a drag by preserving its head, while the latter extends heads.

▸ **Theorem 6** (GPO is Good). *(1) GPO is a rewrite ordering: it is monotonic and it is stable (if $D \succ E$, then $D\,\xi\,C \succ E\,\xi\,C$ for all substitution extensions $C\xi$ of D). (2) GPO is*

well-founded, which makes it a reduction ordering. (3) GPO is total on equivalence classes of drags (modulo isomorphism) if the head order is total.

We define two head orders, one total but not subcyclic, and another that is subcyclic but partial.

Let $\geq$ be a total precedence on Σ, whose strict part $>$ is well-founded, and $D = \langle V, R, L, X\rangle$, a clean drag from which the sprouts and their incoming edges have been removed. Represent a drag as a list of function symbols. The *interpretation* $[\![D]\!]$ of $D = \langle V, R, L, X\rangle$, is a list of symbols in Σ defined as follows: If $\mathcal{A}cc(D) = \varnothing$, then $[\![D]\!] \coloneqq \varnothing$. Otherwise, $[\![D]\!] \coloneqq L(r) \cup [\![W]\!]$, where $R = r \cup R'$ and $W = \langle V \setminus r, (X(r) \setminus r) \cup R', L', X'\rangle$, L', X' being the restrictions of L, X to V', respectively. In words, $[\![D]\!]$ collects the function symbols labeling the internal nodes of a drag by traversing this drag in a depth-first manner starting from R. We can now define our head order on drags: $D \geqslant D'$ iff $\langle |[\![D]\!]|, [\![D]\!]\rangle \ (\geq_{\mathbb{N}}, \geq_{lex})_{lex} \ \langle |[\![D']\!]|, [\![D']\!]\rangle$. The relation $\geqslant$ is a total head order. This head order is not subcyclic, nor cyclic-monotonic.

Alternatively, represent a drag as the multiset of function symbols labeling its accessible vertices: Define the *interpretation* as the multiset of symbols that label the vertices of $D^\sharp$. Given two drags, we define: $D \geqslant D'$ iff $[\![D]\!] \geq_{mul} [\![D']\!]$. This $\geqslant$ is a subcyclic, cyclic-monotonic head order.

▸ **Example 7.** Consider an application of a rule $G = f(x) \to a = G'$, where f and a are drag labels and x is a variable, to the cyclic graph $D = \ \to f \leftrightarrow f$ leading to the noncyclic term $D' = f(a)$. The resultant inequality is $D = \langle \{1,2\}, 1, \{1, 2 \mapsto_L f\}, \{1 \mapsto_X 2, 2 \mapsto_X 1\}\rangle \succ \langle \{1,2\}, 1, \{1 \mapsto_{L'} f, 2 \mapsto_{L'} a\}, 1 \mapsto_{X'} 2\rangle = D'$, foregoing braces around singletons. The first drag is its own head. The head of the second is $f(y) = \langle \{1,2\}, 1, \{1 \mapsto_{L'} f, 2 \mapsto_{L'} y\}, 1 \mapsto_{X'} y, 2\rangle$, which is a subdrag of the first. By the subterm property of head orders, the first is strictly larger than the second in $\geqslant$. Therefore, we are left with a subgoal: $D \succ \nabla D' = a = \langle 2, 2, 2 \mapsto_{L''} a, \varnothing\rangle = D''$. This time, the second drag is also itself a head, but it is not a subdrag of the first. Therefore, we must have $D \succcurlyeq D''$ in the head order for this constraint to pass. A head order with precedence $f > a$ does the trick. ◂

▸ **Theorem 8** (Decidability). *It is decidable whether a system $\mathcal{R}$ terminates under GPO provided the universal first-order fragment of head-order constraints is decidable.*

References

1 Guillaume Bonfante and Bruno Guillaume. Non-simplifying graph rewriting termination. In Rachid Echahed and Detlef Plump, editors, *Proceedings of the 7th International Workshop on Computing with Terms and Graphs (TERMGRAPH)*, pages 4–16, Rome, Italy, March 2013.

2 Nachum Dershowitz and Jean-Pierre Jouannaud. Drags: A simple algebraic framework for graph rewriting. In *Proceedings of the 10th International Workshop on Computing with Terms and Graphs (TERMGRAPH)*, Oxford, UK, July 2018.

3 Detlef Plump. Simplification orders for term graph rewriting. In Igor Prívara and Peter Ruzicka, editors, *Proceedings of the 22nd International Symposium on Mathematical Foundations of Computer Science (MFCS'97)*, volume 1295 of *Lecture Notes in Computer Science*, pages 458–467, Bratislava, Slovakia, August 1997. Springer.

A Perron–Frobenius Theorem for Jordan Blocks for Complexity Proving

Jose Divasón[1], Sebastiaan Joosten[2], René Thiemann[3], and Akihisa Yamada[4]

1 **Universidad de La Rioja, Spain**
2 **University of Twente, the Netherlands**
3 **University of Innsbruck, Austria**
4 **National Institute of Informatics, Japan**

1 Introduction

Matrix interpretations [4] are widely used in automated complexity analysis of term rewrite systems. We consider the approach where from a matrix interpretation one extracts a nonnegative square matrix $A \in \mathbb{R}^{n\times n}_{\geq 0}$ such that $\mathcal{O}(\max_{1\leq i,j\leq n}(A^k)_{ij} \cdot k)$ is a bound for the interpretation of a (specific) term of size k [8]. Hence, the growth rate of the entries of A^k can be used to determine an upper bound on the complexity of the rewrite system.

In this paper we concentrate on algebraic means to investigate the growth rate of A^k. We use the following notions: χ_A denotes the characteristic polynomial of matrix A, eigenvalues are the roots of χ_A, maximal eigenvalues are those of the maximum norm, and the maximum norm is the spectral radius ρ_A. A Jordan block for some eigenvalue λ is a square matrix B with $B_{ii} = \lambda$, $B_{i(i+1)} = 1$, and $B_{ij} = 0$ otherwise. We write $B(s, \lambda)$ to indicate that a Jordan block has size s and eigenvalue λ. A diagonal composition J of Jordan blocks is a Jordan normal form of A if $A = PJP^{-1}$ for some invertible matrix P; J is unique up to permutation of the Jordan blocks, so we call them *the Jordan blocks of* A.

Jordan blocks are an important utility to characterize matrix growth, cf. Theorem 1. This theorem was formalized in Isabelle/HOL [10, 12] when formalizing the complexity criteria provided by Neurauter et al. [7, 9]. It was used in CeTA [13] until year 2017 in order to check matrix growth rates.

▶ **Theorem 1** (Matrix Growth via Algebraic Methods). *Let* $A \in \mathbb{C}^{n\times n}$.

- *The entries in* A^k *are polynomially bounded in* k *iff* $\rho_A \leq 1$.
- *The entries in* A^k *are bounded by* $\mathcal{O}(k^d)$ *iff* $\rho_A \leq 1$ *and for all eigenvalues* λ *of* A *with* $|\lambda| = 1$ *and for all Jordan blocks* $B(s, \lambda)$ *of* A, $s \leq d + 1$.

Using Theorem 1 it seems easy to decide the asymptotic growth rate of a matrix: one can compute all eigenvalues, compute their maximal norms to test $\rho_A \leq 1$, and then compute all Jordan blocks of A for eigenvalues with norm 1 and determine their maximal size.

The problem is that in general the eigenvalues of a matrix are quite complex to compute. Even if all matrix entries are integers or rational numbers, the eigenvalues are complex algebraic numbers. Hence, applying Theorem 1 directly requires expensive algebraic number arithmetic.

The aim of this work is to reduce the complexity of applying Theorem 1 for an important class of matrices: non-negative real square matrices $\mathbb{R}^{n\times n}_{\geq 0}$.

Several properties of non-negative real matrices are studied by Perron and Frobenius [5, 11]. We already formalized several of them in developing an efficient certifier for checking matrix growth, for instance Theorem 2 [2].

© Jose Divasón, Sebastiaan Joosten, René Thiemann, and Akihisa Yamada;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

▶ **Theorem 2.** *The characteristic polynomial χ_A of $A \in \mathbb{R}^{n\times n}_{\geq 0}$ can be factored into*

$$\chi_A(x) = f(x) \cdot \prod_{k\in K} (x^k - \rho_A^k)$$

where K is a non-empty multiset of positive integers, and f is a polynomial whose complex roots have a norm strictly less than ρ_A.

Note that Theorem 2 has at least three implications. First, ρ_A itself is an eigenvalue, so $\rho_A \leq 1$ is the same as demanding that χ_A has no *real* eigenvalue above 1—a criterion which can be efficiently decided by Sturm's method. Second, it permits us to replace the computation of *all* eigenvalues in Theorem 1 by a computation which only involves roots of unity of degree at most n. Third, ρ_A has the maximum algebraic multiplicity among all maximal eigenvalues.

For a precise and efficient growth rate analysis, in this paper we further prove a conjecture in [2, Section 7]: ρ_A also has the largest Jordan blocks among all maximal eigenvalues. Previously we only had a partial result for matrices of dimension up to 4.

We also derive some consequences in Section 3.

2 Largest Jordan Blocks for Non-Negative Real Matrices

The following Theorem 3 is the key new result of this paper. It has been formalized in Isabelle/HOL and will be available in the archive of formal proofs [3, for Isabelle 2018].

▶ **Theorem 3.** *Let $A \in \mathbb{R}^{n\times n}_{\geq 0}$ and λ be a maximal eigenvalue of A. For every Jordan block $B(s,\lambda)$ of A, there exists a Jordan block $B(t,\rho_A)$ of A with $t \geq s$.*

Proof. W.l.o.g. we assume $\rho_A = 1$, since otherwise we can just perform a scalar multiplication of A by $\frac{1}{\rho_A}$ (The case $\rho_A = 0$ is trivial).

Let $A = PJP^{-1}$ be the Jordan decomposition of A. Let e_i denote the i-th unit-vector. Let E be the set of maximal eigenvalues of A, i.e., eigenvalues λ with $|\lambda| = 1$. Let m be the size of a largest Jordan block among all blocks with eigenvalues in E. By Theorem 2 all eigenvalues λ with $|\lambda| = 1$ are roots of unity, i.e., we have a positive integer d_λ such that $\lambda^{d_\lambda} = 1$.

Define

$$c = \prod_{0\leq i<m-1} (m-1-i) \qquad D = \prod_{\lambda\in E} d_\lambda \qquad K_\ell\ k = D\cdot k + \ell + m - 1$$

It is easy to prove $c > 0$. Moreover, if $\lambda \in E$, then $\lambda^{K_\ell\ k}$ does not depend on k.

$$\lambda^{K_\ell\ k} = (\lambda^D)^k \cdot \lambda^\ell \cdot \lambda^{m-1} = 1^k \cdot \lambda^\ell \cdot \lambda^{m-1} = \lambda^\ell \cdot \lambda^{m-1}.$$

Moreover for any Jordan Block B of size s with eigenvalue λ of A and any valid position of B we get the following limit:

$$\frac{c(B^{K_\ell\ k})_{ij}}{(K_\ell\ k)^{m-1}} = \frac{c\binom{K_\ell\ k}{j-i}\lambda^{K_\ell\ k+i-j}}{(K_\ell\ k)^{m-1}} \xrightarrow{k\to\infty} \begin{cases} \lambda^\ell & \text{if } m = s = j,\ i = 1, \text{ and } \lambda \in E \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

The equality in (1) is just the closed form for powers of Jordan blocks, and we shortly illustrate by a case analysis why the limit is correct.

- If $\lambda \notin E$ then there is an exponential decrease in the $\lambda^{\cdots k \cdots}$ expression whereas the remaining expression grows at most polynomial. So the limit will be 0.
- If $\lambda \in E$, but $m = j \wedge i = 1$ does not hold, then the norm of the numerator will be a polynomial of smaller degree than the polynomial in the denominator. So, again the limit will be 0.
- Finally, if $m = j$, $i = 1$, and $\lambda \in E$, then both the numerator and the denominator will be polynomials of the same degree, and c has been defined in a way that the limit will be precisely λ^ℓ.

The next step is to lift (1) from single Jordan blocks to the complete Jordan normal form J. Let us define I as the set of last-row indices of those Jordan blocks of J which have size m and a maximal eigenvalue. So, in particular I is non-empty. Then from (1) we immediately get the following consequence where $\lambda_i = J_{ii}$ is the eigenvalue corresponding to row i.

$$\frac{c(J^{K_\ell\ k})_{ij}}{(K_\ell\ k)^{m-1}} \overset{k\to\infty}{\longrightarrow} \begin{cases} \lambda_j^\ell & \text{if } j \in I \wedge i = j - (m-1) \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Define

$$\begin{aligned} i &= \text{some element of } I \\ j &= \text{some index } j \text{ such that } P_{j(i-(m-1))} \neq 0 \\ v &= |Pe_i| \end{aligned}$$

Then

$$\begin{aligned} \frac{c(A^{K_\ell\ k}v)_j}{(K_\ell\ k)^{m-1}} &= \frac{c(|A^{K_\ell\ k}| \cdot |v|)_j}{(K_\ell\ k)^{m-1}} = \frac{c(|A^{K_\ell\ k}| \cdot |Pe_i|)_j}{(K_\ell\ k)^{m-1}} \\ &\geq \frac{c|A^{K_\ell\ k}Pe_i|_j}{(K_\ell\ k)^{m-1}} = \frac{c|PJ^{K_\ell\ k}e_i|_j}{(K_\ell\ k)^{m-1}} \\ &\overset{k\to\infty}{\longrightarrow} |P_{j(i-(m-1))} \cdot \lambda_i^\ell| = |P_{j(i-(m-1))}| > 0 \end{aligned}$$

Thus, there exists some $b > 0$ and such that for all ℓ and all sufficiently large k:

$$\frac{c(A^{K_\ell\ k}v)_j}{(K_\ell\ k)^{m-1}} \geq b \tag{3}$$

Define

$$\begin{aligned} u &= P^{-1}v \\ a_i &= P_{j,i-(m-1)} \cdot u_i \qquad \text{for } j \text{ as above and arbitrary } i \end{aligned}$$

Then by using (2) conclude

$$\begin{aligned} \frac{c(A^{K_\ell\ k}v)_j}{(K_\ell\ k)^{m-1}} &= Re\left(\frac{c(A^{K_\ell\ k}v)_j}{(K_\ell\ k)^{m-1}}\right) = Re\left(\frac{c(A^{K_\ell\ k}Pu)_j}{(K_\ell\ k)^{m-1}}\right) \\ &= Re\left(\frac{c(PJ^{K_\ell\ k}u)_j}{(K_\ell\ k)^{m-1}}\right) \overset{k\to\infty}{\longrightarrow} Re\left(\sum_{i\in I} a_i\lambda_i^\ell\right) \end{aligned} \tag{4}$$

By combining (4) with (3) we arrive at

$$Re\left(\sum_{i\in I} a_i\lambda_i^\ell\right) > 0 \tag{5}$$

The theorem is equivalent to the statement $\lambda_i = 1$ for some $i \in I$, so let us assume to the contrary that $\lambda_i \neq 1$ for all $i \in I$. From (5) we conclude

$$0 < Re\left(\sum_{\ell=0}^{D-1}\sum_{i\in I} a_i\lambda_i^\ell\right)$$

and hence

$$\begin{aligned} 0 \neq \sum_{\ell=0}^{D-1}\sum_{i\in I} a_i\lambda_i^\ell &= \sum_{i\in I} a_i \sum_{\ell=0}^{D-1}\lambda_i^\ell = \sum_{i\in I} a_i \frac{D}{d_{\lambda_i}} \cdot \sum_{\ell=0}^{d_{\lambda_i}-1}\lambda_i^\ell \\ &= \sum_{i\in I} a_i \frac{D}{d_{\lambda_i}} \cdot \frac{\lambda_i^{d_{\lambda_i}}-1}{\lambda_i - 1} = \sum_{i\in I} a_i \frac{D}{d_{\lambda_i}} \cdot 0 = 0 \end{aligned}$$

where the formula for the geometric sum can only be applied since $\lambda_i \neq 1$. By this contradiction we have finished the proof. ◀

3 Consequences of Theorem 3

We can immediately connect Theorem 3 with Theorem 1 in order to get an improved complexity criterion in the form of Algorithm 1. Here, I is the identity matrix.

The big advantage of this algorithm is that it only employs arithmetic operations, i.e., in particular if A is a matrix with rational entries, then it only involves rational arithmetic.

Algorithm 1: Efficient Certification of $\max_{1\leq i,j\leq n}(A^k)_{ij} \in \mathcal{O}(k^d)$.

Input: $A \in \mathbb{R}_{\geq 0}^{n\times n}$ and degree d.
Output: Accept or assertion failure.

1 Assert $\{x \in \mathbb{R}.\chi_A(x) = 0, x > 1\} = \emptyset$ via Sturm's method
2 Compute m as the multiplicity of root 1 of χ_A
3 **if** $m \leq d+1$ **then** Accept
4 Assert that all Jordan blocks $B(s,1)$ of A have a size $s \leq d+1$. This can be decided by checking that the kernel dimension of $(A-I)^{d+1}$ is m
5 Accept

To measure improvements in practice, we extracted all matrix interpretations from complexity proofs of the international termination and complexity competition [6] in the last three years, which amounts to the validation of the growth rate of 6,690 matrices, whose largest dimension was only 5. This low dimension keeps the overhead of algebraic number computations at a reasonable level. Still, processing all 6,690 matrices became five times faster when using Algorithm 1 instead of Theorem 1 and algebraic number computations.

Since the algorithm runs in polynomial time, it also seems to be possible to use our algorithm for synthesizing matrix interpretations; one can write a polynomial-sized SAT or SMT encoding of whether a symbolic matrix has an *a priori* fixed growth rate by a symbolic execution of Algorithm 1. The required algorithms are

- computation of characteristic polynomial χ_A
- polynomial division and polynomial GCD for applying Sturm's method
- Gauss-Jordan elimination for computing the kernel dimension, and matrix multiplication

Developers of the complexity tool TCT [1] are currently investigating this possibility.

Although our results facilitate precise estimation of the asymptotic growth rate of matrix powers, the reduction of complexity analysis to matrix powers remains approximative. Therefore, it is an important future work to generalize our formalization to *joint* spectral radius, in order to facilitate precise complexity analysis [7].

Acknowledgments

We thank Hans Zwart for discussions on Theorem 3. He came up with an alternative proof at a stage where already a large part of our proof has been formalized.

This research was supported by the Austrian Science Fund (FWF) project Y757. Jose Divasón is partially funded by the Spanish projects MTM2014-54151-P and MTM2017-88804-P. Akihisa Yamada is partially supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST. The authors are listed in alphabetical order regardless of individual contributions or seniority.

References

1 Martin Avanzini, Georg Moser, and Michael Schaper. TcT: Tyrolean Complexity Tool. In *TACAS 2016*, volume 9636 of *LNCS*, pages 407–423, 2016.

2 Jose Divasón, Sebastiaan Joosten, Ondřej Kunčar, René Thiemann, and Akihisa Yamada. Efficient certification of complexity proofs: Formalizing the Perron–Frobenius theorem (invited talk paper). In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 2–13. ACM, 2018.

3 Jose Divasón, Ondřej Kunčar, René Thiemann, and Akihisa Yamada. Perron–Frobenius theorem for spectral radius analysis. *Archive of Formal Proofs*, May 2016. `http://isa-afp.org/entries/Perron_Frobenius.shtml`, Formal proof development.

4 Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. *Journal of Automated Reasoning*, 40(2-3):195–220, 2008.

5 Ferdinand Georg Frobenius. Über Matrizen aus nicht negativen Elementen. In *Sitzungsberichte Preuß. Akad. Wiss.*, pages 456–477, 1912.

6 Jürgen Giesl, Frédéric Mesnard, Albert Rubio, René Thiemann, and Johannes Waldmann. Termination competition (termCOMP 2015). In *CADE-25*, volume 9195 of *LNCS*, pages 105–108, 2015.

7 Aart Middeldorp, Georg Moser, Friedrich Neurauter, Johannes Waldmann, and Harald Zankl. Joint spectral radius theory for automated complexity analysis of rewrite systems. In *CAI 2011*, volume 6742 of *LNCS*, pages 1–20, 2011.

8 Georg Moser, Andreas Schnabl, and Johannes Waldmann. Complexity analysis of term rewriting based on matrix and context dependent interpretations. In *FSTTCS 2008*, volume 2 of *LIPIcs*, pages 304–315, 2008.

9 Friedrich Neurauter, Harald Zankl, and Aart Middeldorp. Revisiting matrix interpretations for polynomial derivational complexity of term rewriting. In *LPAR-17*, volume 6397 of *LNCS*, pages 550–564, 2010.

10 Tobias Nipkow, Lawrence C. Paulson, and Makarius Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

11 Oskar Perron. Zur Theorie der Matrices. *Math. Ann.*, 64:248–263, 1907.

12 René Thiemann and Akihisa Yamada. Formalizing Jordan normal forms in Isabelle/HOL. In *CPP 2016*, pages 88–99. ACM, 2016.

13 René Thiemann and Christian Sternagel. Certification of termination proofs using CeTA. In *TPHOLs'09*, volume 5674 of *LNCS*, pages 452–468, 2009.

T$_T$T$_2$ with Termination Templates for Teaching*

Jonas Schöpf and Christian Sternagel

University of Innsbruck
Innsbruck, Austria
{jonas.schoepf|christian.sternagel}@uibk.ac.at

Abstract

On the one hand, checking specific termination proofs by hand, say using a particular collection of matrix interpretations, can be an arduous and error-prone task. On the other hand, automation of such checks would save time and help to establish correctness of exam solutions, examples in lecture notes etc. To this end, we introduce a template mechanism for the termination tool T$_T$T$_2$ that allows us to restrict parameters of certain termination methods. In the extreme, when all parameters are fixed, such restrictions result in checks for specific proofs.

Keywords and phrases teaching, termination tools, templates, proof checker

1 Introduction

Many of us are familiar with the following two (or at least similar) situations:

> *Enthusiastically we call on our favorite termination tool in order to create an example for a lecture or an exercise for an exam. But is it really necessary that the weights of this KBO are higher than four? And wouldn't it be nicer if the successor symbol was actually interpreted as the successor function in that polynomial termination proof.*
>
> *Hard pressed for time, you have to correct 20 term rewriting exams and each of the students seems to have chosen different matrix interpretations in Example 2. How will you manage before the deadline? Maybe you should just give each student full points.*

As a more concrete example, consider the following well-known term rewrite system (TRS) implementing addition on natural numbers:

$$\mathsf{0} + y \to y \qquad\qquad \mathsf{s}(x) + y \to \mathsf{s}(x + y)$$

If you ask T$_T$T$_2$ [3] whether this TRS is terminating by KBO (`ttt2 -s 'kbo'`), you will get a `YES` with $w_0 = 1$, weights $w(\mathsf{s}) = w(\mathsf{0}) = 1$ and $w(\mathsf{+}) = 0$, and precedence $\mathsf{+} > \mathsf{s} \sim \mathsf{0}$.[1] In a lecture, you might want to restrict to the basic version of KBO with total precedences and moreover it might be nicer for a presentation if all function symbols had the same weight, say 1. Using our templates, this is now possible via

```
ttt2 -s 'kbo -prec "+ > s > 0" -w0 1 -weights "+ = s = 0 = 1"'
```

An instance of the other kind of example we mention above would be the question whether the following matrix interpretations prove termination of the addition TRS (we can verify by `ttt2 -s 'matrix -direct -dim 2'` that there is a similar proof, but not using exactly the same interpretations):

$$[\mathsf{0}] = \begin{pmatrix}0\\0\end{pmatrix} \qquad [\mathsf{s}](x_0) = x_0 + \begin{pmatrix}1\\1\end{pmatrix} \qquad [\mathsf{+}](x_0, x_1) = \begin{pmatrix}1 & 1\\0 & 1\end{pmatrix} x_0 + \begin{pmatrix}1 & 0\\0 & 1\end{pmatrix} x_1 + \begin{pmatrix}1\\0\end{pmatrix}$$

With our template mechanism you can check the given proof by

* This work is supported by the Austrian Science Fund (FWF) project P27502.

[1] Where $\sim$ is "don't care" for strict and equivalence of function symbols for quasi-precedences.

© Jonas Schöpf and Christian Sternagel;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

```
ttt2 -s 'matrix -inters "0 = 0, s = x0 + 1, + = [1,1;0,1]x0 + x1 + [1;0]"'
```

The goal of our work is to successfully handle situations like the above. More specifically, our contributions are as follows:

- Based on a simple idea (Section 2), we devised a template mechanism (Section 3) for four of the most prominent encoding-based termination methods that allows us to employ TTT2 as a "proof checker."
- Moreover, we extended TTT2's web interface (Section 4, where also ☑ is explained): on the one hand, to support our template mechanism and on the other hand, by the possibility to generate URLs that fix input TRSs and custom settings. The latter is especially useful for lectures, where it provides a fast and easy way to demonstrate examples.

2 Main Idea

Many of the termination methods that TTT2 supports are based on SAT/SMT encodings, notably the *lexicographic path order* (LPO), the *Knuth-Bendix order* (KBO), *polynomial interpretations* (PIs), and *matrix interpretations* (MIs).[2]

These methods have the following implementation detail in common: constraints on their parameters (like a precedence for LPO or KBO and the maximal value of matrix entries for MIs) are encoded into a SAT/SMT formula ϕ and then a SAT/SMT-solver is used to obtain a concrete instance. As a consequence such concrete instances of methods seem entirely random to the casual user, which is not always desirable.

For example, we might only be interested in KBO proofs where a certain symbol has least precedence, or we might want to interpret a unary function symbol `s` as successor function without restricting the interpretations of other symbols. In the extreme, we already have a specific proof at hand (say using specific matrix interpretations) and want to use a tool like TTT2 to verify its correctness.

In all of the above cases, an obvious solution is to modify the encoding ϕ in such a way that the desired constraints are fulfilled by every satisfying assignment. This is easily achieved by appending a formula χ that represents these constraints, resulting in $\phi' = \phi \wedge \chi$.

Of course, it would be *really* cumbersome if we had to write χ by hand. Not to mention that we would have to know implementation details concerning the encoding ϕ, for example, which arithmetic variable represents the precedence position of a given function symbol?

Instead we provide a method-specific template mechanism whose main work is to parse and translate constraints that a user provides in form of a human-readable string.

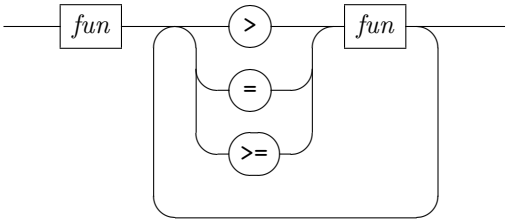
3 Templates for SAT/SMT-Based Methods

In the following we give an overview of the various templates that we provide for the four encoding-based methods from the introduction.

[2] There are others, like *arctic interpretations*, the *(generalized) subterm criterion*, etc. But for the moment templates are only supported by the four methods mentioned above.

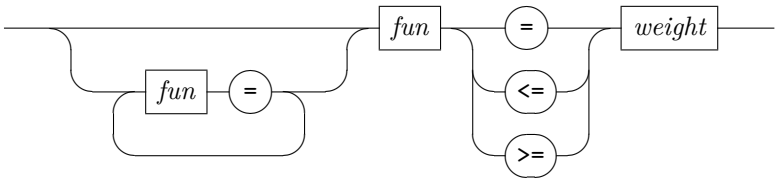
Lexicographic Path Orders The only parameter of an LPO (`ttt2 -s 'lpo'`) is its precedence. It can be specified using the flag `-prec` that takes a *prec* template as argument. A *prec* template represents a (partial) precedence by listing its constituent function symbols in decreasing order (separated by `>`, `=`, or `>=`), according to the syntax diagram on the right-hand side (as soon as `=` or `>=` is used, we implicitly switch from strict to quasi-precedences).

prec



Knuth-Bendix Orders In addition to a precedence, KBO (`ttt2 -s 'kbo'`) is also parameterized by weights, which can be specified using the flags `-w0` (for the weight of variables) and `-weights` that take a single weight and a *weights* template, respectively, as arguments. Weights are natural numbers and can be specified for multiple function symbols at once, as depicted in the following syntax diagram:

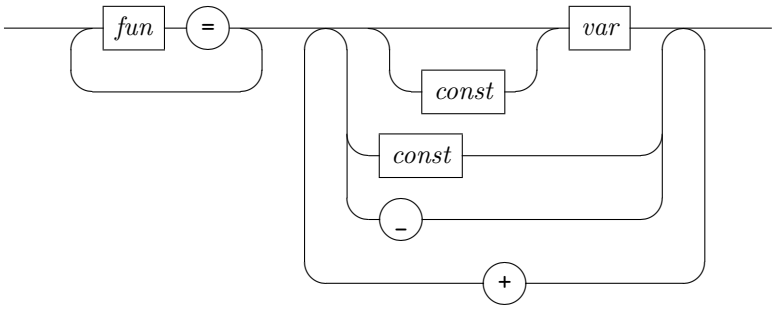
weights



If the last relation symbol of a *weights* template is `<=` or `>=`, then the specified weight is an upper and lower bound, respectively, of all preceding function symbols of the same template.

Linear Interpretations Linear interpretations can be specified using the flag `-inters` (supported by PIs and MIs) that takes an *inters* template as argument. Such interpretations are sums of linear monomials that may either be an optional coefficient followed by a variable, a constant part, or an underscore, according to the syntax diagram:

inters

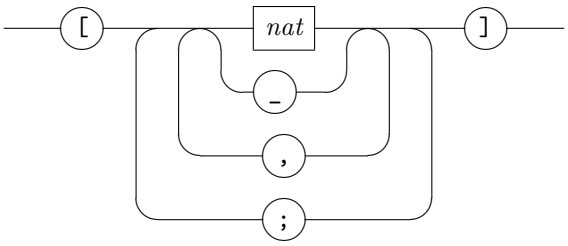


Here, underscores denote arbitrary (unrestricted) parts of an interpretation and variables are associated to function arguments via their index (starting from 0). For example, given a binary function symbol `f`, the template `2x0 + x1` fixes its interpretation to $[\mathtt{f}](x, y) = 2x+y$.

Polynomial Interpretations Polynomial interpretations (`ttt2 -s 'poly'`) are parameterized by linear polynomials as interpretations for function symbols. We obtain them by taking natural numbers for "const" in the *inters* template.

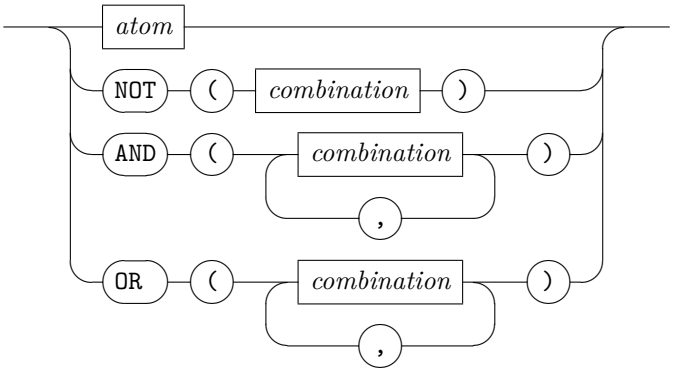
Matrix Interpretations For matrix interpretations (`ttt2 -s 'matrix'`), we instantiate "const" in the *inters* template by matrices of natural numbers as specified by the diagram on the right-hand side.[3] Moreover, we provide the shorthands `0` and `1` for the zero-vector and one-vector, respectively.

matrix



Boolean Combinations of Constraints For all of the above templates (as atoms), we actually support arbitrary boolean combinations of atomic constraints as specified below:

combination



As a common special case a comma-separated list of atoms is allowed at the toplevel, which is then interpreted as logical conjunction of atomic templates.

Examples We conclude this section by giving some example templates:

- Let us first consider an LPO such that `f` is not at the same time bigger than `g` and bigger than `h` in the precedence: `-prec "NOT(AND(f > g, f > h))"` ☑
- Can we find (linear) polynomial interpretations such that the constant part of [+] is 2 and whenever `0` is interpreted as 0, then `s` should also obtain its natural interpretation? `-inters "AND(+ = _ + 2, OR(NOT(0 = 0), s = x0 + 1))"` ☑
- How about matrix interpretations with upper triangular matrices of dimension three that have only ones on their diagonals, for unary `f` and binary `g` and `h`? `-inters "f=g=h=[1,_,_;0,1,_;0,0,1]x0+_, g=h=[1,_,_;0,1,_;0,0,1]x1+_"` ☑

4 Teaching with the Web-Interface

It is the experience of the second author that for teaching, while live demonstrations of tools are often appreciated by students, properly setting up such examples often takes way too much time. To remedy this situation at least for TTT2, we integrated a mechanism into

[3] When the flag `-inters` is present, the matrix dimension is read from the template, which usually means that `-dim` is not required anymore.

its web interface[4] that allows a user to store the current configuration (including the input TRS) into the query part of the web interface URL.

Now, we can easily copy such a URL and turn it, for example, into a PDF hyperlink on slides or in a paper. Incidentally, this is exactly the purpose of the ☑ symbols in this paper. The first one, for example, is generated by the following (incomplete) LaTeX code:

```
\href{http://colo6-c703.uibk.ac.at/ttt2/web/?problem=(VAR\%20x\%20y)...
```

Moreover, our termination templates are accessible through dedicated input fields of the web interface. Thus, a user does not have to type (and know) the corresponding flags.

5 Conclusion and Future Work

We presented the extension of TTT2 by a template mechanism for four of the most common termination techniques that are taught in classes. These templates allow us to narrow down the search space such that in the extreme, we are left with a specific instance of a termination technique. In this way, TTT2 can be used as a "proof checker" for termination proofs (if you want to have near absolute certainty that some termination proof is correct, you should of course validate TTT2's output by a formally verified certifier like CeTA). Furthermore, we demonstrated a simple but useful addition to TTT2's web interface to generate URLs that make the input TRS and current configuration persistent.

We leave templates for methods like *arctic interpretations* [2], the *(generalized) subterm criterion* [4], and *finding loops* [5] as future work. An orthogonal issue that requires further investigation is: which extensions to the current templates would be most useful for teaching?

Some similar options to our termination templates were available through a GUI in an old version of AProVE [1, Section 2]. Unfortunately, this GUI was abandoned in later versions.

References

1 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Automated Termination Proofs with AProVE. In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA)*, volume 3091 of *Lecture Notes in Computer Science*, pages 210–220. Springer, 2004. 10.1007/978-3-540-25979-4_15.

2 Adam Koprowski and Johannes Waldmann. Arctic Termination ... Below Zero. In *Proceedings of the 19th International Conference on Rewriting Techniques and Applications (RTA)*, volume 5117 of *Lecture Notes in Computer Science*, pages 202–216. Springer, 2008. 10.1007/978-3-540-70590-1_14.

3 Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean Termination Tool 2. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA)*, volume 5595 of *Lecture Notes in Computer Science*, pages 295–304. Springer, 2009. 10.1007/978-3-642-02348-4_21.

4 Christian Sternagel. The generalized subterm criterion in TTT2. In *Proceedings of the 15th Workshop on Termination (WST)*, 2016. arXiv:1609.03432.

5 Harald Zankl, Christian Sternagel, Dieter Hofbauer, and Aart Middeldorp. Finding and certifying loops. In *Proceedings of the 36th International Conference on Theory and Practice of Computer Science (SOFSEM)*, volume 5901 of *Lecture Notes in Computer Science*, pages 755–766. Springer, 2010. 10.1007/978-3-642-11266-9_63.

[4] http://colo6-c703.uibk.ac.at/ttt2/web/

Procedure-Modular Termination Analysis

Cristina David[1], Daniel Kroening[2], and Peter Schrammel[3]

1 **Department of Computer Science and Technology, University of Cambridge, United Kingdom**
2 **Department of Computer Science, University of Oxford, United Kingdom**
3 **Department of Informatics, University of Sussex, United Kingdom**

Abstract

Non-termination is the root cause of a variety of program bugs, such as hanging programs and denial-of-service vulnerabilities. This makes an automated analysis that can prove the absence of such bugs highly desirable. To scale termination checks to large systems, an interprocedural termination analysis seems essential. This is a largely unexplored area of research in termination analysis, where most effort has focussed on small but difficult single-procedure problems.

We present a modular termination analysis for C programs using template-based interprocedural summarisation. Our analysis combines a context-sensitive, over-approximating forward analysis with the inference of under-approximating preconditions for termination. Bit-precise termination arguments are synthesised over lexicographic linear ranking function templates. Our experimental results show the advantage of interprocedural reasoning over monolithic analysis in terms of efficiency, while retaining comparable precision.

Keywords and phrases verification, termination, interprocedural analysis

1 Problem Description

Termination bugs compromise safety-critical software systems by making them unresponsive. In particular, termination bugs can be exploited in denial-of-service attacks [1]. Termination guarantees are therefore instrumental for ensuring software reliability. Termination provers are static analysis tools that aim to construct a proof of termination for a given input program and the implementations of these tools have made tremendous progress in the past few years. They compute proofs for complex loops that may require linear lexicographic (e.g., [2, 8]) or non-linear termination arguments in a completely automatic way. However, there remain major practical challenges in analysing the termination of real-world code.

First of all, as observed by [4], most approaches in the literature are specialised to linear arithmetic over unbounded mathematical integers. Even though unbounded arithmetic may reflect the intuitively-expected program behaviour, the program actually executes over bounded machine integers. The semantics of C allows unsigned integers to wrap around when they over/underflow. Hence, arithmetic on k-bit-wide unsigned integers must be performed modulo 2^k. According to the C standards, over/underflows of signed integers are undefined behaviour, but practically also wrap around on most architectures. Thus, accurate termination analysis requires a *bit-precise* analysis of the program semantics (i.e., an analysis that captures the semantics of programs down to each individual bit by using machine arithmetic). Tools must be configurable with architectural specifications such as the width of data types and endianness. The following examples illustrate that termination behaviour on machine integers can be completely different than on mathematical integers.

© Cristina David, Daniel Kroening, Peter Schrammel;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

For example, the program fragment

```
void foo1(unsigned n)
{
  for(unsigned x=0; x<=n; x++);
}
```

does terminate with mathematical integers, but does *not* terminate with machine integers if `n` equals the largest unsigned integer. On the other hand, the program fragment

```
void foo2(unsigned x)
{
  while(x>=10) x++;
}
```

does not terminate with mathematical integers, but terminates with machine integers because unsigned machine integers wrap around.

A second challenge is to make termination analysis scale to larger programs. The annual Software Verification Competition (SV-COMP) [11] includes a division in termination analysis, which reflects a representative picture of the state of the art. The SV-COMP 2016 termination benchmarks contain challenging termination problems on smaller programs with at most 453 instructions (average 53), feature at most seven functions (average three) and four loops (average one).

In this paper, we present a technique that we have successfully run on programs that are one order of magnitude larger, containing up to 5000 instructions. Larger instances require different algorithmic techniques to scale, and we conjecture that the key technique is a modular, interprocedural analysis which decomposes a problem into sub-problems rather than a monolithic analysis which would put all its resources into solving the problem all at once. Modular termination analysis raises several conceptual and practical challenges that do not arise in monolithic termination analysers. For example, when proving termination of a program, a possible approach is to prove that all procedures in the program terminate *universally*, i.e., in any possible calling context. However, this criterion is too optimistic, as termination of individual procedures often depends on the calling context, i.e., procedures terminate *conditionally* only in specific calling contexts.

The approach that we take is verifying universal program termination in a top-down manner by proving termination of each procedure relative to its calling contexts, and propagating upwards which calling contexts guarantee termination of the procedure. It is too difficult to determine these contexts precisely; analysers thus compute preconditions for termination. A *sufficient precondition* identifies pre-states in which the procedure will definitely terminate, and is thus suitable for proving termination. By contrast, a *necessary precondition* identifies pre-states in which the procedure may terminate. Its negation are states in which the procedure will not terminate, which is useful for proving non-termination.

In this paper we focus on interprocedural termination analysis and computation of sufficient preconditions for termination. Preconditions enable information reuse, and thus scalability, as it is frequently possible to avoid repeated analysis of parts of the code base, e.g., libraries whose procedures are called multiple times or parts of the code that did not undergo modifications between successive analysis runs. This paper is a short version of [3].

Algorithm 1: *analyze*

```
1  global Sums^o, Invs^o, Preconds^u;
2  function analyzeForward(f, CallCtx_f^o)
3      foreach procedure call h in f do
4          CallCtx_h^o = compCallCtx^o(f, CallCtx_f^o, h);
5          if needToReAnalyze^o(h, CallCtx_h^o) then
6              analyzeForward(h, CallCtx_h^o);
7      join^o((Sums^o[f], Invs^o[f]), compInvSum^o(f, CallCtx_f^o))
8  function analyzeBackward(f, CallCtx_f^u)
9      termConds = CallCtx_f^u;
10     foreach procedure call h in f do
11         CallCtx_h^u = compCallCtx^u(f, CallCtx_f^u, h);
12         if needToReAnalyze^u(h, CallCtx_h^u) then
13             analyzeBackward(h, CallCtx_h^u);
14         termConds ← termConds ∧ Preconds^u[h];
15     join^u(Preconds^u[f],
               compPrecondTerm(f, Invs^o[f], termConds));
16 function analyze(f_entry)
17     analyzeForward(f_entry, true);
18     analyzeBackward(f_entry, true);
19     return Preconds^u[f_entry];
```

2 Overview of the Approach

We next introduce the architecture of our interprocedural termination analysis. Our analysis combines, in a non-trivial synergistic way, the inference of invariants, summaries, calling contexts, termination arguments, and preconditions, which have a concise characterisation in second-order logic. To see how the different analysis components fit together, we now go through the pseudo-code of our termination analyser (Algorithm 1). We use three global maps $Sums^o$, $Invs^o$, and $Preconds^u$ to store the summaries, invariants and preconditions that we compute for procedures f of the program. Note that superscripts o and u in predicate symbols indicate over- and under-approximation, respectively. Function *analyze* is given the entry procedure f_{entry} of the program as argument and proceeds in two analysis phases.

Phase one is an *over-approximate* forward analysis, given in subroutine *analyzeForward*, which recursively descends into the call graph from the entry point f_{entry}. Subroutine *analyzeForward* infers for each procedure call in f an over-approximating calling context $CallCtx^o$, using procedure summaries and other previously-computed information. Before analyzing a callee, the analysis checks if the callee has already been analysed and whether the stored summary can be re-used, i.e., if it is compatible with the new calling context $CallCtx^o$. Finally, once summaries for all callees are available, the analysis infers loop invariants and a summary for f itself, which are stored for later re-use.

The second phase is an *under-approximate* backward analysis, subroutine *analyzeBackward*, which infers termination preconditions. Again, we recursively descend into the call graph. Analogous to the forward analysis, we infer for each procedure call in f an under-approximating calling context $CallCtx^u$ and recur only if necessary (Line 12). Finally, we compute the under-approximating precondition for termination (Line 15). This precondition is inferred w.r.t. the termination conditions that have been collected: the backward calling context (Line 9), the preconditions for termination of the callees (Line 14), and the termination arguments for f itself. The interested reader can find the descriptions of the subroutines $join^o$, $join^u$, $needToReAnalyze^o$, and $needToReAnalyze^u$ in [3].

Table 1 Tool comparison

	expected	2LS IPTA	2LS MTA	TAN	Ultimate	llvm2kittel+ T2	llvm2kittel+ KITTeL
terminating	264	263	234	18	150	6	178
non-terminating	333	320	328	3	325	135	–
potentially non-terminating	—	14	35	425	0	0	4
timed out	—	0	0	151	118	61	233
out of memory	—	0	0	0	4	111	182
errors	—	0	0	0	0	284	0
total run time (h)	—	0.35	1.88	92.8	63.6	42.9	144.9

3 Results and Discussion

We implemented the approach in 2LS, a static analysis tool for C programs [10]. Interprocedural termination analysis (IPTA) is the approach presented in this article; monolithic termination analysis (MTA) is IPTA applied to the program where all procedures are inlined. We used the *product line* benchmarks of the [11] benchmark repository. In contrast to other categories, this benchmark set contains programs with non-trivial procedural structure. This benchmark set contains 597 programs with 1100 to 5700 lines of code (2705 on average), 33 to 136 procedures (67 on average), and four to ten loops (5.5 on average). Of these benchmarks, 264 terminate universally, whereas 333 never terminate when starting from `main`.

Modular termination analysis is fast We compared IPTA with MTA. Table 1 shows that the total analysis time of IPTA is 5.3 times lower than the time taken by MTA. The geometric mean speed-up of IPTA w.r.t. MTA on those benchmarks for which both approaches return a definitive answer (terminating or non-terminating) is 2.02.

Modular termination analysis is precise We compare IPTA with MTA. Table 1 shows that IPTA proves 99.6 % of the terminating benchmarks, whereas 88.6 % were proven by MTA. MTA can prove 98.5 % of the non-terminating benchmarks including 8 benchmarks that IPTA classifies potentially non-terminating due to imprecision by the use of summaries to handle function calls. Therefore neither approach is strictly better than the other.

2LS is competitive with existing termination analysis tools on procedural programs We tried to compare 2LS with 10 termination tools for C programs. We succeeded to run the following 4 tools, namely TAN [13], Ultimate [5], T2 [12], and KITTeL [6]. For the latter two tools we used the front-end llvm2KITTeL [9] for generating the required input formats. TAN [7] and KITTeL [4] support bit-precise C semantics. Ultimate uses mathematical integer reasoning but tries to ensure conformance with bit-vector semantics. For each of the tools, Table 1 lists the number of instances solved, timed out, run out of memory or aborted because of an internal error. None of the tools reported wrong results w.r.t. the expected outcome (column "expected").

Ultimate proves 56.8 % of the terminating benchmarks correctly, and 97.6 % of the non-terminating ones. T2 proved 2.3 % of the terminating benchmarks and 40.5 % of the non-terminating ones, whereas KITTeL proved 67.4 % of the terminating ones. KITTeL cannot prove non-termination. The comparison with the latter two tools suffered from the deficiencies of the available front-ends. llvm2KITTeL timed out on some benchmarks and was unable to generate syntactically correct T2 files for almost half the benchmarks (error).

This suggests that current termination tools are quite specialised in handling and per-

forming well on different sets of benchmarks, which makes it difficult to compare them in a fair manner.

Our results show that the technique implemented in 2LS is very efficient in analyzing procedural programs. However, one has to be careful in comparing the run times as the tools have different capabilities w.r.t. finding termination and non-termination arguments. In particular, 2LS uses relatively weak abstractions (linear lexicographic ranking functions and very simple polyhedral invariants), which make it very fast, but these abstractions are not expressive enough to provide as complex termination arguments as the other tools are able produce. Implementing more powerful domains is ongoing work.

More detailed results and an evaluation of precondition generation can be found in [3].

References

1 2009. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1890`.

2 Amir M. Ben-Amram and Samir Genaim. Ranking functions for linear-constraint loops. *Journal of the ACM*, 61(4):26:1–26:55, 2014.

3 Hong-Yi Chen, Cristina David, Daniel Kroening, Peter Schrammel, and Björn Wachter. Bit-precise procedure-modular termination analysis. *ACM Trans. Program. Lang. Syst.*, 40(1):1:1–1:38, 2018.

4 Stephan Falke, Deepak Kapur, and Carsten Sinz. Termination analysis of imperative programs using bitvector arithmetic. In *Verified Software: Theories, Tools, Experiments*, volume 7152 of *LNCS*, pages 261–277. Springer, 2012.

5 Matthias Heizmann, Daniel Dietsch, Marius Greitschus, Jan Leike, Betim Musa, Claus Schätzle, and Andreas Podelski. Ultimate automizer with two-track proofs (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9636 of *LNCS*, pages 950–953. Springer, 2016. `http://ultimate.informatik.uni-freiburg.de/` (special build based on version SV-COMP-2016).

6 2016. `https://github.com/s-falke/kittel-koat` (version 6ee36da).

7 Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. Termination analysis with compositional transition invariants. In *Computer-Aided Verification*, volume 6174 of *LNCS*, pages 89–103. Springer, 2010.

8 Jan Leike and Matthias Heizmann. Ranking templates for linear loops. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *LNCS*, pages 172–186. Springer, 2014.

9 2016. `https://github.com/hkhlaaf/llvm2kittel` (version e37be65e).

10 Peter Schrammel and Daniel Kroening. 2LS for program analysis (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9636 of *LNCS*, pages 905–907. Springer, 2016.

11 2016. `https://github.com/sosy-lab/sv-benchmarks/releases/tag/svcomp16`.

12 2016. `https://github.com/mmjb/T2` (version 90c5d0e).

13 2014. `http://www.cprover.org/termination/` (version SV-COMP-2014).

Well-founded models in proofs of termination*

Salvador Lucas

DSIC, Universitat Politècnica de València, Spain
`http://slucas.webs.upv.es/`

Abstract

We prove that operational termination of declarative programs can be characterized by means of well-founded relations between specific formulas which can be obtained from the program. We show how to generate such relations by means of logical models where the interpretation of some binary predicates are required to be well-founded relations. Such logical models can be automatically generated by using existing tools. This provides a basis for the implementation of tools for automatically proving operational termination of declarative programs.

Keywords and phrases Abstraction, Logical models, Operational Termination, Well-foundedness

1 Introduction

The operational semantics of declarative programs, which are viewed as theories $\mathcal{S}$ of a logic $\mathcal{L}$ [13], can be described by means of an inference system $\mathcal{I}(\mathcal{S})$ which is usually obtained from a generic inference system when the particular expressions of the program are considered. The termination behavior of the program is somehow encoded in $\mathcal{I}(\mathcal{S})$. The notion of *operational termination* (OT) characterizes such termination behavior as the absence of infinite well-formed proof trees, which are those that an interpreter would usually build when trying to prove a goal [10]. Such a notion provides a language independent framework for termination analysis.

In [11] we have introduced a *framework* (the *OT Framework*) to prove operational termination. The central notion is that of *proof jump*. A *proof jump* ψ for $\mathcal{I}(\mathcal{S})$ (or just $\mathcal{S}$) is a pair $(A \Uparrow \vec{B}_n)$, where $n \geq 1$ and $A, B_1, \ldots, B_n$ are formulas; A and B_n are called the *head* and the *hook* of ψ, respectively. Given an inference rule $\frac{B_1 \cdots B_n}{A}$ (or $\frac{\vec{B}_n}{A}$ for short) with label ρ and $1 \leq i \leq n$, $[\rho]^i$ denotes the i-th proof jump $A \Uparrow B_1, \ldots, B_i$ which is obtained from ρ. The set of *proof jumps of* $\mathcal{I}(\mathcal{S})$ is $\mathcal{J}_\mathcal{S} = \{(A \Uparrow \vec{B}_i) \mid \frac{\vec{B}_n}{A} \in \mathcal{I}(\mathcal{S}), 1 \leq i \leq n\}$. Proof jumps $(A \Uparrow \vec{B}_n)$ provide a clear distinction between the source of progress of an infinite behavior during a *computation-as-proof* (by means of the head A and hook B_n) and the logical context surrounding such a progress (we can assume $B_1, \ldots, B_{n-1}$ provable, after instantiation). An infinite $(\mathcal{S}, \mathcal{J})$-*chain* is a sequence $(\psi_i)_{i \in \mathbb{N}}$ of proof jumps $\psi_i : (A^i \Uparrow \vec{B}^i_{n_i}) \in \mathcal{J}$ together with a substitution σ such that for all $i \in \mathbb{N}$, $\sigma(B^i_{n_i}) = \sigma(A^{i+1})$ and for all j, $1 \leq j < n_i$, $\mathcal{S} \vdash \sigma(B^i_j)$. We write $\mathcal{S} \vdash \varphi$ for a formula φ if there is a proof of φ using $\mathcal{I}(\mathcal{S})$. A theory $\mathcal{S}$ is *operationally terminating* iff there is no infinite $(\mathcal{S}, \mathcal{J}_\mathcal{S})$-chain [11, Theorem 1].

For instance, consider the Maude [1] program in [2, Sect. 1.1] displayed in Listing 1. This program defines a sort `Pal` as subsort of a sort `List` of sequences of symbols from the predefined sort `Qid` (of Maude identifiers). The inference system $\mathcal{I}$(`PALINDROME`) describing the operational semantics of `PALINDROME` is in Figure 1. Note that in this paper (*schematic*) inference rules $\frac{B_1 \ \cdots \ B_n}{A}$ actually denote *instances* $\frac{\sigma(B_1) \ \cdots \ \sigma(B_n)}{\sigma(A)}$ by a substitution σ. The rules are obtained from the generic inference system in [2, Figure 4]. There are *two* membership

* Partially supported by the EU (FEDER), Spanish MINECO project TIN2015-69175-C4-1-R, and GV project PROMETEOII/2015/013

© Salvador Lucas;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

Listing 1 Palindrome specification in Maude

```
fmod PALINDROME is
  protecting QID .  ***Imports sort Qid (quoted identifiers)
  sorts List Pal .  subsorts Qid < Pal < List .
  op nil : -> Pal .
  op __ : List List -> List [assoc id: nil] .
  var I : Qid .
  var P : Pal .
  mb I P I : Pal .  *** membership axiom
endfm
```

$$(SR)_Q\ \frac{x \to y \quad y : \texttt{Qid}}{x : \texttt{Qid}} \qquad (SR)_P\ \frac{x \to y \quad y : \texttt{Pal}}{x : \texttt{Pal}} \qquad (SR)_L\ \frac{x \to y \quad y : \texttt{List}}{x : \texttt{List}}$$

$$(M1)_{\texttt{Q<P}}\ \frac{x :: \texttt{Qid}}{x :: \texttt{Pal}} \qquad (M1)_{\texttt{P<L}}\ \frac{x :: \texttt{Pal}}{x :: \texttt{List}} \qquad (M1)^c_Q\ \frac{}{c :: \texttt{Qid}}\ \text{for } c \text{ of sort } \texttt{Qid}$$

$$(M1)_{\texttt{Nil}}\ \frac{}{\texttt{nil} :: \texttt{Pal}} \qquad (M1)_{\_\_}\ \frac{x :: \texttt{List} \quad y :: \texttt{List}}{x\ y :: \texttt{List}} \qquad (M1)_{\texttt{mbP}}\ \frac{I :: \texttt{Qid} \quad P :: \texttt{Pal}}{IPI :: \texttt{Pal}}$$

$$(M2)_Q\ \frac{x :: \texttt{Qid}}{x : \texttt{Qid}} \qquad (M2)_P\ \frac{x :: \texttt{Pal}}{x : \texttt{Pal}} \qquad (M2)_L\ \frac{x :: \texttt{List}}{x : \texttt{List}}$$

$$(C)_{\_\_,1}\ \frac{x \to y}{x\ z \to y\ z} \qquad (C)_{\_\_,2}\ \frac{x \to y}{z\ x \to z\ y} \qquad (Rf)_L\ \frac{}{x \to^* x}$$

$$(T)_L\ \frac{x \to y \quad y \to^* z}{x \to^* z}$$

Figure 1 PALINDROME program and inference rules

predicates $\_ : s$ and $\_ :: s$ for each sort s; the first one involves *rewritings* of the considered terms. Actually, predicate symbols $\to$ and $\to^*$ are considered despite the absence of rewriting rules in the program. We refer the reader to [2, Section 3] for further details. Associativity of the sequence operator `__` is not considered here due to some undesirable interactions between associativity and memberships in Maude [1, Section 22.2.8]. This is a minor issue for the purpose of this paper. In order to illustrate the notion of proof jump, consider the inference rule $(SR)_Q$ in Figure 1, for which $[(SR)_Q]^1$ is $x : \texttt{Qid} \Uparrow x \to y$ and $[(SR)_Q]^2$ is $x : \texttt{Qid} \Uparrow x \to y,\ \ y : \texttt{Qid}$. Overall, $\mathcal{J}_{\texttt{PALINDROME}}$ contains 19 proof jumps.

In this paper we investigate the role and use of *well-founded relations* and proof jumps in proofs of operational termination. This has been partially considered in [11, 7]. Here we provide some new results (in particular Theorem 1) and examples of use. In particular, we obtain a mechanized proof of operational termination of PALINDROME.

2 Well-founded relations characterize operational termination

We introduce a characterization of operational termination in terms of well-founded relations. As usual, a relation R on a set A is called *well-founded* if there is no infinite sequence $(a_i)_{i\in\mathbb{N}}$ of elements $a_i \in A$ such that $a_i\ R\ a_{i+1}$ for all $i \in \mathbb{N}$. Our main result is the following.

▶ **Theorem 1.** *A theory $\mathcal{S}$ is* operationally terminating *iff there is a well-founded relation $\sqsupset$ such that, for all $A \Uparrow \vec{B}_n \in \mathcal{J}_{\mathcal{S}}$,*

$$\text{for all substitutions } \sigma, \text{ if } \mathcal{S} \vdash \sigma(B_i) \text{ for all } i,\ 1 \le i < n, \text{ then } \sigma(A) \sqsupset \sigma(B_n) \qquad (1)$$

In order to *use* Theorem 1 to prove operational termination of $\mathcal{S}$, we need to check that, for all proof jumps $A \Uparrow \vec{B}_n \in \mathcal{J}_{\mathcal{S}}$, the statement (1) holds. Although (1) is an "implication",

the *provability statements* $\mathcal{S} \vdash \sigma(B_i)$, and the presence of a symbol $\sqsupset$ which does *not* belong to the language of $\mathcal{S}$, prevents (1) from being a formal sentence. As developed in [7], we use theory transformations to overcome this problem. In the following, we restrict the attention to *Order-Sorted First-Order Logic* (OS-FOL) theories $\mathcal{S}$ over a signature with predicates $\Omega = (S, \leq, \Sigma, \Pi)$, where S is a set of sorts, $\leq$ is an ordering on S, Σ is an S-ranked set of function symbols, and Π is an S-ranked set of predicate symbols [5]. We also assume that $A, B_1, \ldots, B_n$ are *atoms* in all inference rules $\frac{\vec{B}_n}{A} \in \mathcal{I}(\mathcal{S})$. We recast (1) as a *logic formula*

$$(\forall \vec{x})\ B_1 \wedge \cdots \wedge B_{n-1} \Rightarrow A^{\downarrow}\ \pi_{\sqsupset}\ B_n^{\downarrow} \tag{2}$$

of an extended signature with a new sort s_τ, where (i) $\pi_{\sqsupset} : s_\tau s_\tau$ is a new binary (infix) predicate accepting terms of sort s_τ, and (ii) $\_^{\downarrow}$ is a transformation $P(t_1, \ldots, t_n)^{\downarrow} = f_P(t_1, \ldots, t_n)$ from atoms $P(t_1, \ldots, t_n)$ into *terms* $f_P(t_1, \ldots, t_n)$ of sort s_τ, where $f_P : w \to s_\tau$ are new function symbols for each predicate $P : w$. Now, the relation $\sqsupset$ between atoms A, B of the original logic is defined by $A \sqsupset B$ iff $\mathcal{A} \models A^{\downarrow}\ \pi_{\sqsupset}\ B^{\downarrow}$ for an appropriate interpretation $\mathcal{A}$. The relation $\sqsupset$ is well-founded if $\mathcal{A}$ has no empty domain (i.e., $\mathcal{A}_s \neq \varnothing$ for all sorts $s \in S$) and $\pi_{\sqsupset}^{\mathcal{A}}$ is well-founded on $\mathcal{A}_{s_\tau}$. Actually, we can give a sufficient condition for proving operational termination by interpretation. In the following result, $\overline{\mathcal{S}}$ is the theory obtained from $\mathcal{I}(\mathcal{S})$ by interpreting each inference rule $\frac{B_1 \cdots B_n}{A}$ as a sentence $(\forall \vec{x})\ B_1 \wedge \cdots \wedge B_n \Rightarrow A$.

▶ **Theorem 2.** *A theory $\mathcal{S}$ is* operationally terminating *iff there is an interpretation $\mathcal{A}$ with no empty domain such that (i) $\mathcal{A} \models \overline{\mathcal{S}}$, (ii) for all $\psi : A \Uparrow \vec{B}_n \in \mathcal{J}_{\mathcal{S}}$, $\mathcal{A} \models (\forall \vec{x}) B_1 \wedge \cdots \wedge B_{n-1} \Rightarrow A^{\downarrow}\ \pi_{\sqsupset}\ B_n^{\downarrow}$ and (iii) $\pi_{\sqsupset}^{\mathcal{A}}$ is a well-founded relation.*

The interpretations which are necessary to use Theorem 2 can be obtained from model generators like AGES [6] or Mace4 [12]. AGES interprets the sort, function, and predicate symbols of an OS-FOL theory as *parametric* domains, functions and predicates. Some binary predicates can be required to be interpreted by a *well-founded* relation. Sentences in the theory are transformed into *constraints* over the parameters which are then solved by using standard constraint solving methods and tools (based on SAT, SMT, etc.) [8].

▶ **Example 3.** Operational termination of `PALINDROME` can be proved by using Theorem 2 together with AGES. For this purpose a simplification is introduced: we use a sort `S` instead of `Qid` to make some symbols (actually `a` and `b`) *available* for sequence construction. A model $\mathcal{A}$ is obtained where sorts are interpreted as follows: $\mathcal{A}_{\mathtt{S}} = \mathcal{A}_{\mathtt{Pal}} = \mathbb{N} - \{0\}$, $\mathcal{A}_{\mathtt{List}} = \mathbb{N}$, and $\mathcal{A}_{s_\tau} = \mathbb{N} \cup \{-1\}$. Function symbols are as follows: $\mathtt{a}^{\mathcal{A}} = \mathtt{b}^{\mathcal{A}} = 1$, $\mathtt{nil}^{\mathcal{A}} = 1$, $x\ \_\ \_^{\mathcal{A}} y = x + y + 1$, $f^{\mathcal{A}}_{\_:\mathtt{S}}(x) = 3x$, $f^{\mathcal{A}}_{\_:\mathtt{Pal}}(x) = 2x + 1$, $f^{\mathcal{A}}_{\_:\mathtt{List}}(x) = 2x + 2$, $f^{\mathcal{A}}_{\_::\mathtt{S}}(x) = -1$, $f^{\mathcal{A}}_{\_::\mathtt{Pal}}(x) = 2x$, $f^{\mathcal{A}}_{\_::\mathtt{List}}(x) = 2x + 1$, $f^{\mathcal{A}}_{\to}(x, y) = 2x - 1$, and $f^{\mathcal{A}}_{\to^*}(x, y) = 4x + y + 1$. Finally,

$$\begin{array}{lll}
_ : \mathtt{S}^{\mathcal{A}}(x) \Leftrightarrow \mathit{true} & _ : \mathtt{Pal}^{\mathcal{A}}(x) \Leftrightarrow \mathit{true} & _ : \mathtt{List}^{\mathcal{A}}(x) \Leftrightarrow \mathit{true} \\
_ :: \mathtt{S}^{\mathcal{A}}(x) \Leftrightarrow x \geq 1 & _ :: \mathtt{Pal}^{\mathcal{A}}(x) \Leftrightarrow x \geq 1 & _ :: \mathtt{List}^{\mathcal{A}}(x) \Leftrightarrow x \geq 1 \\
x \to^{\mathcal{A}} y \Leftrightarrow x > y & x(\to^*)^{\mathcal{A}} y \Leftrightarrow \mathit{true} & x\ \pi_{\sqsupset}^{\mathcal{A}}\ y \Leftrightarrow x > y
\end{array}$$

Mace4 computes (one-sorted) *finite* models only, but it is very fast. There is no support for well-foundedness, though. However, we can use the fact that a finite relation R on a set A is well-founded iff R is not cyclic, i.e., there is no $a \in A$ such that $a\ R^+\ a$. We instruct Mace4 to obtain a well-founded interpretation for $\pi_{\sqsupset}$ by also adding the following sentences:

$$(\forall x\ y)\ x\ \pi_{\sqsupset}\ y \Rightarrow x\ \pi^+_{\sqsupset}\ y \qquad (\forall x\ y\ z)\ x\ \pi_{\sqsupset}\ y \wedge y\ \pi^+_{\sqsupset}\ z \Rightarrow x\ \pi^+_{\sqsupset}\ z \qquad \neg(\exists x)\ x\ \pi^+_{\sqsupset}\ x$$

where $\pi^+_{\sqsupset} : s_\tau\ s_\tau$ is a new predicate symbol. We could not deal with `PALINDROME` using Mace4 and Theorem 2. In Example 6 below, we advantageously use Mace4. First, we briefly introduce the OT Framework, which improves the efficiency of the previous semantic methods to use well-founded relations in proofs of operational termination.

3 The OT Framework and the Removal Pair Processor

As in the DP Framework for TRSs [4], proofs of operational termination in the OT Framework for General Logics [11] are successively *decomposed* or *simplified* into smaller or simpler *problems* until (hopefully) reaching a *trivial one* which stops the process. An *OT problem* τ in a logic $\mathcal{L}$ is a pair $(\mathcal{S},\mathcal{J})$ where $\mathcal{S}$ is a theory and $\mathcal{J}$ is a set of proof jumps; τ is *finite* if there is no infinite $(\mathcal{S},\mathcal{J})$-chain; τ is called *infinite* if it is *not* finite. The set of all OT problems in $\mathcal{L}$ is $OTP(\mathcal{L})$. The *initial OT problem* τ_I of $\mathcal{S}$ is $(\mathcal{S},\mathcal{J}_\mathcal{S})$; thus, $\mathcal{S}$ is operationally terminating iff τ_I is finite. An *OT processor* $\mathsf{P} : OTP(\mathcal{L}) \to \mathcal{P}(OTP(\mathcal{L})) \cup \{\mathsf{no}\}$ maps an OT problem into either a *set of OT problems* or the answer "no". A processor P is *sound* if for all OT problems τ, if $\mathsf{P}(\tau) \neq \mathsf{no}$ and all OT problems in $\mathsf{P}(\tau)$ are finite, then τ is finite. A processor P is *complete* if for all OT problems τ, if $\mathsf{P}(\tau) = \mathsf{no}$ or $\mathsf{P}(\tau)$ contains an infinite OT problem, then τ is infinite. By repeatedly applying processors, we can construct a tree (called *OT tree*) for an OT problem τ whose nodes are labeled with OT problems or *"yes"* or *"no"*, and whose root is labeled with τ. For every inner node with label τ', there is a processor P satisfying one of the following: (i) $\mathsf{P}(\tau') = \mathsf{no}$ and τ' has just one child that is labeled with *"no"*; (ii) $\mathsf{P}(\tau') = \varnothing$ and τ' has just one child that is labeled with *"yes"*; or (iii) $\mathsf{P}(\tau') \neq \mathsf{no}$, $\mathsf{P}(\tau') \neq \varnothing$, and the children of τ' are labeled with the OT problems in $\mathsf{P}(\tau')$. If all leaves of an OT tree for τ are labeled with *"yes"* and all used processors are *sound*, then τ is *finite*. If there is a leaf labeled with *"no"* and all processors used on the path from the root to this leaf are *complete*, then τ is *infinite* [11, Theorem 3].

A *removal pair* $(\gtrsim, \sqsupset)$, consists of binary relations $\gtrsim$ and $\sqsupset$ on formulas such that $\sqsupset$ is *well-founded* and $\gtrsim \circ \sqsupset \subseteq \sqsupset$ or $\sqsupset \circ \gtrsim \subseteq \sqsupset$. We can remove a proof jump $A \Uparrow \vec{B}_n \in \mathcal{J}$ from an OT problem $(\mathcal{S},\mathcal{J})$ provided that the *hook* B_n is '*smaller*' (w.r.t. $\sqsupset$) than the *head* A.

▶ **Definition 4.** [11] Let $(\mathcal{S},\mathcal{J}) \in OTP(\mathcal{L})$, $\psi : A \Uparrow \vec{B}_n \in \mathcal{J}$, and $(\gtrsim, \sqsupset)$ be a *removal pair*. Then, $\mathsf{P}_{RP}(\mathcal{S},\mathcal{J}) = \{(\mathcal{S}, \mathcal{J} - \{\psi\})\}$ if and only if (i) for all $C \Uparrow \vec{D}_m \in \mathcal{J} - \{\psi\}$ and substitutions σ, if $\mathcal{S} \vdash \sigma(D_i)$ for all $1 \leq i < m$, then $\sigma(C) \gtrsim \sigma(D_m)$ or $\sigma(C) \sqsupset \sigma(D_m)$, and (ii) for all substitutions σ, if $\mathcal{S} \vdash \sigma(B_i)$ *for all* $1 \leq i < n$, then $\sigma(A) \sqsupset \sigma(B_n)$.

P_{RP} is sound and complete [11, Theorem 8]. In [7] we describe a semantic approach to the implementation of P_{RP} when applied to an OT problem $\tau = (\mathcal{S},\mathcal{J})$ as the problem of finding a model $\mathcal{A}$ for a theory $\mathcal{S}_\tau$ which is obtained by extending $\mathcal{S}$ with appropriate sentences.

▶ **Definition 5** (Semantic version of P_{RP} [7])**.** Let $(\mathcal{S},\mathcal{J}) \in OTP(\mathcal{L})$, $\mathcal{A}$ be an interpretation with no empty domain, and $\psi : A \Uparrow \vec{B}_n \in \mathcal{J}$. Then, $\mathsf{P}_{RP}(\mathcal{S},\mathcal{J}) = \{(\mathcal{S}, \mathcal{J} - \{\psi\})\}$ if $\mathcal{A} \models \overline{\mathcal{S}}$, and the following conditions hold: (i) if $\mathcal{J} - \{\psi\} \neq \varnothing$, then

$$\mathcal{A} \models \big((\forall xyz : s_\tau)(x\ \pi_{\gtrsim}\ y \wedge y\ \pi_{\sqsupset}\ z \Rightarrow x\ \pi_{\sqsupset}\ z)\big) \vee \big((\forall xyz : s_\tau)(x\ \pi_{\sqsupset}\ y \wedge y\ \pi_{\gtrsim}\ z \Rightarrow x\ \pi_{\sqsupset}\ z)\big)$$

(ii) for each $C \Uparrow \vec{D}_m \in \mathcal{J} - \{\psi\}$, there is $\pi_{\bowtie} \in \{\pi_{\gtrsim}, \pi_{\sqsupset}\}$ such that $\mathcal{A} \models \bigwedge_{i=1}^{m-1} D_i \Rightarrow C^{\downarrow}\ \pi_{\bowtie}\ D_m^{\downarrow}$, and (iii) $\pi_{\sqsupset}^{\mathcal{A}}$ is well-founded and $\mathcal{A} \models \bigwedge_{i=1}^{n-1} B_i \Rightarrow A^{\downarrow}\ \pi_{\sqsupset}\ B_n^{\downarrow}$

▶ **Example 6.** Consider the CTRS $\mathcal{R} = \{\mathsf{a} \to \mathsf{b}, \mathsf{f}(\mathsf{a}) \to \mathsf{b}, \mathsf{g}(x) \to \mathsf{g}(\mathsf{a}) \Leftarrow \mathsf{f}(x) \to x\}$ in [3, page 46]. Its associated inference system $\mathcal{I}(\mathcal{R})$ is:

$$(Rf)\ \frac{}{x \to^* x} \qquad (T)\ \frac{x \to y \quad y \to^* z}{x \to^* z} \qquad (C)_{\mathsf{f},1}\ \frac{x \to y}{\mathsf{f}(x) \to \mathsf{f}(y)} \qquad (C)_{\mathsf{g},1}\ \frac{x \to y}{\mathsf{g}(x) \to \mathsf{g}(y)}$$

$$(Rl)_1\ \frac{}{\mathsf{a} \to \mathsf{b}} \qquad (Rl)_2\ \frac{}{\mathsf{f}(\mathsf{a}) \to \mathsf{b}} \qquad (Rl)_3\ \frac{\mathsf{f}(x) \to^* x}{\mathsf{g}(x) \to \mathsf{g}(\mathsf{a})}$$

The initial OT problem is $\tau_I = (\mathcal{R},\mathcal{J}_\mathcal{R}) = (\mathcal{R}, \{[(T)]^1, [(T)]^2, [(C)_{\mathsf{f},1}]^1, [(C)_{\mathsf{g},1}]^1, [(Rl)_3]^1\})$. We use Definition 5 to remove $[(T)]^2$ from τ_I. We found a model $\mathcal{A}$ with domain $\mathcal{A} =$

$\{0,1,2,3\}$ with Mace4 (no model was found with AGES!). Function symbols are interpreted as follows: $\mathsf{a}^{\mathcal{A}} = 0$, $\mathsf{b}^{\mathcal{A}} = 1$, $\mathsf{f}^{\mathcal{A}}(x) = (x+2) \mod 4$, $\mathsf{g}^{\mathcal{A}}(x) = x \mod 2$, $f^{\mathcal{A}}_{\to}(x,y) = f^{\mathcal{A}}_{\to *}(x,y) = x \mod 2$. Predicate symbols are interpreted as follows: $\to^{\mathcal{A}} = \{(0,1),(0,3),(2,1),(2,3)\}$, $(\to^*)^{\mathcal{A}} = \{(x,x) \mid x \in \mathcal{A}\} \cup \to^{\mathcal{A}}$, $\pi^{\mathcal{A}}_{\gtrsim} = \{(0,0),(1,1)\}$, and $\pi^{\mathcal{A}}_{\sqsupset} = (\pi^{+}_{\sqsupset})^{\mathcal{A}} = \{(0,1)\}$. Thus, $\mathsf{P}_{RP}(\tau_I) = \{\tau_1\}$ where $\tau_1 = (\overline{\mathcal{R}}, \{[(T)]^1, [(C)_{\mathsf{f},1}]^1, [(C)_{\mathsf{g},1}]^1, [(Rl)_3]^1\})$. With successive applications of P_{RP} (using Definition 5 with AGES), we remove $[(T)]^1$, then $[(Rl)_3]^1$, followed by $[(C)_{\mathsf{f},1}]^1$, and finally $[(C)_{\mathsf{g},1}]^1$ to finish the proof of operational termination of $\mathcal{R}$.

Note that the conditional rule $\mathsf{g}(x) \to \mathsf{g}(\mathsf{a}) \Leftarrow \mathsf{f}(x) \to x$ in Example 6 is *infeasible*, i.e., no substitution σ satisfies $\sigma(\mathsf{f}(x)) \to^*_{\mathcal{R}} \sigma(x)$. This is proved by using [9, Theorem 6] and the *same* interpretation in Example 6. Thus, such a rule *cannot be used* to perform any rewriting step. However, infeasible rules *cannot* be removed (to 'simplify' the operational termination proof) without changing relevant properties of a CTRS, in particular operational termination, see [9, footnote 1]. Indeed, considering the proof jumps in $\mathcal{J}_{\mathcal{R}}$ to *model* operational termination is essential for the interpretation in Example 6 to prove operational termination of $\mathcal{R}$.

4 Conclusions

We have characterized operational termination of a theory $\mathcal{S}$ by the existence of a well-founded relation on formulas which introduces a *decrease* from (any instance of) the head to the hook of the proof jumps associated to $\mathcal{S}$. We have shown how to use this result by means of *models* of $\overline{\mathcal{S}}$ which satisfy some additional conditions. The use of tools for the automatic generation of (possibly finite) models like AGES and Mace4 permits the automation of the proofs.

References

1 M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. All About Maude – A High-Performance Logical Framework. LNCS 4350, 2007.

2 F. Durán, S. Lucas, C. Marché, J. Meseguer, X. Urbain, Proving Operational Termination of Membership Equational Programs, *Higher-Order and Symb. Comp.* 21(1-2):59–88, 2008.

3 J. Giesl and T. Arts. Verification of Erlang Processes by Dependency Pairs. *Applicable Algebra in Engineering, Communication and Computing* 12:39–72, 2001.

4 J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and Improving Dependency Pairs. *Journal of Automatic Reasoning* 37(3):155–203, 2006.

5 J. Goguen and J. Meseguer. Models and Equality for Logical Programming. In *Proc. of TAPSOFT'87*, LNCS 250:1–22, 1987.

6 R. Gutiérrez, S. Lucas, and P. Reinoso. A tool for the automatic generation of logical models of order-sorted first-order theories. In *Proc. of PROLE'16*, pages 215–230, 2016.

7 S. Lucas. Use Of Logical Models For Proving Operational Termination In General Logics. In *Selected papers from WRLA'16*, LNCS 9942:1–21, 2016.

8 S. Lucas and R. Gutiérrez. Automatic Synthesis of Logical Models for Order-Sorted First-Order Theories. *Journal of Automated Reasoning* 60(4):465–501, 2018.

9 S. Lucas and R. Gutiérrez. Use of logical models for proving infeasibility in term rewriting. *Information Processing Letters*, 136C:90-95, 2018.

10 S. Lucas, C. Marché, and J. Meseguer. Operational termination of conditional term rewriting systems. *Information Processing Letters* 95:446–453, 2005.

11 S. Lucas and J. Meseguer. Proving Operational Termination Of Declarative Programs In General Logics. In *Proc. of PPDP'14*, pages 111–122, ACM Digital Library, 2014.

12 W. McCune Prover9 & Mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

13 J. Meseguer. General Logics. In *Logic Colloquium'87*, pages 275-329, 1989.

Verification of Rewriting-based Query Optimizers

Krishnamurthy Balaji[1], Piyush Gupta[1], and Aalok Thakkar[1,2]

1 **Adobe Systems, Noida, India**
kbalaji@adobe.com, piygupta@adobe.com
2 **Chennai Mathematical Institute, Chennai, India**
aalok@cmi.ac.in

Abstract

We report on our ongoing work on automated verification of rewriting-based query optimizers. Rewriting-based query optimizers are a widely adapted in relational database architecture however, designing these rewrite systems remains a challenge. In this paper, we discuss automated termination analysis of optimizers where rewrite-rules are expressed in HoTTSQL. We discuss how it is not sufficient to reason about rule specific (local) properties such as semantic equivalence, and it is necessary to work with set-of-rules specific (global) properties such as termination and loop-freeness to prove correctness of the optimizer. We put forward a way to translate the rules in HoTTSQL to Term Rewriting Systems, opening avenues for the use of termination tools in the analysis of rewriting-based transformations of HoTTSQL database queries.

Keywords and phrases Rewriting, Termination, Verification, Query Optimization

1 Introduction

Relational query languages provide a high-level declarative interface to access data stored in relational databases. Before an execution engine implements a set of physical operators to evaluate the query, heuristics are used to optimize the evaluation with respect to parameters such as execution time, monetary fees in a cloud scenario, and allocated and accessed memory [6], [13].

Since 1990s, a standard technique in query optimization is to use a set of rewrite rules which optimize the objective by transforming the input query to a normal form by repeated applications of the rules until a fixed point is reached [14]. Although this technique has been around for long, there has been only little progress towards proving its correctness. The property of semantics preservation under rewriting has been studied and [3] puts forward a method to verify this property for rules expressed in HoTTSQL, a SQL like query language. While semantics preservation is a necessary property of a query optimizer [12], there are other important properties such as termination or loop-freeness which may be desired or necessary depending on application and implementation.

Modern query optimizers, such as Catalyst for SparkSQL, not only support user-defined types but also allow users to add their own optimization rewrite rules [1], creating a need for guarantees that the mentioned properties are invariant of the new additions. Our paper sheds light on the necessary and desired specifications for such query optimizers, and put forward a way to reason about two such properties – termination and loop-freeness. In Sect. 2 we illustrate the problem with an example and discuss the optimizer properties, and in Sect. 3 we discuss translating the HoTTSQL syntax to a syntax which allows automated termination analysis. In Sect. 4 we discuss a weaker specification than termination and in Sect. 5 we conclude with a discussion of our contribution and its limitations.

© Krishnamurthy Balaji, Piyush Gupta, and Aalok Thakkar;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

2 Optimizer Properties

Although the execution of an optimizer consists of the application of individual query rewriting rules, not all desired properties of the system can be verified locally (by looking at the rules separately); in many cases, it is necessary to look at rules in context of other rules. One such property is rewriting termination.

▶ **Example 1.** Consider the Selection Push Down rule which moves a selection (filter) directly after the scan of the input table to reduce the amount of data in the execution pipeline as early as possible [4].

```
SELECT * FROM R WHERE a AND b →
SELECT * FROM (SELECT * FROM R WHERE a) WHERE b
```

On other hand, if (`SELECT * FROM R WHERE a`) returns a significant fraction of `R`, the following rule may be added by the user:

```
SELECT * FROM (SELECT * FROM R WHERE a) WHERE b →
SELECT * FROM R WHERE a AND b
```

Though both (Selection Push Down, and the user defined rule) are semantics preserving, having the two rules together introduces a loop in the transition graph, which makes the rewriting process non-terminating.

In order to avoid such conflicts, a user must look into the predefined rules and undertake their tedious analysis in order to check if a given new rule can be added. In this case, one looks at termination. Another property that is often desirable is confluence. In a terminating rewrite system, local confluence is equivalent to confluence, and hence one can check for local confluence to conclude existence and uniqueness of normal forms [2] [11]. The following three properties can be identified as desired specifications for a rewriting-based query optimizer that ensures a unique normal form:

- **Termination:** For all queries Q, any sequence of queries obtained by rewriting starting with Q should terminate.
- **Local Confluence:** For all queries Q, if there are rewrite rules r_1 and r_2 such that $Q \xrightarrow{r_1} Q_1$ and $Q \xrightarrow{r_2} Q_2$, then there exists a query Q' such that $Q_1 \xrightarrow{\hat{\rho}_1} Q'$ and $Q_2 \xrightarrow{\hat{\rho}_2} Q'$ where $\hat{\rho}_1$ and $\hat{\rho}_2$ are sequences of rewrite rules.
- **Semantics Preservation:** If $Q \xrightarrow{r} Q'$ then the result obtained by executing Q on any database D should be the same as the result obtained by executing Q' on D.

3 Reasoning about Termination

HoTTSQL is an SQL-like language that was proposed for checking semantics preservation [3]. The following is a summary of HoTTSQL syntax:

$$
\begin{aligned}
q \in \text{ Query } &::= \mathsf{TABLE} \mid \mathsf{SELECT}\ p\ q \mid \mathsf{FROM}\ q_1, \ldots, q_n \mid q\ \mathsf{WHERE}\ p \\
&\mid q_1\ \mathsf{UNION\ ALL}\ q_2 \mid q_1\ \mathsf{EXCEPT}\ q_2 \mid \mathsf{DISTINCT}\ q \\
b \in \text{ Predicate } &::= e_1 = e_2 \mid \mathsf{NOT}\ b \mid b_1\ \mathsf{AND}\ b_2 \mid b_1\ \mathsf{OR}\ b_2 \mid \mathsf{true} \mid \mathsf{false} \\
e \in \text{ Expression } &::= P2E\ p \mid \mathsf{f}(e_1, \ldots, e_n) \mid \mathsf{agg}(q) \mid \mathsf{CASTEXPR}\ p\ e \\
p \in \text{ Projection } &::= * \mid \mathsf{Left} \mid \mathsf{Right} \mid \mathsf{Empty} \mid p_1.p_2 \mid p_1, p_2 \mid E2P\ e
\end{aligned}
$$

We first translate queries from HoTTSQL to Term Rewriting Systems (TRS) [7] as used in inputs in Termination Problems Data Base (TPDB) so that automated tools for termination analysis such as Tyrolean Termination Tool 2 (TTT2) [8], Automated Program Verification Environment (AProVE) [5], and others. We introduce the following functions:

$$
\begin{aligned}
\text{Query Functions } &: \mathsf{select(p,q)} \mid \mathsf{from1(q)} \mid \mathsf{from2(q1,q2)} \mid \mathsf{where(q,p)} \\
&\mid \mathsf{unionall(q1,q2)} \mid \mathsf{except(q1,q2)} \mid \mathsf{distinct(q)} \\
\text{Predicate Functions } &: \mathsf{eq(e1,e2)} \mid \mathsf{not(b)} \mid \mathsf{and(b1,b2)} \mid \mathsf{or(b1,b2)} \\
\text{Expression Functions } &: \mathsf{p2e(p)} \mid \mathsf{agg(q)} \mid \mathsf{castexpr(p,e)} \\
\text{Projection Functions } &: \mathsf{projdot(p1,p2)} \mid \mathsf{projcom(p1,p2)} \mid \mathsf{e2p(e)}
\end{aligned}
$$

We consider the following as constants:

$$\mathsf{true},\ \mathsf{false},\ \mathsf{star},\ \mathsf{left},\ \mathsf{right},\ \mathsf{empty}$$

Note that we use from1 and from2 to encode FROM of different arities. Higher arities of FROM are encoded using composition of from2. The functions $\mathsf{f}(e_1, \ldots, e_n)$ used to form expressions may be encoded as the functions themselves on a case-to-case basis. Each instance of a TABLE is encoded as a separate variable. The translation from HoTTSQL syntax to the proposed functional syntax is canonical and can be implemented in linear time using a stack.

▶ **Example 2.** We shall encode the Selection Push Down rule from Example 1 in the functional syntax. r, a, and b are variables. Then we have:

select(star,where(from1(r),and(a,b))) →
select(star,where(select(star,where(from1(r),a)),b))

This syntax is designed to be compatible with Tyrolean Termination Tool 2, and so if the rules are once converted from the HoTTSQL syntax to TRS syntax, it can be given as an input to the tool to automatically reason about termination.

4 Weaker Specifications

A major challenge of rewriting systems is that the rewriting process may be of a high computational complexity [10] [15], and we see the same translate to runtime complexity of rewriting based optimizers. In that scenario, the reduction in the runtime for query evaluation may be compensated by the increase in the runtime for query rewriting. A standard ad-hoc method is to introduce a threshold on the number of rewrites [1]. For a rewriting based optimizer that implements this heuristic, termination is not a necessary

requirement[1]. However, the presence of loops can lead to redundant rewritings. We can adapt the results in [9] to check for loop-freeness of by using existing tools for the automatic generation of first-order models.

5 Conclusion

We have put forward a discussion of verification of rewriting-based query optimizers, particularly the need of looking at set-of-rules specific (global) properties, particularly termination and loop-freeness. We have also proposed a translation from HoTTSQL to the Term Rewriting Systems syntax.

A major limitation is the expressiveness of HoTTSQL. A number of rules which are used in practice such as the Predicate Pushdown Rule need pattern matching for their implementation, however, the HoTTSQL syntax does not offer it. We put forward a *wish list* of the types of query-rewrite rules we would like to add to classic term rewriting translation that is proposed in Section 3 in order to encode the rules of popular query optimizers like Catalyst. The following are the three possible extensions:

1. Classic Term Rewriting + Variable Matching: Rules such as predicate pushdown can be encoded by matching bound and free variables according to the schema
2. Classic Term Rewriting + Constrained Rewriting: Rules with integral constraints can be encoded using constrained rewriting
3. Classic Term Rewriting + Pattern Matching: All query-rewrite rules can be encoded by general pattern matching

The future work branches in three directions. Firstly, we would like to work towards developing developer tools which help design rewrite rules keeping the mentioned specifications in perspective. Secondly, we would like to extend the scope of HoTTSQL and our syntax as mentioned in the previous paragraph so we may be able to reason about query optimizers in practice. Finally, we would like to understand other desired and necessary properties of the optimizers, particularly confluence and develop a fuller theory for verified rewriting-based query optimizers.

References

1 Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

2 Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, New York, NY, USA, 1998.

3 Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. Hottsql: Proving query rewrites with univalent sql semantics. *SIGPLAN Not.*, 52(6):510–524, June 2017.

4 Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.).* Pearson Education, 2009.

[1] Though it may be necessary if one wishes to use it to reason about confluence and existence and uniqueness of normalized form

5 Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Analyzing program termination and complexity automatically with aprove. *Journal of Automated Reasoning*, 58(1):3–31, Jan 2017.

6 Matthias Jarke and Jürgen Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, June 1984.

7 J. W. Klop. Handbook of logic in computer science (vol. 2). chapter Term Rewriting Systems, pages 1–116. Oxford University Press, Inc., New York, NY, USA, 1992.

8 Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean termination tool 2. In Ralf Treinen, editor, *Rewriting Techniques and Applications*, pages 295–304, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

9 Salvador Lucas. A semantic approach to the analysis of rewriting-based systems. *CoRR*, abs/1709.05095, 2017.

10 Georg Moser, Andreas Schnabl, and Johannes Waldmann. Complexity Analysis of Term Rewriting Based on Matrix and Context Dependent Interpretations. In Ramesh Hariharan, Madhavan Mukund, and V Vinay, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 304–315, Dagstuhl, Germany, 2008. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

11 M. H. A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.

12 S. T. Shenoy and Z. M. Ozsoyoglu. Design and implementation of a semantic query optimizer. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):344–361, Sep 1989.

13 Immanuel Trummer and Christoph Koch. Approximation schemes for many-objective query optimization. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1299–1310, New York, NY, USA, 2014. ACM.

14 Sieger van Denneheuvel, Karen Kwast, Gerard R. Renardel de Lavalette, and Edith Spaan. Query optimization using rewrite rules. In Ronald V. Book, editor, *Rewriting Techniques and Applications*, pages 252–263, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

15 Chihping Wang and Ming-Syan Chen. On the complexity of distributed query optimization. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):650–662, Aug 1996.

Control-Flow Refinement via Partial Evaluation

Jesús J. Doménech[1], Samir Genaim[2], and John P. Gallagher[3]

1 **Universidad Complutense de Madrid, Spain**
`jdomenec@ucm.es`
2 **Universidad Complutense de Madrid, Spain**
`sgenaim@ucm.es`
3 **Roskilde University, Denmark and IMDEA Software Institute, Spain**
`jpg@ruc.dk`

1 Introduction

Control-flow refinement is a technique used in program analysis to make the implicit control-flow of a given program explicit. Typically, this is done to increase the precision of the corresponding analysis. Consider for example the program on the left:

```
while ( x > 0 )
  if ( y < z ) y++; else x--;
```

```
while (x > 0 && y < z) y++;
while (x > 0 && y >= z) x--;
```

Its execution has two implicit phases: in the first one, variable y is incremented until it reaches the value of z, and in the second phase the value of x is decremented until it reaches the value 0. Control-flow refinement techniques can transform this program into the one on the right, in which the two phases are explicit.

In the context of termination analysis, such a transformation simplifies the termination proof; while the original program requires a lexicographic termination argument, the transformed one requires only linear ones. In addition, in the context of resource usage (cost) analysis, tools that are based on bounding loop iterations using linear ranking functions fail to infer the cost of the first program, while they succeed in inferring a linear upper-bound on the cost of the second one. In general, splitting the control-flow might also help in inferring more precise invariants, without the need for expensive disjunctive abstract domains, and thus improve any analysis that relies on such invariants (e.g. termination and cost analyses).

In the context of cost (and implicitly termination) analysis, control-flow refinement has been studied in [6] and [8]. The first [6] considers a general form of recurrence relations called cost equations, and the latter [8] considers structured imperative programs. Both handle programs with integer variables. These works demonstrated that control-flow refinement is crucial for handling programs that were considered challenging by the cost analysis community.

We started our research with the obvious observation that control-flow refinement is based on a special kind of partial evaluation tailored for a very particular analysis. We decided to explore what we would get, in terms of precision of cost and termination analysis, if instead, we use an existing off-the-shelf partial evaluation tool. This would allow integrating control-flow refinement into existing static analysis tools without any effort.

In this extended abstract, we describe preliminary experimental results obtained by using an off-the-shelf partial evaluation tool for control-flow refinement, and its impact on the precision of termination and cost analysis. Our experiments are done for Integer Transition Systems, which are graphs where edges are annotated with linear constraints describing transition relations between corresponding nodes.

© Jesús J. Doménech, Samir Genaim, and John P. Gallagher;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

2 Control-flow Refinement via Partial Evaluation for Cost and Termination Analyses

In this section we describe our experiments on the use of a partial evaluation tool for control-flow refinement, and its impact on the precision of termination and cost analysis.

Programs. Our programs are Integer Transition Systems. Briefly, such programs are described by control-flow graphs (`CFG`s) where edges are annotated with linear constraints over primed and unprimed integer variables. The unprimed variables represent the current state and the primed variables the next state; for example $\{x > 0, x' = x + 1\}$ describes the transition relation in which x must be positive to apply it, and the value of x increases by 1. We use this form since it is used as an intermediate representation in many termination and cost analyses.

Used Tools. For termination analysis we use `iRankFinder`[1]. It is a termination analysis tool based on using linear and (several kinds of) lexicographic-linear ranking functions. It also infers supporting invariants using the abstract domain of polyhedra. It does not use any control flow-refinement technique.

For cost analysis we use several tools: (1) `KoAT` [4], which is designed to analyze `CFG`s but does not use any control-flow refinement; (2) `PUBs` [2], which is the back-end of the `COSTA` [3] and `SACO` [1] systems. It accepts cost relations as input (a form of recurrence relation). We translate `CFG`s to this form when possible. (3) `CoFloCo`, which similarly to `PUBs` uses cost relations as input but it also applies control-flow refinement. We do not use or compare to the techniques of [8] since the corresponding tools are not available.

For partial evaluation [9], we use a partial evaluation tool (`PE`) for constrained Horn clauses (CHCs) [10, 7] since `CFG`s can be seen as linear CHC programs. It yields a *polyvariant* specialization, that is, it generates a finite number of versions of each predicate (representing a program point), distinguished according to the constraints that hold upon reaching that point. Polyvariance is essential in order to achieve control-flow refinement.

In what follows, we first describe some selected interesting examples, and then describe and discuss our experiments on a larger set of examples.

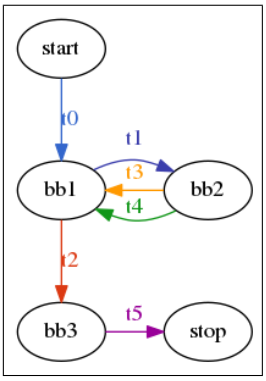
2.1 Selected Examples

Iterative McCarthy 91

Our first example is the iterative McCarthy program depicted in Fig. 1. For an input $x > 100$ the loop body executes the *then* branch once and exits the loop; and for an input $x \leq 100$ the execution is done in two phases: in the first phase the *else* branch is executed repeatedly until the value of x reaches the interval $x \in [101..111]$; and in the second phase the execution alternates between the *then* and *else* branches until c reaches 0. Note that it is guaranteed that c reaches 0 because executing the *then* and *else* branches consecutively increment x by 1 but do not change the value of c, and when reaching a state in which x is 111 the *then* branch is executed twice and thus the value of c is decreased by 1 (and x goes back to 91). The complexity of this program is $O(|x|)$; this is because for $x < 100$ the first phase executes at most $\frac{100-x}{11} + 1$ times, and the second phase executes at most $21(\frac{100-x}{10} + 1) + 1$ (because the value of c when entering the second phase is at most $\frac{100-x}{11} + 2$ and in the second phase it takes at most 21 iterations to decrease c by 1).

[1] `http://irankfinder.loopkiller.com`

```
int McCarthy(int x){
    int c = 1;
    while (c > 0) {
        if (x > 100) {
            x -= 10;
            c--;
        } else {
            x += 11;
            c++;
        }
    }
    return x;
}
```



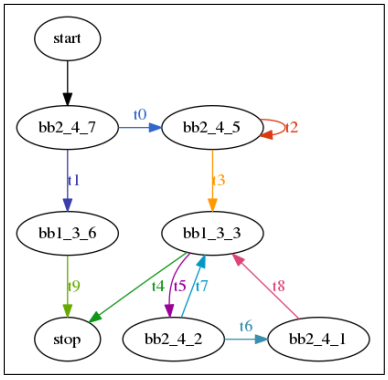


Figure 1 Iterative McCarthy 91

The CFG depicted in Fig. 1 (middle) corresponds to the iterative McCarthy program. It was obtained automatically using `llvm2KITTeL`[2]. Transition `t1` $= \{c > 0, c' = c, x' = x\}$ corresponds to entering the loop body, and transitions `t3` $= \{x > 100, x' = x - 10, c' = c - 1\}$ and `t4` $= \{x \leq 100, x' = x + 11, c' = c + 1\}$ to the *then* and *else* branches, respectively. Analyzing the termination behavior of this CFG using `iRankFinder` results in a termination proof with a lexicographical termination witness $\langle 10c - x + 90, x\rangle$ for the recursive SCC that is defined by the nodes `bb1` and `bb2`. Cost analysis using `KoAT` results in the bound $O(x^2)$, which is not precise enough. `CoFloCo` and `PUBs` were not able to infer any bound for this example. Recall that `CoFloCo` applies some control-flow refinement as well.

Applying `PE` on this CFG results in the CFG depicted Fig. 1 (right) – transition names in the two CFGs are not related. `PE` splits the control into two cases: one in which the input satisfies $x > 100$, which is represented by `t1` and `t9`, and one in which the input satisfies $x \leq 100$ which is represented by the rest of transitions. The subgraph that corresponds to the second case has 2 SCCs. The first one (node `bb2_4_5`) corresponds to the first phase in which the *else* branch is executed repetitively, and the second SCC (nodes `bb1_3_3`, `bb2_4_2` and `bb2_4_1`) corresponds to the second phase in which the *then* and *else* branches alternate.

For this CFG, `iRankFinder` infers that first SCC terminates with a linear termination witness $100 - x$, and that the second terminates with a linear termination witness $335c - 6x$. Having linear ranking functions makes cost analysis simpler as well, and indeed applying `KoAT` on this CFG we infer the bound $O(x)$. Note that `PUBs` would infer a linear bound for this CFG as well since all SCCs are ranked by linear ranking functions.

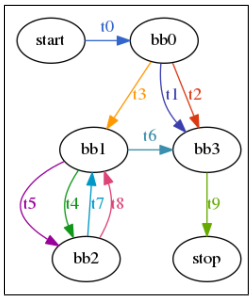
Greatest Common Divisor

Consider the GCD program depicted in Fig. 2. It calculates the GCD of two positive integers using iterative subtracting. This program terminates and has a linear runtime complexity.

The CFG depicted in Fig. 2 (middle) is automatically obtained using `llvm2KITTeL`. The following are transitions that we will discuss later: `t3` $= \{x > 0, y > 0, x' = x, y' = y\}$, `t4` $= \{x > y, x' = x, y' = y\}$, `t5` $= \{x < y, x' = x, y' = y\}$, `t7` $= \{x < y, y' = y - x, x' = x\}$, and `t8` $= \{x \geq y, y' = y, x' = x - y\}$. Note that in `t8`, which corresponds to the *else* branch,

[2] `https://github.com/s-falke/llvm2kittel`

```
int gcd(int x, int y){
 assert(x>0 && y>0);
 while(x != y){
   if(x<y)
     y = y-x;
   else
     x = x-y;
 }
 return x;
}
```



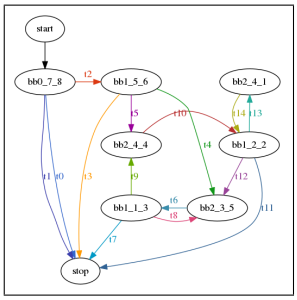


Figure 2 Greatest Common Divisor

we have the constraint $x \geq y$ while at runtime it will always be the case that $x > y$. This happens because `llvm2KITTeL` translates the if-statement independently from the context (the while-condition), and thus for the *else* branch it takes the negation of the if-condition.

`iRankFinder` fails to prove termination of this `CFG`. The reason is that proving termination requires the invariant $\{x \geq 1, y \geq 1\}$ for node $bb2$ in order to rank transitions `t7` and `t8`. `iRankFinder` fails to infer this invariant mainly due to the constraint $x \geq y$, which at some point in the fixpoint computation introduces $x \geq 0$ at node $bb1$, and then the widening operation loses the lower bound of x (a more clever widening would have solved the problem). In order to solve this problem it is enough to unfold transitions in the recursive SCC, and thus collapse the two nodes into one (t_4 followed by t_7, and t_5 followed by t_8) which then will eliminate $x \geq y$. Note that unfolding is the basic operation in partial evaluation. `KoAT` and `PUBs` fail to infer a bound for this program, while `CoFloCo` infers the expected linear bound. Recall that `CoFloCo` applies control-flow refinement.

Applying `PE` to this `CFG` results in the one in Fig. 2 on the right. We can see that `PE` actually did not collapse the nodes of the recursive SCC into one, which would have solved the problem. It actually splits the two phases of the loop into separated ones and introduced transitions for moving from one to another – nodes `bb1_2_2` and `bb2_4_1` correspond to the *else* branch and nodes `bb1_1_3` and `bb2_3_5` correspond to the *then* branch. In addition, it has merged transitions so that the constraint $x \geq y$ disappeared.

Applying `iRankFinder` of this `CFG` infers a linear ranking function for all components. This is possible since now it infers the invariant $\{x \geq 1, y \geq 1\}$ where needed. Applying `KoAT` on this program we get the desired bound $O(|x| + |y|)$. `PUBs` would also get a linear bound.

2.2 Preliminary Experimental Evaluation

We measured the precision of applying `KoAT` and `iRankFinder` on two sets of examples, before and after partial evaluation. We preferred to use `KoAT` for cost analysis since it accepts `CFG`s as input without any need for preprocessing. We used two sets of integer transition systems: `SET-A` (416 benchmarks taken from the termination competition database) and `SET-B` (a set of 188 benchmarks used in [5] to evaluate the precision of `CoFloCo`).

The results of applying `iRankFinder` on set `SET-A` (resp. `SET-B`) are: for 12 (resp. 8) `CFG`s using `PE` improves the result from *unknown* to *terminating*; for 24 (resp. 13) `CFG`s using `PE` improves the result from *lexicographic* to *linear* termination witness; for 9 (resp. 1) `CFG`s the analysis times out when analyzing the `CFG` generated by `PE`, since it was very large, while the analysis of the original `CFG` provides a proof of termination; for 5 (resp. 2) `CFG`s using `PE`

results in a *lexicographic* termination witness while the original in a *linear* one; and for the the rest of CFGs the results are equivalent.

The results of applying KoAT on set SET-A (resp. SET-B) are: for 4 (resp. 10) CFGs using PE improves from *unknown* to some upper-bound; for 0 (resp. 10) CFGs using PE improves the the complexity class of the upper-bound; for 2 (resp. 3) CFGs the analysis times out when analyzing the CFG generated by PE, since it was very large, while the analysis of the original CFG provides an upper-bound; for 1 (resp. 3) CFGs using PE results in an upper-bound of a worse complexity class; and for the the rest of CFGs the results are equivalent.

In summary, in many cases using PE improves the results of both termination and cost analyses. However, in some CFGs it generates large CFGs that the analysis is not able to handle, and in some other cases they lead to worse results. We still need to explore these (large) examples in details in order to understand the reason.

3 Conclusions and Future Work

In this extended abstract we explored the use of PE as a control-flow refinement technique in the context of termination and cost analysis. Our preliminary experiments show that PE can improve both analyses, however, there are several cases where the results are worse that we are currently investigating to identify the source for this imprecision.

Apart from this, for future work we plan to follow up these preliminary research results. In particular, we intend to pursue the following directions. (1) To measure the effect of using PE on performance, in particular, explore the possibility of applying it only on parts of the CFG where the analyzer fails; (2) to measure the effect of using PE on other termination and cost analysis tools (apart from KoAT and iRankFinder); (3) to explore the use of PE for inferring lower-bounds; (4) to specialize the PE tool we use in order to take into account that is applied in the context of termination and cost analysis.

References

1 E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *Proc. of TACAS'14*, volume 8413 of *LNCS*, pages 562–567. Springer, 2014.

2 E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

3 E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Proc. of FMCO'07*, volume 5382 of *LNCS*, pages 113–132. Springer, 2008.

4 M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Analyzing runtime and size complexity of integer programs. *ACM Trans. Program. Lang. Syst.*, 38(4):13:1–13:50, 2016.

5 A. Flores-Montoya. *Cost Analysis of Programs Based on the Refinement of Cost Relations*. PhD thesis, Technische Universität Darmstadt, Darmstadt, August 2017.

6 A Flores-Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In *Proc. of APLAS'14*, volume 8858 of *LNCS*, pages 275–295. Springer, 2014.

7 J. P. Gallagher. Tutorial on specialisation of logic programs. In *Proc. of the ACM SIGPLAN Symposium on PEPM'93*, pages 88–98, 1993.

8 S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *ACM Sigplan Notices*, volume 44, pages 375–385. ACM, 2009.

9 N. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Software Generation*. Prentice Hall, 1993.

10 B. Kafle, J. P. Gallagher, G. Gange, P. Schachte, H. Søndergaard, and P. J. Stuckey. An iterative approach to precondition inference using constrained Horn clauses. *ICLP'18*, 2018.

Inference of Linear Upper-Bounds on the Expected Cost by Solving Cost Relations

Alicia Merayo and Samir Genaim

Universidad Complutense de Madrid, Spain
{amerayo,sgenaim}@ucm.es

1 Introduction

Resource usage analysis (a.k.a. cost analysis) aims at *statically* determining the number of resources required to safely execute a given program. A *resource* can be any quantitative aspect of the program, such as memory consumption, execution steps, etc. Over the past decade several cost analysis frameworks, for different programming languages, have been developed [6, 8, 12, 14, 3, 9]. They can infer precise upper-bounds on the *worst-case* cost.

There are problems that involve probabilistic choices and for which *worst-case* cost is not the adequate. For example, consider a program that transmits packets of data over the network. Due to network failures, the transmission of a packet might fail and it has to be transmitted again. Assuming that the probability of success (resp. failure) is $\frac{3}{4}$ (resp. $\frac{1}{4}$), our goal is to estimate the number of attempts required in order to successfully transmit n packets. This problem can be modeled using the following loop:

```
while ( n>0 ) {
  n=n-1; ⊕(3/4) skip;
  tick(1);
}
```

where $\oplus$ is a probabilistic choice operator and `tick(1)` is a cost annotation. Note that the left-hand (resp. right-hand) side of $\oplus$ represents a successful (resp. failed) transmission.

If we consider the probabilistic choice as a non-deterministic choice, and analyze this program using a *worst-case* cost analyzer, we would not obtain an upper-bound since the loop is considered non-terminating. The adequate notion of cost for this setting, i.e., in the presence of such probabilistic operations, is the *expected cost*. Let S be an infinite sequence of *independent* choices for the probabilistic operation (i.e., left or right), $\texttt{cost}(n, S)$ be the cost of the execution when taking the choices as indicated in the sequence S, and $\texttt{Pr}(S)$ is the probability of taking such choices (i.e., multiplying the probabilities of all choices), then the expected cost is defined as $\sum_S \texttt{Pr}(S) * \texttt{cost}(n, S)$. Note that the sum is over all possible sequences S, and that $\sum_S \texttt{Pr}(S) = 1$. In this case the expected cost can be reduced to solving the following recurrence relation for $n > 0$ (for $n \leq 0$ we let $C(n) = 0$):

$$C(n) = 1 + \frac{3}{4}C(n-1) + \frac{1}{4}C(n)$$

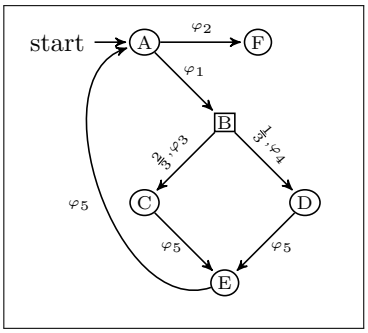
The function $\max(0, \frac{4}{3}n)$ is a possible solution. Note that if we turn $=$ to $\geq$ in the above equation, we get a definition for an upper-bound on the expected cost.

Cost analysis of probabilistic programs and term-rewriting has been recently considered in several works [13, 7, 5, 11]. For example, the work of [13], which is based on an extension of amortized cost analysis, is developed for a simple imperative language and able to infer polynomial bounds on the expected cost of challenging problems.

The goal of our work is to support the inference of expected cost in our cost analyzer `SACO` [1], which infers upper-bounds on the worst-case cost of `ABS` programs [10] — an abstract

© Alicia Merayo and Samir Genaim;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

```
while ( x<n ) {
  x=x+1;⊕ 2/3 x=x-1;
  tick(1);
}
```



$$
\begin{aligned}
\varphi_1 &= \{x \geq n, x' = x, n' = n\} \\
\varphi_2 &= \{x < n, x' = x, n' = n\} \\
\varphi_3 &= \{x' = x + 1, n' = n\} \\
\varphi_4 &= \{x' = x - 1, n' = n\} \\
\varphi_5 &= \{x' = x, n' = n\} \\
& \quad \texttt{cost}(E) = 1
\end{aligned}
$$

Figure 1 Example 1

behaviour modeling language based on concurrent objects. The underlying techniques of SACO for inferring cost are based on first generating a set of cost relations (a general form of recurrence relations) that describes the cost of the program and then solving them into closed-form upper-bounds using a dedicated solver [2]. Thus, we are interested in developing a tool for inferring the expected cost for such cost relations, since it would make the integration with SACO straightforward. In this extended abstract, we report on preliminary results in this direction, in particular on a preliminary implementation that can infer linear upper-bounds on the expected cost of simple control-flow graphs by solving corresponding cost relations.

2 An informal account to our approach

A program in our setting consists of a control-flow graph (CFG) and a tuple of integer variables $\bar{x} = \langle x_1, \cdots, x_m \rangle$. A CFG consists of a set of nodes N representing locations, and a set of edges E representing transitions. Each node n is annotated with a non-negative cost, denoted by $\texttt{cost}(n)$, that corresponds to the resources consumed by each visit to n. Note that we could also annotate transitions with cost, this requires no change in our approach. Each edge $n_1 \xrightarrow{\varphi} n_2$ is annotated with a conjunction of linear constraints φ over variable $\bar{x}$ and primed variables $\bar{x}'$, it defines the transition relation $\{(\bar{v}, \bar{v}') \in \mathbb{Z}^{2m} \mid \bar{x} = \bar{v} \wedge \bar{x}' = \bar{v}' \models \varphi\}$ between nodes n_1 and n_2. For every node we assume that its outgoing transitions, if any, cover the whole state space, i.e., it is always possible to advance. The set of nodes N consists of regular nodes N_r (we draw them as circles) and probabilistic nodes N_p (we draw them as squares). A probabilistic node represents a probabilistic choice, and thus its *outgoing edges are annotated with probabilities that sum to* 1 (in addition to the transition constraints). Moreover, we assume that such edges include only deterministic updates, i.e., the corresponding constraints include only equalities of the form $x' = a_0 + \Sigma_i a_i x_i$ (we can refine this to any non-deterministic update as far as it is always applicable, i.e., covers the whole state space). The idea is that when the execution is at a probabilistic node, the choice of the next node depends on the probabilities associated to the corresponding transitions, and the next state is calculated using the corresponding deterministic update. On the other hand, when the execution is at a regular node the next node is chosen in a non-deterministic way from all applicable transitions. A simple program (taken from [13]) and a corresponding CFG are depicted in Figure 1.

The inference of a linear upper-bound on the expected cost of a given CFG is done in two steps: (1) generation of cost relations from the CFG; and (2) solving the cost relations into a closed-form upper bound. Before solving the cost relations we typically simplify them using unfolding as done in [2]. This simplification step usually has an impact on both the

performance and the precision of the solving process.

Generating cost relations. Each regular node $n \in N_r$ with outgoing transitions $n \xrightarrow{\varphi_i} n_i$, $1 \leq i \leq k$, contributes k cost relations of the form

$$C_n(\bar{x}) = \texttt{cost}(n) + C_{n_i}(\bar{x}') \quad | \quad \varphi_i$$

and each probabilistic node $n \in N_p$ with outgoing edges $n \xrightarrow{p_i, \varphi_i} n_i$, for $1 \leq i \leq k$, contributes one cost relation

$$C_n(\bar{x}) = \texttt{cost}(n) + \sum_{i=1}^{k} p_i \cdot C_{n_i}(\bar{x}'_i) \quad | \quad \biguplus_{i=1}^{k} \varphi_i$$

where $\uplus\varphi_i$ is the union of all φ_i, after renaming the primed variables $\bar{x}'$ of each φ_i to fresh variables $\bar{x}'_i$. Note that we view a conjunction of linear constraints as a set, this is why we use a union. For the `CFG` of Figure 1 we generate the following cost relations:

$$\begin{array}{lll}
C_A(x,n) = & C_F(x',n') & \{x \geq n, x' = x, n' = n\} \\
C_A(x,n) = & C_B(x',n') & \{x < n, x' = x, n' = n\} \\
C_B(x,n) = & \frac{2}{3} \cdot C_C(x'_1,n'_1) + \frac{1}{3} \cdot C_D(x'_2,n'_2) & \{x'_1 = x+1, x'_2 = x-1, n'_1 = n, n'_2 = n\} \\
C_C(x,n) = & C_E(x',n') & \{x' = x, n' = n\} \\
C_D(x,n) = & C_E(x',n') & \{x' = x, n' = n\} \\
C_E(x,n) = & 1 + C_A(x',n') & \{x' = x, n' = n\} \\
C_F(x,n) = & 0 &
\end{array}$$

The meaning of a cost relation is as expected, it recursively defines the cost of node n in terms of its successors in the context of some constraints (those of the corresponding transitions). Note that, due to non-determinism, a query $C(\bar{v})$ might evaluate to several possible values, and thus, an upper-bound is defined to be an upper-bound on the set of all such values. The correctness follows from the weakest-precondition reasoning of [11].

Solving cost relations. Solving cost relations into an upper-bound, on expected cost of the `CFG`, means seeking functions $f_n : \mathbb{Z}^m \mapsto \mathbb{R}^+$, for each node n, such that for each cost relation $\langle C_i(\bar{x}) = c + \sum_j p_j C_j(\bar{x}_j) \mid \varphi \rangle$ the following formula is satisfied (here $\bar{z}$ is the set of all variables that appear in the cost relation):

$$\forall \bar{z}.\ \varphi \models f_i(\bar{x}) \geq c + \sum_j p_j \cdot f_j(\bar{x}_j),$$

This formula states that f_i is enough for paying the local cost c of C_i plus the cost of its successors (taking into account the probabilities p_j). The set of all resulting formulas is then automatically solved as described in [4], with some modifications to take the multiplicands that correspond to probabilities into account. This procedure is based on the use of Farkas' lemma and SMT solving. Briefly, we first define a template function $f_n(\bar{x}) = \max(0, a_0 + \sum a_i x_i)$ for each node n (where a_i are the template parameters) and substitute them in the formulas, then Farkas' lemma is used to transform each formula as the one above into a set of (possibly non-linear) constraints over the template parameters a_i, and finally by solving these constraints using an SMT solver we obtain concrete values for the template parameters, i.e., we instantiate each template f_n. Note that the non-linearity in the generated constraints comes from the fact that our templates are not linear, they include expressions like $\max(0, f)$. In order to apply Farka's lemma we need to eliminate the max first, which we do by splitting

```
while (x+3 <= n) {
  if ( y<m ) {
    skip; ⊕1/2 y=y+1;
  else
    skip; ⊕1/4 x=x+1; ⊕1/4 x= x+2;⊕1/4 x=x+3;
  tick(1);
}
```

$\varphi_1 = \{x+3 \leq n\} \cup \varphi_5$
$\varphi_2 = \{x+3 > n\} \cup \varphi_5$
$\varphi_3 = \{y < m\} \cup \varphi_5 \qquad \texttt{cost}(F) = \texttt{cost}(M) = 1$
$\varphi_4 = \{y' \geq m\} \cup \varphi_5$
$\varphi_5 = \{x' = x, n' = n, y' = y, m' = m\}$
$\varphi_6 = \{x' = x, n' = n, y' = y+1, m' = m\}$
$\varphi_7 = \{x' = x+1, n' = n, y' = y, m' = m\}$
$\varphi_8 = \{x' = x+2, n' = n, y' = y, m' = m\}$
$\varphi_9 = \{x' = x+3, n' = n, y' = y, m' = m\}$

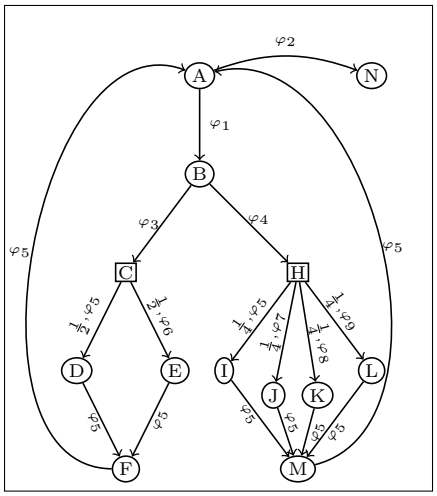


Figure 2 Example 2

the corresponding formula into two cases for $f \leq 0$ and $f \geq 0$. This introduce $f \leq 0$ and $f \geq 0$ to the left-hand of the formula, and since f has template variables we get non-linear constraints when applying Farkas' lemma.

In order to infer an upper-bound for the cost relations that we generated above, we first simplify them using unfolding into the following:

$$\begin{array}{lll} C_A(x,n) = & 0 & \{x \geq n\} \\ C_A(x,n) = & 1 + \frac{2}{3} \cdot C_A(x'_1, n') + \frac{1}{3} \cdot C_A(x'_2, n') & \{x < n, x'_1 = x+1, x'_2 = x-1, n' = n\} \end{array}$$

and then solving them as described above, using the template $f_A(x,n) = \max(0, a_0 + a_1 n + a_2 x)$, results in the upper-bound $f_A(x,n) = \max(0, 3n - 3x)$.

Let us now consider the program depicted in Figure 2, taken from [13], and its corresponding `CFG` that is depicted in the same figure. Generating the cost relations and simplifying them using unfolding we obtain the following cost relations:

$$\begin{array}{lll} C_A(x,y,m,n) = & 0 & \mid \{x+3 > n\} \\ C_A(x,y,m,n) = & 1 + \sum_{i=0}^{1} \frac{1}{2} \cdot C_F(x, y'_i, n, m) & \mid \{x+3 \leq n, y \geq m, y'_0 = y, y'_1 = y+1\} \\ C_A(x,y,m,n) = & 1 + \sum_{i=0}^{3} \frac{1}{4} \cdot C_F(x'_i, y, n, m) & \mid \{x+3 \leq n, y \geq m, x'_0 = x, \\ & & \quad x'_1 = x+1, x'_2 = x+2, x'_3 = x+3\} \end{array}$$

The first one corresponds to the loop exit, and the second (resp. third) one to executing the *then* (resp. *else*) branch of the if-condition. Now in order to solve these cost relations we start by choosing the template $f_A(x,y,m,n) = \max(0, a_0 + a_1 x + a_2 y + a_3 n + a_4 m)$, however, the solver fails to find a solution using this template. In such case we refine our template to include one more component, namely we use $f_A(x,y,m,n) = \max(0, a_0 + a_1 x + a_2 y + a_3 n + a_4 m) + \max(b_0 + b_1 x + b_2 y + b_3 n + b_4 m)$. Now the solver succeeds to find the upper bound $f_A(x,y,m,n) = \max(0, 2m - 2y) + \max(0, \frac{2}{3}n - \frac{2}{3}x)$. Note that, this example required a more complicated template since the loop executes in two independent phases. In the first one we increase y until $y \geq m$ holds, and in the second we increase x until $x + 3 > n$ holds.

3 Concluding remarks

We described a preliminary work on inferring upper-bounds on the expected cost for `CFGs` via cost relations, with the goal of integrating this process in the `SACO` tool. It can also be used to prove *almost sure* termination. Our approach generates a set of cost relations from a given `CFG` and solves them using a technique similar to the one described in [4]. It is currently limited to linear upper-bounds and can handle small examples, e.g., most of the examples with a linear cost that are described in [13].

Out tools is still not as powerful as [13]. We plan to carry on this research direction, mainly to improve the implementation to handle larger programs and handle cases where complex invariants are needed which we do not support yet. We plan to extend the `ABS` language [10] with probabilistic constructs and integrate our implementation in the `SACO`.

References

1 E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *Proc. of TACAS'14*, volume 8413 of *LNCS*, pages 562–567. Springer, 2014.

2 E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

3 E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science (Special Issue on Quantitative Aspects of Programming Languages)*, 413(1):142–159, 2012.

4 D. E. Alonso-Blas and S. Genaim. On the Limits of the Classical Approach to Cost Analysis. In *Proc. of SAS 2012*, volume 7460 of *LNCS*, pages 405–421. Springer, 2012.

5 M. Avanzini, U. Dal Lago, and A. Yamada. On probabilistic term rewriting. In *In Proc. of FLOP'18*, volume 10818 of *LNCS*, pages 132–148. Springer, 2018.

6 M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Analyzing runtime and size complexity of integer programs. *ACM Trans. Program. Lang. Syst.*, 38(4):13:1–13:50, 2016.

7 K. Chatterjee, H. Fu, and A. Murhekar. Automated recurrence analysis for almost-linear expected-runtime bounds. In *In Proc. of CAV'17*, volume 10426 of *LNCS*, pages 118–139. Springer, 2017.

8 S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. of POPL'09*, pages 127–139. ACM, 2009.

9 J. Hoffmann, K. Aehlig, and M. Hofmannn. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14:1–14:62, 2012.

10 E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. of FMCO'10 (Revised Papers)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2012.

11 B. L. Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo. Weakest precondition reasoning for expected run-times of probabilistic programs. In *In Proc. of ESOP'16*, volume 9632 of *LNCS*, pages 364–389. Springer, 2016.

12 Z. Kincaid, J. Breck, A.F. Boroujeni, and T. W. Reps. Compositional recurrence analysis revisited. In *In Proc. of PLDI'17*, pages 248–262. ACM, 2017.

13 V. C. Ngo, Q. Carbonneaux, and J. Hoffmann. Bounded expectations: Resource analysis for probabilistic programs. In *In Proc. of PLDI'18*, 2018. To appear.

14 M. Sinn, F. Zuleger, and H. Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *J. Autom. Reasoning*, 59(1):3–45, 2017.

Embracing Infinity – Termination of String Rewriting by Almost Linear Weight Functions

Dieter Hofbauer

ASW – Berufsakademie Saarland, Germany
d.hofbauer@asw-berufsakademie.de

Abstract

Weight functions are among the simplest methods for proving termination of string rewrite systems, and of rather limited applicability. In this working paper, we propose a generalized approach. As a first step, syllable decomposition yields a transformed, typically infinite rewrite system over an infinite alphabet, as the title indicates. Combined with generalized weight functions, termination proofs become feasible also for systems that are not necessarily simply terminating. The method is limited to systems with linear derivational complexity, however, and this working paper is restricted to original alphabets of size two. The proof principle is almost self-explanatory, and if successful, produces simple proofs with short proof certificates, often even shorter than the problem instance. A prototype implementation was used to produce nontrivial examples.

1 Preliminaries

As usual, a string as a sequence of letters is denoted by $a_0 \dots a_n$ or by $(a_0, \dots, a_n)$ in case the former notation is possibly ambiguous. The *length* of a string x is $|x|$, and $\mathbb{Z}$ denotes the set of integers and $\mathbb{N}$ the set of non-negative integers. The *derivation height* function modulo some terminating rewrite system R over alphabet Σ maps a string $x \in \Sigma^*$ to $\mathrm{dh}_R(x) = \max\{n \in \mathbb{N} \mid \exists y \in \Sigma^* : x \to_R^n y\}$. The *derivational complexity* $\mathrm{dc}_R : \mathbb{N} \to \mathbb{N}$ of R is defined as $\mathrm{dc}_R(n) = \max\{\mathrm{dh}_R(x) \mid |x| \le n\}$, cf. [6].

2 Syllable Decomposition

Decomposing strings by splitting at occurrences of a distinguished letter defines a natural bijection between strings over the original alphabet and strings over a different alphabet, which is typically infinite. This syllable decomposition is used, for instance, in [8] for defining the Kachinuki order, which coincides on strings with the recursive path order from [2]. There, the syllable decomposition is recursively continued, whereas in this paper it appears only as a single first step, similar to the approach in [4].

▶ **Definition 1.** The syllable decomposition of a string $x \in \Sigma^*$ with respect to a letter $a \in \Sigma$ is the string $\mathrm{split}_a(x) = (x_1, x_2, \dots, x_n) \in \Gamma^+$ with $\Gamma = (\Sigma \setminus \{a\})^*$ such that $x = x_1 a x_2 a \dots a x_n$.

Here, a is called the *split letter*. Note that the above decomposition is unique and that $|\mathrm{split}_a(x)| \ge 1$. The alphabet of the resulting string is $\Gamma = (\Sigma \setminus \{a\})^*$, which is infinite except for the trivial case $\Sigma = \{a\}$. In this paper, we consider the simple case of a two-letter alphabet, $\Sigma = \{a, b\}$ say. Therefore, an element of $\Gamma = \{b^n \mid n \in \mathbb{N}\}$ is uniquely determined by its length.

▶ **Definition 2.** For a string $x \in \Sigma^*$ and a letter $a \in \Sigma$ with $\mathrm{split}_a(x) = (x_1, \dots, x_n)$ let $\mathrm{slit}_a(x) = (|x_1|, \dots, |x_n|) \in \mathbb{N}^+$.

This natural bijection between Γ and $\mathbb{N}$ can be extended to string rewriting: For a rewrite system R over Σ there is a rewrite system $\mathrm{slit}_a(R)$ over $\mathbb{N}$ so that $\mathrm{slit}_a : \Sigma^* \to \mathbb{N}^+$

© Dieter Hofbauer;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

is an isomorphism between the relational structures $(\Sigma^*, \to_R)$ and $(\mathbb{N}^+, \to_{\mathrm{slit}_a(R)})$. As a consequence, termination of R is equivalent to termination of $\mathrm{slit}_a(R)$. Just as the underlying alphabet $\mathbb{N}$, the rewrite system $\mathrm{slit}_a(R)$ is infinite in order to properly handle contexts of rule application.

In the sequel, for $p, q \in \mathbb{N}$ and $(k_1, \ldots, k_n) \in \mathbb{N}^+$, we use the overloaded notation $p{+}(k_1, k_2, \ldots, k_{n-1}, k_n){+}q = (p{+}k_1, k_2, \ldots, k_{n-1}, k_n{+}q)$. Note that $p{+}(k_1){+}q = (p{+}k_1{+}q)$.

▶ **Definition 3.** For a rewrite rule $\ell \to r$ over alphabet Σ and $a \in \Sigma$ define the set of rewrite rules $\mathrm{slit}_a(\ell \to r) = \{p + \mathrm{slit}_a(\ell) + q \to p + \mathrm{slit}_a(r) + q \mid p, q \in \mathbb{N}\}$ over alphabet $\mathbb{N}$, and for a rewrite system R over Σ let $\mathrm{slit}_a(R) = \cup_{\ell \to r \in R} \mathrm{slit}_a(\ell \to r)$.

▶ **Example 4.** For the one-rule rewrite system $R = \{aa \to aba\}$ over alphabet $\{a, b\}$ we get $\mathrm{split}_a(R) = \{(\epsilon, \epsilon, \epsilon) \to (\epsilon, b, \epsilon)\}$, thus $\mathrm{slit}_a(R) = \{(p, 0, q) \to (p, 1, q) \mid p, q \in \mathbb{N}\}$.

▶ **Example 5.** Using a different split letter, for the system R in Example 4 we obtain $\mathrm{split}_b(R) = \{(aa) \to (a, a)\}$, thus $\mathrm{slit}_b(R) = \{(p + 2 + q) \to (p + 1, 1 + q) \mid p, q \in \mathbb{N}\}$. Note that $\mathrm{slit}_b(aaa) = (3)$ cannot be reduced by rule $(2) \to (1, 1)$, but instead by the two rules $(3) \to (2, 1)$ and $(3) \to (1, 2)$, corresponding to the two possible applications of R to aaa, first $aaa \to_R abaa$ where $\mathrm{slit}_b(abaa) = (1, 2)$, and second $aaa \to_R aaba$ where $\mathrm{slit}_b(aaba) = (2, 1)$.

▶ **Lemma 6.** *For an alphabet Σ of size two and $a \in \Sigma$, slit_a is a bijection between Σ^* and $\mathbb{N}^+$. Further, for a rewrite system R over Σ and $x, y \in \Sigma^*$, $x \to_R y$ iff $\mathrm{slit}_a(x) \to_{\mathrm{slit}_a(R)} \mathrm{slit}_a(y)$.*

As an immediate consequence, we obtain

▶ **Theorem 7.** *For an alphabet Σ of size two and $a \in \Sigma$, termination of R over Σ and termination of $\mathrm{slit}_a(R)$ over $\mathbb{N}$ are equivalent.*

3 Termination by Almost Linear Weight Functions

Weight functions are among the simplest methods for proving termination of rewrite systems. In the case of string rewriting, a weight function maps letters to numbers, in this paper simply natural numbers, and a weight function $w : \Sigma \to \mathbb{N}$ is additively extended to a weight function on strings $w : \Sigma^* \to \mathbb{N}$ by $w(\epsilon) = 0$ and $w(ax) = w(a) + w(x)$ for $a \in \Sigma$, $x \in \Sigma^*$.

▶ **Lemma 8.** *For a rewrite system R over Σ and a weight function $w : \Sigma \to \mathbb{N}$, if $w(\ell) > w(r)$ for each rule $\ell \to r$ in R, then R is terminating, and R has linear derivational complexity.*

▶ **Example 9.** For $\bar{R} = \{aba \to aa\}$, the inverse of Example 4, choose the weight function $w : \{a, b\} \to \mathbb{N}$ where $w(a) = 0$ and $w(b) = 1$. Then $w(aba) = 1 > 0 = w(aa)$, so for arbitrary strings $x, y \in \Sigma^*$, $x \to_{\bar{R}} y$ implies $w(x) > w(y)$. As $(\mathbb{N}, >)$ is well-founded, this implies termination of $\bar{R}$. Similarly, choosing $w(a) = w(b) = 1$ would reflect the fact that $\bar{R}$ is length-decreasing.

This method, however, is of rather limited applicability. On the one hand, the rewrite system has to be simply terminating (see [1], e. g., and [7] for an in-depth survey), on the other hand its derivational complexity has to be linear. The variant proposed in this paper removes the first restriction, but not the second one.

▶ **Example 10.** No weight function proves termination of $R = \{aa \to aba\}$ from Example 4 since R is not simply terminating. Instead, consider $S = \mathrm{slit}_a(R) = \{(p, 0, q) \to (p, 1, q)\}$ and the weight function $w : \mathbb{N} \to \mathbb{N}$ where $w(0) = 1$ and $w(n) = 0$ for $n \neq 0$. Note that $\mathbb{N}$ is the alphabet of S, thus the domain of w, and also the codomain of w as a weight function. Then

$w((p,0,q)) = w(p)+w(0)+w(q) = w(p)+1+w(q) > w(p)+0+w(q) = w(p)+w(1)+w(q) = w((p,1,q))$, and again, for $x, y \in \mathbb{N}^+$, $x \to_S y$ implies $w(x) > w(y)$, proving termination of S, thus termination of R by Theorem 7.

In the rest of this section, this approach is further explained and justified. As Lemma 8 puts no restriction whatsoever on the weight function, this approach also applies to infinite alphabets and infinite rewrite systems. However, in order to obtain both finite proof obligations and finite proof certificates, only restricted classes of functions come into consideration. In this paper we suggest *almost linear functions* as a first approach; possible extensions are discussed in Section 6. Here, we define an almost linear function as a linear function with a finite set of possible exceptions.

▶ **Definition 11.** An *almost linear function* is the union of a function $e : E \to \mathbb{Z}$ with finite domain $E \subseteq \mathbb{Z}$ and a linear function $h : \mathbb{Z} \setminus E \to \mathbb{Z}$ with $h(n) = an + b$ for $a, b \in \mathbb{Z}$.

As a short-hand notation for such a function f with $e = \{(p_1, q_1), \ldots, (p_k, q_k)\}$ we use $f : p_1 \mapsto q_1, \ldots, p_k \mapsto q_k$, else $n \mapsto an + b$, or simply $f : n \mapsto an + b$ in case $e = \emptyset$.

▶ **Example 12.** (Example 10 cont'd) This introductory example used $w : 0 \mapsto 1$, else $n \mapsto 0$.

▶ **Example 13.** (Example 5 cont'd) For $\mathrm{slit}_b(R) = \{(p + 2 + q) \to (p + 1, 1 + q) \mid p, q \in \mathbb{N}\}$ the almost linear weight function $w : 0 \mapsto 0$, else $n \mapsto n - 1$ succeeds: For $p, q \in \mathbb{N}$ we get $w(p + 2 + q) = p + 2 + q - 1 > p + q = w(p + 1) + w(1 + q)$.

To be suitable as a weight function $w : \mathbb{N} \to \mathbb{N}$, an almost linear function f has to be *non-negative on* $\mathbb{N}$, i. e., $f(\mathbb{N}) \subseteq \mathbb{N}$. This property is easily verified, as follows.

▶ **Lemma 14.** *An almost linear function as in Def. 11 is non-negative on* $\mathbb{N}$ *if and only if* $e(E \cap \mathbb{N}) \subseteq \mathbb{N}$ *and either* $a = 0$ *and* $b \geq 0$*, or* $a > 0$ *and* $\{n \in \mathbb{N} \mid an + b < 0\} \subseteq E$*.*

▶ **Example 15.** Both functions from Examples 12 and 13 are non-negative on $\mathbb{N}$.

As an immediate consequence of Lemma 8 and Theorem 7 we state

▶ **Theorem 16.** *For an alphabet* Σ *of size two,* $a \in \Sigma$*, a rewrite system* R *over* Σ*, and an almost linear function* w*, if* w *is non-negative on* $\mathbb{N}$ *and* $w(\ell) > w(r)$ *for each rule* $\ell \to r$ *in* $\mathrm{slit}_a(R)$*, then* R *is terminating.*

Concerning derivational complexity, we obtain the following result (slightly less trivial, as it depends on Definition 11).

▶ **Theorem 17.** *Under the assumptions of Theorem 16,* R *has linear derivational complexity.*

The class of almost linear functions enjoys some simple closure properties that turn out to be useful for verifying inequalities resulting from interpreted rewrite rules.

▶ **Lemma 18.** *For an almost linear function* f *and a constant* $c \in \mathbb{Z}$*, also* $\lambda n.f(n) + c$ *and* $\lambda n.f(n + c)$ *are almost linear functions.*

▶ Remark. Note that both functions are always non-negative on $\mathbb{N}$ in case f has this property and $c \in \mathbb{N}$. Further note that the coefficients of the linear part of all three functions f, $\lambda n.f(n) + c$ and $\lambda n.f(n + c)$ coincide.

When an almost linear weight function w is used to interpret rules in $\mathrm{slit}_a(R)$, a criterion is needed to ensure, for any given rule $\ell \to r$ in R, that each rule $\ell' \to r'$ in $\mathrm{slit}_a(\ell \to r)$ satisfies $w(\ell') > w(r')$. This can be achieved in a uniform way, as sketched in the following.

▶ **Lemma 19.** *Let f_1, f_2 be almost linear functions with exceptions $e_i : E_i \to \mathbb{Z}$ and linear parts $h_i(n) = an + b_i$ for $i \in \{1, 2\}$, as in Definition 11. Then $\forall n \in \mathbb{N} : f_1(n) > f_2(n)$ if and only if $b_1 > b_2$ and $\forall n \in E_1 \cap \mathbb{N} : e_1(n) > f_2(n)$ and $\forall n \in E_2 \cap \mathbb{N} : f_1(n) > e_2(n)$.*

Note that we assume that the coefficients of h_1 and h_2 coincide, cf. the remark after Lemma 18. A uniform comparison of all rules in $\mathrm{slit}_a(\ell \to r)$ amounts to verify slightly more complex conditions, as the left or right hand side corresponds to one almost linear function or the sum of two such functions, depending on $p + q$ alone, or on both p and q, respectively, by Lemma 18. For this purpose, the case distinction from Lemma 19 can be generalized accordingly; details will be provided in a full version of this paper.

▶ **Example 20.** Let R be `Zantema_06/17` from the Termination Problems Database [9]. Here, the weight function $w : 0 \mapsto 0, 1 \mapsto 1$, else $n \mapsto 4n - 5$ succeeds. Among the 15 rules are $aaaaa \to aaabaaa$ and $aba \to bbb$, so $\mathrm{slit}_b(R)$ contains $\{(p+5+q) \to (p+3, q+3) \mid p, q \in \mathbb{N}\}$ and $\{(p+1, 1+q) \to (p, 0, 0, q) \mid p, q \in \mathbb{N}\}$. For the first set, exceptions in w are irrelevant as $p+5+q > 1$, $p+3 > 1$ and $3+q > 1$, so is suffices to check $w(p+5+q) = 4(p+5+q) - 5 = 4(p+q)+15 > 4(p+q)+14 = 4(p+3)-5+4(3+q)-5 = w(p+3)+w(3+q)$. For the second set, exceptions have to be taken into account, resulting in five cases: $p = q = 0$, $p = 0$ and $q = 1$, $p = 1$ and $q = 0$, $p = q = 1$, and finally $p, q > 1$. For the case $p = q = 1$, for instance, we get $w((1+1, 1+1)) = w(2) + w(2) = 6 > 2 = w(1) + w(0) + w(0) + w(1) = w((1, 0, 0, 1))$.

4 More Examples from the Termination Problems Database

All examples in this section are taken from the Termination Problems Database [9].

▶ **Example 21.** A successful weight function for `Bouchare_06/11` was found with split_a and $w : 0 \mapsto 5, 1 \mapsto 4, 2 \mapsto 16, 3 \mapsto 17, 4 \mapsto 27, 5 \mapsto 30, 6 \mapsto 38$, else $n \mapsto 6n + 1$. Note the large number of exceptions in this and in particular in the following example.

▶ **Example 22.** The system `ICFP_2010/136497` remained unsolved during the termination competitions[1] in 2016 and 2017. Here, split_1 together with $w : 0 \mapsto 0, 1 \mapsto 24, 2 \mapsto 42, 3 \mapsto 60, 4 \mapsto 76, 5 \mapsto 93, 6 \mapsto 109, 7 \mapsto 125, 8 \mapsto 142$, else $n \mapsto 17n + 5$ succeeds.

▶ **Example 23.** For `Bouchare_06/14` both split_a together with $w : 0 \mapsto 6$, else $n \mapsto 10n - 1$ and split_b together with $w : 0 \mapsto 6, 3 \mapsto 13$, else $x \mapsto 3x + 2$ succeed. This example indicates a trade-off between a higher coefficient in the linear part versus more exceptions.

5 Implementation

Most weight functions in this paper have been generated by an experimental prototype implementation within the class library of the termination prover *MultumNonMulta*, see [5]. The experiments performed show the rather limited applicability of the approach, and on the other hand its strength since proofs are (mostly) found instantaneously. The problem set `SRS_Standard` in TPDB Version 10.5 [9] contains 358 rewrite systems over a two-letter alphabet; 22 of them could be solved by our protoype. The simplest way to implement the search for an almost linear weight function consists in completely enumerating all such functions within given bounds for the coefficient and the constant of the linear part and the domain and codomain of the exception part. For examples with many exceptions as in the Examples 21

[1] See `http://termination-portal.org/wiki/Termination_Competition`

and 22, a more elaborate algorithm was applied, involving integer constraint solving using the GNU Linear Programming Kit (GLPK, see `www.gnu.org/software/glpk/`).

6 Discussion and Extensions

Just as with the approach in this paper, a termination proof by match-bounds [3] implies linear derivational complexity, so the relation between both methods might be of interest. As it turns out, both are orthogonal: The system $\{aa \to a\}$ is not match-bounded, but length-reducing. Conversely, the system $\{aabb \to bbbaaa\}$ is match-bounded [3], but it is easily shown that no almost linear function can prove termination of this system.

Further extending the class of weight functions can expand the scope of application of this approach. For instance, the one linear part could be replaced by several linear functions, applied periodically. The following example shows a simple instance of this idea.

▶ **Example 24.** No almost linear weight function was found that proves termination of $R = \{abb \to a, aa \to bbb, bba \to aba\}$ (`Bouchare_06/05`). However, even without any exceptions, $w : 2n \mapsto n + 3$, $2n + 1 \mapsto n$ constitutes a simple termination proof (found and checked manually; an implementation is work in progress).

Other possible extensions of the approach presented here include relative termation proofs and applications to ring (or: cycle) rewriting, cf. [10].

References

1 N. Dershowitz. *A note on simplification orderings.* Inf. Process. Lett. 9(5):212–215, 1979.

2 N. Dershowitz. *Orderings for term-rewriting systems*, Theor. Comput. Sci. 17:279–301, 1982.

3 A. Geser, D. Hofbauer, J. Waldmann. *Match-bounded string rewriting systems.* Appl. Algebra Eng. Commun. Comput. 15(3-4):149–171, 2004.

4 A. Geser, D. Hofbauer, J. Waldmann. *Semantic Kachinuki order.* Proc. 16th Int. Workshop on Termination (WST), 2018.

5 D. Hofbauer. *MultumNonMulta: System description.* Proc. 15th Int. Workshop on Termination (WST), 2016.

6 D. Hofbauer, C. Lautemann. Termination proofs and the length of derivations. Proc. 3rd Int. Conf. on Rewriting Techniques and Applications (RTA), Springer LNCS, Vol. 355, pp. 167–177, 1989.

7 A. Middeldorp, H. Zantema. *Simple termination of rewrite systems.* Theor. Comput. Sci. 175(1):127–158, 1997.

8 K. Sakai. *Knuth-Bendix algorithm for Thue system based on Kachinuki ordering.* ICOT Technical Memorandum TM-0087, Institute for New Generation Computer Technology, 1984.

9 TPDB, The Termination Problems Database, Version 10.5. `termination-portal.org/wiki/TPDB/`

10 H. Zantema, H. J. S. Bruggink, B. König. *Termination of cycle rewriting.* Proc. Int. Conf. RTA-TLCA, Springer LNCS, Vol. 8560, pp. 476–490, 2014.

Improving Static Dependency Pairs for Higher-Order Rewriting

Carsten Fuhs[1] and Cynthia Kop[2]

1 **Birkbeck, University of London, United Kingdom** carsten@dcs.bbk.ac.uk
2 **Radboud University Nijmegen, The Netherlands** c.kop@cs.ru.nl

Abstract

We revisit the static dependency pair method for termination of higher-order term rewriting. In this extended abstract, we propose a static dependency pair framework based on an extended notion of computable dependency chains that harnesses the computability-based reasoning used in the soundness proof of static dependency pairs. This allows us to propose a new termination proving technique to use in combination with static DPs: the *computable* subterm criterion.

1 Introduction

This paper deals with higher-order term rewriting with β-reduction and λ-abstractions. Here a particular topic of interest is *termination*, the property that all (well-formed) terms have only finite reductions. In the first-order setting, the *Dependency Pair (DP) framework* [8] has proven to be an extremely successful foundation for automated termination analysis tools. While several DP approaches (static [12, 14] and dynamic [13, 10]) exist for higher-order rewriting, so far a general DP framework has been proposed only in the PhD thesis [9]. We build on ideas from [2, 9] to propose such a DP framework, here specialised to static DPs, and include a completely new processor which can offer a simple syntactic termination criterion.

2 Algebraic Functional Systems with Meta-variables

Henceforth, we shall assume familiarity with term rewriting, simple types and the λ-calculus. We use a simplified version of *Algebraic Functional Systems with Meta-variables (AFSMs)* that Kop [9] proposes to capture a number of higher-order rewrite formalisms (cf. [9, Ch. 3]).

We fix disjoint sets $\mathcal{F}$ of *function symbols* and $\mathcal{V}$ of *variables*, each symbol a equipped with a type σ. We also fix a set $\mathcal{M}$, disjoint from $\mathcal{F}$ and $\mathcal{V}$, of *meta-variables*, each equipped with a *type declaration* $[\sigma_1 \times \cdots \times \sigma_k] \to \tau$ (where τ and all σ_i are simple types). *Meta-terms* are expressions s where $s : \sigma$ can be derived for some type σ by the following clauses:

(V)	$x : \sigma$	if	$x : \sigma \in \mathcal{V}$	(@)	$s\ t : \tau$	if	$s : \sigma \to \tau$ and $t : \sigma$
(F)	$\mathtt{f} : \sigma$	if	$\mathtt{f} : \sigma \in \mathcal{F}$	(Λ)	$\lambda x.s : \sigma \to \tau$	if	$x : \sigma \in \mathcal{V}$ and $s : \tau$
(M)	$Z[s_1, \ldots, s_k] : \tau$	if	$Z : [\sigma_1 \times \cdots \times \sigma_k] \to \tau \in \mathcal{M}$ and $s_1 : \sigma_1, \ldots, s_k : \sigma_k$				

Terms are meta-terms without meta-variables, so derived without clause (M). *Patterns* are meta-terms where all meta-variable occurrences have the form $Z[x_1, \ldots, x_k]$ with all x_i distinct variables. The λ binds variables as in the λ-calculus. Unbound variables are called *free*, $FV(s)$ is the set of free variables in s, and $FMV(s)$ is the set of meta-variables occurring in s. A meta-term s is *closed* if $FV(s) = \emptyset$. Meta-terms are considered modulo α-conversion. Application (@) is left-associative; abstractions (Λ) extend as far to the right as possible. A meta-term s *has type* σ if $s : \sigma$; it *has base type* if $\sigma \in \mathcal{S}$, the set of *sorts*. A meta-term s has a *sub-meta-term* t (*subterm* if t is a term), written $s \trianglerighteq t$, if (a) $s = t$, (b) $s = \lambda x.s'$ and $s' \trianglerighteq t$, (c) $s = s_1\ s_2$ and $s_1 \trianglerighteq t$ or $s_2 \trianglerighteq t$, or (d) $s = Z[s_1, \ldots, s_k]$ and some $s_i \trianglerighteq t$.

A *meta-substitution* is a type-preserving function γ from variables and meta-variables to meta-terms; if $Z : [\sigma_1 \times \cdots \times \sigma_k] \to \tau$ then $\gamma(Z)$ has the form $\lambda y_1 \ldots y_k.u : \sigma_1 \to \ldots \to \sigma_k \to \tau$.

© Carsten Fuhs and Cynthia Kop;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

Let $\mathtt{dom}(\gamma) = \{x \in \mathcal{V} \mid \gamma(x) \neq x\} \cup \{Z \in \mathcal{M} \mid \gamma(Z) \neq \lambda y_1 \ldots y_k.Z[y_1, \ldots, y_k]\}$ (the *domain* of γ). We let $[b_1 := s_1, \ldots, b_n := s_n]$ be the meta-substitution γ with $\gamma(b_i) = s_i$, $\gamma(z) = z$ for $z \in \mathcal{V} \setminus \{\vec{b}\}$, and $\gamma(Z) = \lambda y_1 \ldots y_k.Z[y_1, \ldots, y_k]$ for $Z \in \mathcal{M} \setminus \{\vec{b}\}$. A *substitution* is a meta-substitution mapping everything in its domain to terms. The result $s\gamma$ of applying a meta-substitution γ to a meta-term s is obtained recursively (with implicit α-conversion):

$$x\gamma = \gamma(x) \text{ if } x \in \mathcal{V} \qquad (s\ t)\gamma = (s\gamma)\ (t\gamma)$$
$$\mathtt{f}\gamma = \mathtt{f} \text{ if } \mathtt{f} \in \mathcal{F} \qquad (\lambda x.s)\gamma = \lambda x.(s\gamma) \text{ if } \gamma(x) = x \wedge x \notin FV(s\gamma)$$
$$Z[s_1, \ldots, s_k]\gamma = t[x_1 := s_1\gamma, \ldots, x_k := s_k\gamma] \text{ if } \gamma(Z) = \lambda x_1 \ldots x_k.t$$

Essentially, applying a meta-substitution with meta-variables in its domain combines a substitution with a β-development, e.g., $X[\mathtt{nil}, \mathtt{0}][X := \lambda x.\mathtt{plus}\ (\mathtt{len}\ x)]$ equals $\mathtt{plus}\ (\mathtt{len}\ \mathtt{nil})\ \mathtt{0}$.

A *rule* is a pair $\ell \Rightarrow r$ of *closed* meta-terms of the same type both in β-normal form with ℓ a pattern of the form $\mathtt{f}\ \ell_1 \cdots \ell_n$ with $\mathtt{f} \in \mathcal{F}$, and $FMV(r) \subseteq FMV(\ell)$. A set of rules $\mathcal{R}$ induces a rewrite relation $\Rightarrow_{\mathcal{R}}$ as the smallest monotonic relation on terms that includes β-reduction (denoted as $\Rightarrow_\beta$) and has $\ell\delta \Rightarrow_{\mathcal{R}} r\delta$ whenever $\ell \Rightarrow r \in \mathcal{R}$ and δ is a substitution on domain $FMV(\ell)$. Rewriting is allowed at any position of a term, even below a λ. $\mathcal{R}$ is terminating if there is no infinite reduction $s_0 \Rightarrow_{\mathcal{R}} s_1 \Rightarrow_{\mathcal{R}} \ldots$. The set $\mathcal{D} \subseteq \mathcal{F}$ of *defined symbols* consists of those $\mathtt{f} \in \mathcal{F}$ such that a rule $\mathtt{f}\ \ell_1 \cdots \ell_n \Rightarrow r$ exists.

An AFSM is a pair $(\mathcal{F}, \mathcal{R})$; types of (meta-)variables can be derived from context.

▶ **Example 1** (Ordinal recursion). Let $\mathcal{F}$ contain at least $\mathtt{0} : \mathtt{ord}$, $\mathtt{s} : \mathtt{ord} \to \mathtt{ord}$, $\mathtt{lim} : (\mathtt{nat} \to \mathtt{ord}) \to \mathtt{ord}$ for ordinals, $\mathtt{zero} : \mathtt{nat}$, $\mathtt{succ} : \mathtt{nat} \to \mathtt{nat}$ for $\mathbb{N}$, and the symbol $\mathtt{rec} : \mathtt{ord} \to \mathtt{nat} \to (\mathtt{ord} \to \mathtt{nat} \to \mathtt{nat}) \to ((\mathtt{nat} \to \mathtt{ord}) \to (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}) \to \mathtt{nat}\}$. Let $\mathcal{R}$ be:

$$\mathtt{rec}\ \mathtt{0}\ K\ F\ G \Rightarrow K, \qquad \mathtt{rec}\ (\mathtt{s}\ X)\ K\ F\ G \Rightarrow F\ X\ (\mathtt{rec}\ X\ K\ F\ G),$$
$$\mathtt{rec}\ (\mathtt{lim}\ H)\ K\ F\ G \Rightarrow G\ H\ (\lambda m.\mathtt{rec}\ (H\ m)\ K\ F\ G)$$

Then $\mathtt{rec}\ (\mathtt{s}\ \mathtt{0})\ \mathtt{zero}\ (\lambda vz.z)\ (\lambda xy.\mathtt{zero}) \Rightarrow_{\mathcal{R}} (\lambda vz.z)\ \mathtt{0}\ (\mathtt{rec}\ \mathtt{0}\ \mathtt{zero}\ (\lambda vz.z)\ (\lambda xy.\mathtt{zero})) \Rightarrow_\beta (\lambda z.z)\ (\mathtt{rec}\ \mathtt{0}\ \mathtt{zero}\ (\lambda vz.z)\ (\lambda xy.\mathtt{zero})) \Rightarrow_\beta \mathtt{rec}\ \mathtt{0}\ \mathtt{zero}\ (\lambda vz.z)\ (\lambda xy.\mathtt{zero}) \Rightarrow_{\mathcal{R}} \mathtt{zero}$.

3 Computability

A common technique in higher-order termination is Tait and Girard's *computability* notion [15]. There are several ways to define computability predicates; here we follow, e.g., [1, 3, 4, 5] in considering *accessible meta-variables* using a form of the *computability closure* [3]:

▶ **Definition 2** (Accessible arguments). We fix a quasi-ordering $\succeq^{\mathcal{S}}$ on the set of sorts (base types) $\mathcal{S}$ with well-founded strict part $\succ^{\mathcal{S}} := \succeq^{\mathcal{S}} \setminus \preceq^{\mathcal{S}}$. For $\sigma \equiv \sigma_1 \to \ldots \to \sigma_m \to \kappa$ (with $\kappa \in \mathcal{S}$) and sort ι, let $\iota \succeq^{\mathcal{S}}_{+} \sigma$ if $\iota \succeq^{\mathcal{S}} \kappa$ and each $\iota \succ^{\mathcal{S}}_{-} \sigma_i$, and let $\iota \succ^{\mathcal{S}}_{-} \sigma$ if $\iota \succ^{\mathcal{S}} \kappa$ and each $\iota \succeq^{\mathcal{S}}_{+} \sigma_i$. (The relation $\iota \succeq^{\mathcal{S}}_{+} \sigma$ corresponds to "ι occurs only positively in σ" in [1, 4, 5].)

For $\mathtt{f} : \sigma_1 \to \ldots \to \sigma_m \to \iota \in \mathcal{F}$, let $Acc(\mathtt{f}) = \{i \mid 1 \leq i \leq m \wedge \iota \succeq^{\mathcal{S}}_{+} \sigma_i\}$. For $x : \sigma_1 \to \ldots \to \sigma_m \to \iota \in \mathcal{V}$, let $Acc(x) = \{i \mid 1 \leq i \leq m \wedge \sigma_i$ has the form $\tau_1 \to \ldots \to \tau_n \to \kappa$ for some $n \in \mathbb{N}$ with $\iota \succeq^{\mathcal{S}} \kappa\}$. We write $s \trianglerighteq_{\mathtt{acc}} t$ if either $s = t$, or $s = \lambda x.s'$ and $s' \trianglerighteq_{\mathtt{acc}} t$, or $s = a\ s_1 \cdots s_n$ with $a \in \mathcal{F} \cup \mathcal{V}$ and $s_i \trianglerighteq_{\mathtt{acc}} t$ for some $i \in Acc(a)$.

▶ **Theorem 3** ($\mathcal{R}$-computability). *For $\mathcal{R}$ a set of rules, there exists a predicate "$\mathcal{R}$-computable" on terms which satisfies the following properties:*

- *$s : \sigma \to \tau$ is $\mathcal{R}$-computable iff $s\ t$ is $\mathcal{R}$-computable whenever $t : \sigma$ is $\mathcal{R}$-computable;*
- *$s : \iota$ for ι a sort is $\mathcal{R}$-computable iff (1) s is terminating under $\Rightarrow_{\mathcal{R}} \cup \Rrightarrow_I$ and (2) if $s \Rightarrow^*_{\mathcal{R}} \mathtt{f}\ s_1 \cdots s_m$ then s_i is $\mathcal{R}$-computable for all $i \in Acc(\mathtt{f})$. Here, $\mathtt{f}\ s_1 \cdots s_m \Rrightarrow_I s_i\ t_1 \cdots t_n$ if both sides have (possibly different) base types, $i \in Acc(\mathtt{f})$, and all t_j are $\mathcal{R}$-computable.*

The above notion of computability is adapted from [1, 3, 4, 5] to account for AFSMs. It is an instance of a *strong computability predicate* following [11], identified by a syntactic criterion. This instance gives a more liberal restriction (in our Def. 9) than their default predicate SC, which is directly used to define the "plain function passing" criterion in [12, 14].

▶ **Example 4.** Consider a quasi-ordering $\succeq^{\mathcal{S}}$ such that $\mathtt{ord} \succ^{\mathcal{S}} \mathtt{nat}$. In Ex. 1, we then have $\mathtt{ord} \succeq^{\mathcal{S}}_{+} \mathtt{nat} \to \mathtt{ord}$. Therefore, $1 \in \mathit{Acc}(\mathtt{lim})$, which gives $\mathtt{lim}\ H \trianglerighteq_{\mathtt{acc}} H$.

4 Static DPs for Accessible Function Passing AFSMs

We will adapt static DPs to our AFSM formalism and propose an alternative applicability criterion. Similar to DPs in the first-order setting, static DPs employ *marked symbols*:

▶ **Definition 5** (Marked symbols, DPs). Define $\mathcal{F}^\sharp := \mathcal{F} \uplus \{\mathtt{f}^\sharp : \sigma \mid \mathtt{f} : \sigma \in \mathcal{D}\}$. For a meta-term s, let $s^\sharp := \mathtt{f}^\sharp\ s_1 \cdots s_k$ if $s = \mathtt{f}\ s_1 \cdots s_k$ with $\mathtt{f} \in \mathcal{D}$; let $s^\sharp := s$ otherwise. A *DP* is a pair $\ell \Rrightarrow p$ where ℓ is a closed pattern $\mathtt{f}\ \ell_1 \cdots \ell_m$, p is a meta-term $\mathtt{g}\ p_1 \cdots p_k$, and both ℓ and p are β-normal and have (possibly different) base types.

The original static approaches define DPs as pairs $\ell^\sharp \Rrightarrow p^\sharp$ with $\ell \Rightarrow r$ a rule and p a subterm $\mathtt{g}\ p_1 \cdots p_k$ of r (their rules use *terms*, not *meta-terms*). This can set bound variables from r free in p. Here, we replace such variables by meta-variables. (So our "variables" mimic (λ-)*bound* variables in functional programming, and our "meta-variables" *free* variables.)

▶ **Definition 6** (SDP). For a meta-term s, $\mathit{metafy}(s)$ denotes s with all free variables replaced by corresponding fresh meta-variables. For an AFSM $(\mathcal{F}, \mathcal{R})$, $SDP(\mathcal{R}) = \{\ell^\sharp \Rrightarrow \mathit{metafy}(p^\sharp) \mid \ell \Rightarrow r \in \mathcal{R} \wedge r \trianglerighteq p \wedge \ell$ and p have base types $\wedge\ p$ has the form $\mathtt{g}\ p_1 \cdots p_k$ for some $\mathtt{g} \in \mathcal{D}\}$.

Right-hand sides of static DPs may contain meta-variables that do not occur on the left:

▶ **Example 7.** For Ex. 1, we obtain $SDP(\mathcal{R}) = \{\mathtt{rec}^\sharp\ (\mathtt{s}\ X)\ K\ F\ G \Rrightarrow \mathtt{rec}^\sharp\ X\ K\ F\ G$, $\mathtt{rec}^\sharp\ (\mathtt{lim}\ H)\ K\ F\ G \Rrightarrow \mathtt{rec}^\sharp\ (H\ M)\ K\ F\ G\}$.

Dependency chains capture sequences of function calls, similar to the first-order setting:

▶ **Definition 8** (Dependency chain, minimal chain). Let $\mathcal{P}$ be a set of DPs and $\mathcal{R}$ be a set of rules. A (finite or infinite) $(\mathcal{P}, \mathcal{R})$-*dependency chain* (or just $(\mathcal{P}, \mathcal{R})$-chain) is a sequence $[(\rho_0, s_0, t_0), (\rho_1, s_1, t_1), \ldots]$ where each $\rho_i \in \mathcal{P}$ and all s_i, t_i are terms, such that for all i:

1. if $\rho_i = \ell_i \Rrightarrow p_i$, then there exists a substitution γ on domain $\mathit{FMV}(\ell_i) \cup \mathit{FMV}(p_i)$ such that $s_i = \ell_i\gamma$ and $t_i = p_i\gamma$; and
2. we can write $t_i = \mathtt{f}\ u_1 \cdots u_n$ with $\mathtt{f} \in \mathcal{F}^\sharp$, $s_{i+1} = \mathtt{f}\ w_1 \cdots w_n$ and each $u_j \Rightarrow^*_{\mathcal{R}} w_j$.

A $(\mathcal{P}, \mathcal{R})$-chain is *minimal* if the strict subterms of all t_i are terminating under $\Rightarrow_{\mathcal{R}}$.

Static DPs are *sound* if the AFSM's rules are *accessible function passing (AFP)*. Intuitively: meta-variables of a higher type may occur only in "safe" places in the left-hand sides of rules.

▶ **Definition 9** (Accessible function passing). An AFSM $(\mathcal{F}, \mathcal{R})$ is *accessible function passing (AFP)* if there exists a sort ordering $\succeq^{\mathcal{S}}$ following Def. 2 such that:

- all function symbols $\mathtt{f}$ are fully applied in $\mathcal{R}$, i.e., they occur only with the maximum number of arguments permitted by their type;
- for all $\mathtt{f}\ \ell_1 \cdots \ell_m \Rightarrow r \in \mathcal{R}$ and all $Z \in \mathit{FMV}(r)$: there are some variables $x_1, \ldots, x_k$ and some i such that $\ell_i \trianglerighteq_{\mathtt{acc}} Z[x_1, \ldots, x_k]$.

This definition is strictly more liberal than the notions of *plain function passing* in [12, 14] as adapted to AFSMs; this lets us handle examples like ordinal recursion (Ex. 1) not covered by [12, 14]. However, [12, 14] consider a different formalism, with polymorphism and rules whose left-hand side is not a pattern. Our restriction is closer to the "admissible" rules in [2], which

are defined using a pattern computability closure [1]. It is also an instance of the ATRFP notion [11], which is parametrised by a strong computability predicate and accessibility relation.

▶ **Example 10.** The AFSM from Ex. 1 is AFP because of the sort ordering $\texttt{ord} \succ^{\mathcal{S}} \texttt{nat}$ (see also Ex. 4), yet it is not plain function passing following [14].

▶ **Theorem 11.** *If $(\mathcal{F}, \mathcal{R})$ is non-terminating and AFP, then there is an infinite minimal $(SDP(\mathcal{R}), \mathcal{R})$-chain.*

This theorem corresponds to results in [2, 11, 12], but imposes a different admissibility restriction: our notion is strictly more liberal than the syntactic criterion in [12], is likely less liberal than the semantic restriction in [11] (although we could not find an example that is ATRFP but not AFP), and mostly (although not entirely) implies the restriction in [2].

The computability inherent in dependency chains using *SDP* lets us strengthen Thm. 11: rather than considering *minimal* chains, we require (some) subterms of all t_i to be *computable*:

▶ **Definition 12.** A $(\mathcal{P}, \mathcal{R})$-chain $[(\ell_0 \Rrightarrow p_0, s_0, t_0), (\ell_1 \Rrightarrow p_1, s_1, t_1), \dots]$ is *$\mathcal{U}$-computable* for a set of rules $\mathcal{U}$ if $\Rightarrow_{\mathcal{U}} \supseteq \Rightarrow_{\mathcal{R}}$, for all i there exists a substitution γ_i with $s_i = \ell_i\gamma_i$ and $t_i = p_i\gamma_i$, and $(\lambda x_1 \dots x_n.v)\gamma_i$ is $\mathcal{U}$-computable for all v such that $p_i \trianglerighteq v$ and $FV(v) = \{x_1, \dots, x_n\}$.

▶ **Theorem 13.** *(a) If an AFSM $(\mathcal{F}, \mathcal{R})$ is non-terminating and AFP, then there is an infinite $\mathcal{R}$-computable $(SDP(\mathcal{R}), \mathcal{R})$-chain. (b) Every $\mathcal{U}$-computable $(\mathcal{P}, \mathcal{R})$-chain is minimal.*

This theorem does not have a true counterpart in the literature. The main result of [11] does require the immediate arguments of each s_i, t_i to be computable, but not other sub-metaterms. Note that the reverse of (a) does *not* hold; terminating AFSMs $\mathcal{R}$ with infinite $\mathcal{R}$-computable $(SDP(\mathcal{R}), \mathcal{R})$-chains do exist [7, Ex. 3.23 (report version 1)].

5 Static DP Framework & Computable Subterm Criterion Processor

The static DP framework follows the first-order DP framework [8], as an extendable framework for proving termination where new termination methods can easily be added as *processors*. In Thm. 16, we will propose a new processor: the *computable subterm criterion*.

Thus far, we have reduced the problem of termination to the non-existence of certain chains. Following the first-order DP framework, we formalise this further via *DP problems*:

▶ **Definition 14** (DP problem)**.** A *DP problem* is a tuple $(\mathcal{P}, \mathcal{R}, m)$ with $\mathcal{P}$ a set of DPs, $\mathcal{R}$ a set of rules, and $m \in \{\mathsf{minimal}, \mathsf{arbitrary}\} \cup \{\mathsf{computable}_{\mathcal{U}} \mid \mathcal{U} \text{ a set of rules}\}$. A DP problem $(\mathcal{P}, \mathcal{R}, m)$ is *finite* if there exists no infinite $(\mathcal{P}, \mathcal{R})$-chain that is $\mathcal{U}$-computable if $m = \mathsf{computable}_{\mathcal{U}}$ or minimal if $m = \mathsf{minimal}$. For the different levels of permissiveness, we use a transitive-reflexive relation $\succeq$ generated by $\mathsf{computable}_{\mathcal{U}} \succeq \mathsf{minimal} \succeq \mathsf{arbitrary}$.

Thm. 13 now becomes: an AFSM $(\mathcal{F}, \mathcal{R})$ is terminating if (but not only if) it is AFP and $(SDP(\mathcal{R}), \mathcal{R}, \mathsf{computable}_{\mathcal{R}})$ is finite. We add a flag value $\mathsf{computable}_{\mathcal{R}}$ over the first-order framework for chains with computability restrictions. The core idea of the DP framework is to simplify a set of DP problems stepwise via *processors* until nothing remains to be proved:

▶ **Definition 15** (Processor)**.** A *dependency pair processor* (or just *processor*) is a function that takes a DP problem and returns a set of DP problems. A processor *Proc* is *sound* if a DP problem M is finite whenever all elements of $Proc(M)$ are finite.

To prove finiteness of a DP problem M: (1) let $A := \{M\}$; (2) while $A \neq \emptyset$: select a $Q \in A$ and a sound processor *Proc*, let $A := (A \setminus \{Q\}) \cup Proc(Q)$. If this terminates, M is a finite DP problem. Many processors are possible; here we present an extension of the subterm criterion [12, 10, 11], dubbed *computable subterm criterion*, that *needs* the new flag.

▶ **Theorem 16** (Computable subterm criterion processor). *Let* $M = (\mathcal{P}_1 \uplus \mathcal{P}_2, \mathcal{R}, \mathsf{computable}_{\mathcal{U}})$ *be a DP problem. A* projection function ν *maps meta-terms to natural numbers such that for all DPs* $\ell \Rrightarrow p \in \mathcal{P}_1 \uplus \mathcal{P}_2$*, the function* $\overline{\nu}$ *with* $\overline{\nu}(\mathtt{f}\ s_1 \cdots s_m) = s_{\nu(\mathtt{f})}$ *is well-defined for* ℓ *and* p*. For meta-terms* s *and* t *of base types, we define* $s \sqsupset t$ *if* $s \neq t$ *and (a)* $s \trianglerighteq_{\mathtt{acc}} t$ *or (b) there exists a meta-variable* Z *with* $s \trianglerighteq_{\mathtt{acc}} Z[x_1, \ldots, x_k]$ *and* $t = Z[t_1, \ldots, t_k]\ s_1 \cdots s_n$*. Then the processor* $\mathit{Proc}_{\mathtt{compsub}}$ *that maps* M *to* $\{(\mathcal{P}_2, \mathcal{R}, \mathsf{computable}_{\mathcal{U}})\}$ *is sound if a projection function* ν *exists with* $\overline{\nu}(\ell) \sqsupset \overline{\nu}(p)$ *for all* $\ell \Rrightarrow p \in \mathcal{P}_1$ *and* $\overline{\nu}(\ell) = \overline{\nu}(p)$ *for all* $\ell \Rrightarrow p \in \mathcal{P}_2$*.*

▶ **Example 17.** $\mathcal{R}$ from Ex. 1 is terminating if $(\mathcal{P}, \mathcal{R}, \mathsf{computable}_{\mathcal{R}})$ with $\mathcal{P} = \mathit{SDP}(\mathcal{R})$ is finite (see Ex. 7). Consider the projection function ν with $\nu(\mathtt{rec}^\sharp) = 1$. As $\mathtt{s}\ X \trianglerighteq_{\mathtt{acc}} X$ and $\mathtt{lim}\ H \trianglerighteq_{\mathtt{acc}} H$, we have $\mathtt{s}\ X \sqsupset X$ and $\mathtt{lim}\ H \sqsupset H\ M$. So $\mathit{Proc}_{\mathtt{compsub}}(\mathcal{P}, \mathcal{R}, \mathsf{computable}_{\mathcal{R}}) = \{(\emptyset, \mathcal{R}, \mathsf{computable}_{\mathcal{R}})\}$. As there are no DPs left, this implies termination of the original $\mathcal{R}$.

6 Conclusion

We have extended the static DP method by a more relaxed applicability criterion and the new *computable subterm criterion*. The full version [7] of the paper has proofs and further extensions, such as *formative* reductions [6, 10], applications to proving non-termination, and dynamic DPs [10] in a unified DP framework with many other processors.

References

1 F. Blanqui. Termination and confluence of higher-order rewrite systems. In *RTA '00*, 2000.
2 F. Blanqui. Higher-order dependency pairs. In *Proc. WST '06*, 2006.
3 F. Blanqui. Termination of rewrite relations on λ-terms based on Girard's notion of reducibility. *Theoretical Computer Science*, 611:50–86, 2016.
4 F. Blanqui, J. Jouannaud, and M. Okada. Inductive-data-type systems. *Theoretical Computer Science*, 272(1-2):41–68, 2002.
5 F. Blanqui, J. Jouannaud, and A. Rubio. The computability path ordering. *Logical Methods in Computer Science*, 11(4), 2015.
6 C. Fuhs and C. Kop. First-order formative rules. In *Proc. RTA-TLCA '14*, 2014.
7 C. Fuhs and C. Kop. The unified higher-order dependency pair framework. Technical Report `arXiv:1805.09390 [cs.LO]`, 2018. `https://arxiv.org/abs/1805.09390`.
8 J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. LPAR '04*, 2005.
9 C. Kop. *Higher Order Termination*. PhD thesis, VU Amsterdam, 2012.
10 C. Kop and F. van Raamsdonk. Dynamic dependency pairs for algebraic functional systems. *Logical Methods in Computer Science*, 8(2):10:1–10:51, 2012.
11 K. Kusakari. Static dependency pair method in functional programs. *IEICE Transactions on Information and Systems*, E101.D(6):1491–1502, 2018.
12 K. Kusakari, Y. Isogai, M. Sakai, and F. Blanqui. Static dependency pair method based on strong computability for higher-order rewrite systems. *IEICE Transactions on Information and Systems*, 92(10):2007–2015, 2009.
13 M. Sakai, Y. Watanabe, and T. Sakabe. An extension of the dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, E84-D(8):1025–1032, 2001.
14 S. Suzuki, K. Kusakari, and F. Blanqui. Argument filterings and usable rules in higher-order rewrite systems. *IPSJ Transactions on Programming*, 4(2):1–12, 2011.
15 W. Tait. Intensional interpretation of functionals of finite type. *Journal of Symbolic Logic*, 32(2):187–199, 1967.

Tool papers

16th International Workshop on Termination (WST 2018).
Editor: Salvador Lucas

TcT: Tyrolean Complexity Tool

Georg Moser and Michael Schaper

Department of Computer Science, University of Innsbruck, Austria
{georg.moser, michael.schaper}@uibk.ac.at

TcT is a fully automated complexity analyser supporting various formal systems and programming languages. Our tool is implemented in `Haskell`, open source, released under the BSD3 license, and available at

`http://cl-informatik.uibk.ac.at/software/tct`.

TcT does not make use of a *unique* problem representation, but employs a *variety* of different representations. The `tct-core` library implements an abstract *complexity framework* and complements it with a simple but powerful problem-independent *strategy language* that facilitates proof search. For details, we refer to [2].

`tct-trs` provides analysis of (innermost) runtime and (innermost) derivational complexity of *term rewrite systems (TRSs)*. This module implements most of the known techniques, see [4] for an overview, and supports proof certification via `CeTA` [3].

To analyse computer programs TcT incorporates *complexity reflecting abstractions*, that is, the resource bound on the obtained abstract programs reflects upon the resource usage of the original programs.

`tct-hoca` incorporates an abstraction of higher-order functional programs to TRSs [1] and makes use of the `tct-trs` module to analyse the resulting problem.

The newest version of TcT is complemented with new techniques for analysing the amortised worst-case runtime complexity of TRSs [5].

References

1 M. Avanzini, U. Dal Lago, and G. Moser. Analysing the complexity of functional programs: higher-order meets first-order. In *Proc. 20th ICFP*, pages 152–164. ACM, 2015.

2 M. Avanzini, G. Moser, and M. Schaper. TcT: Tyrolean Complexity Tool. In *Proc. 22th TACAS*, LNCS, pages 407–423, 2016.

3 M. Avanzini, C. Sternagel, and R. Thiemann. Certification of Complexity Proofs using CeTA. In *Proc. of 26th RTA*, volume 36 of *LIPIcs*, 2015.

4 G. Moser. Proof Theory at Work: Complexity Analysis of Term Rewrite Systems. *CoRR*, abs/0907.5527, 2009. Habilitation Thesis.

5 G. Moser and M. Schneckenreither. Automated Amortised Resource Analysis for Term Rewrite Systems. In *Proc. of 14th FLOPS*, pages 214–229, 2018.

© Georg Moser and Michael Schaper;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

AProVE at the Termination Competition 2018

M. Brockschmidt[1], S. Dollase[2], F. Emrich[3], F. Frohn[2], C. Fuhs[4], J. Giesl[2], M. Hark[2], J. Hensel[2], D. Korzeniewski[2], M. Naaf[2], and T. Ströder[2]

1 **Microsoft Research, Cambridge, UK**
2 **LuFG Informatik 2, RWTH Aachen University, Germany**
3 **University of Edinburgh, UK**
4 **Birkbeck, University of London, UK**

AProVE is a tool for termination and complexity proofs of Java, C, Haskell, Prolog, and rewrite systems (possibly with built-in integers). To analyze programs, AProVE automatically converts them to term rewrite systems (TRSs) with built-in integers or to integer transition systems (ITSs). Then, numerous techniques are employed to prove termination and to infer complexity bounds for the resulting rewrite systems. The generated proofs can be exported to check their correctness using automatic certifiers. See [3] for an overview and the techniques implemented in AProVE. The following features were recently added:

Improving the Inference of Runtime Complexity Bounds. We developed an easy-to-check criterion which allows us to use all techniques for innermost runtime complexity analysis of TRSs to analyze full runtime complexity, too [1]. Moreover, we integrated a transformation from TRSs to ITSs such that complexity tools for ITSs can now also be applied for modular complexity analysis of TRSs [6]. Finally, we improved our techniques for complexity analysis of ITSs, resulting in a re-implementation of our ITS-complexity analyzer KoAT.

Improving the Termination Analysis of C Programs. We improved AProVE's capabilities for termination analysis of C programs further, in particular for disproving termination and for termination proofs of recursive C programs, even if these programs use pointer arithmetic combined with direct memory access [4].

Complexity Analysis for Java and C Programs. We modified our existing transformation from Java to integer TRSs into a transformation which generates ITSs and allows us to use back-end complexity analysis tools for ITSs to infer complexity bounds for Java programs [2]. Moreover, we also adapted complexity analysis techniques in order to infer bounds for C programs that operate on bitvector integers [5].

References

1 F. Frohn and J. Giesl. Analyzing runtime complexity via innermost runtime complexity. In *Proc. LPAR '17*, EPiC 46, pages 249–268, 2017.
2 F. Frohn and J. Giesl. Complexity analysis for Java with AProVE. In *Proc. iFM '17*, LNCS 10510, pages 85–101, 2017.
3 J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning*, 58(1):3–31, 2017.
4 J. Hensel, F. Emrich, F. Frohn, T. Ströder, and J. Giesl. AProVE: Proving and disproving termination of memory-manipulating C programs (competition contribution). In *Proc. TACAS '17*, LNCS 10206, pages 350–354, 2017.
5 J. Hensel, J. Giesl, F. Frohn, and T. Ströder. Termination and complexity analysis for programs with bitvector arithmetic by symbolic execution. *Journal of Logical and Algebraic Methods in Programming*, 97:105–130, 2018.
6 M. Naaf, F. Frohn, M. Brockschmidt, C. Fuhs, and J. Giesl. Complexity analysis for term rewriting by integer transition systems. *Proc. FroCoS '17*, LNAI 10483, pp. 132-150, 2017.

© Brockschmidt, Dollase, Emrich, Frohn, Fuhs, Giesl, Hark, Hensel, Korzeniewski, Naaf, Ströder;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

TermComp 2018 Participant: TTT2*

Florian Meßner and Christian Sternagel

University of Innsbruck, Austria
{florian.g.messner,christian.sternagel}@uibk.ac.at

The *Tyrolean Termination Tool* (in its 2nd incarnation TTT2 [2]) is an automated tool for proving (and disproving) termination of term rewrite systems that is developed in the *Computational Logic* group at the University of Innsbruck in Austria.

http://cl-informatik.uibk.ac.at/ttt2

Besides various minor changes and improvements, the most notable addition to version v1.17 of TTT2 for this years termination competition is the following.

External Nonreachability Tools. We added an interface that supports external nonreachability tools to the edg processor that computes an *estimated dependency graph*. If such a tool is given, it is called for a potential edge, whenever the checks implemented in TTT2 could not prove nonreachability. The user can provide any program that takes a TRS on startup and individual nonreachability problems successively on standard input. Results are read from standard output, where the answer NO is interpreted as nonreachability and every other string as "don't know."

Related to this extension of TTT2, we developed nonreach, a tool for nonreachability analysis that comes with a collection of fast checks based on tcap [1] and the *symbol transition graph* [3]. Moreover, nonreach employs root-nonreachability checks—based on variations of these fast nonreachability checks—to decompose problems into a disjunction of smaller subproblems. Finally, nonreach makes use of narrowing to transform a problem into a conjunction of (potentially easier) problems. Taken together, we obtain a divide and conquer approach to check for nonreachability.

References

1 Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. Proving and Disproving Termination of Higher-Order Functions. In *Proceedings of the 5th International Workshop on Frontiers of Combining Systems (FroCoS)*, volume 3717 of *Lecture Notes in Artificial Intelligence*, pages 216–231. Springer, 2005. doi:10.1007/11559306_12.

2 Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean Termination Tool 2. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA)*, volume 5595 of *Lecture Notes in Computer Science*, pages 295–304. Springer, 2009. doi:10.1007/978-3-642-02348-4_21.

3 Akihisa Yamada. Reachability for termination. 4th Austria–Japan Summer Workshop on Term Rewriting (AJSW), 2016. http://cl-informatik.uibk.ac.at/users/ayamada/AJSW2016-slides.pdf.

* The research described in this paper is supported by FWF (Austrian Science Fund) project P27502.

© Florian Meßner and Christian Sternagel;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

MultumNonMulta at TermComp 2018

Dieter Hofbauer

ASW – Berufsakademie Saarland, Germany
d.hofbauer@asw-berufsakademie.de

MultumNonMulta (MnM) proves termination and non-termination of string rewriting systems automatically (available at `http://dieter-hofbauer.de/software/MnM`). The purpose of this prototype implementation is to demonstrate the power of a combination of few but strong termination proof methods.

Termination proofs of MnM are based on matrix interpretations as described in [7]. Among the various approaches for synthesizing matrix interpretations are complete enumeration of restricted matrix shapes and, most prominently, constraint solving. MultumNonMulta uses a *backward completion procedure* as another approach, exploiting the view of matrix interpretations as weighted automata [5]. No external constraint solver is involved, with the only exception of the GNU Linear Programming Kit (GLPK) for generating weight orderings as a particular form of matrix interpretations.

Only four transformation methods on rewriting systems are employed: The basic version of the dependency pairs transformation [1] without any refinements, a simple form of context closure (R over Σ maps to $\{\mathrm{fold}(x\ell y) \to \mathrm{fold}(xry) \mid \ell \to r \in R, x, y \in \Sigma\}$ over Σ^2, where $\mathrm{fold}(a_1 \dots a_n) = (a_1, a_2) \dots (a_{n-1}, a_n)$, cf. [8]), string reversal (each rule $\ell \to r$ is replaced by $\mathrm{rev}(\ell) \to \mathrm{rev}(r)$, where $\mathrm{rev}(a_1 \dots a_n) = a_n \dots a_1$) and, for length-preserving systems, rule inversion (each rule $\ell \to r$ is replaced by $r \to \ell$).

The competition version of MnM comprises a loop-checker (formerly the separate tool KnockedForLoops) for proving non-termination [4, 9]. It implements a brute-force breadth-first enumeration of forward closures, based on the fact that the existence of a loop is equivalent to the existence of a looping forward closure [3], together with a simple combinatorial optimization. For experimental comparisons with other approachs see [9, 2].

Recent experiments on termination proofs by almost linear weight functions [6] are implemented within MnM, but this feature was switched off for TermComp 2018 due to its currently limited performance.

References

1 T. Arts, J. Giesl. *Termination of term rewriting using dependency pairs*. Theoretical Computer Science 236:133–178, 2000.

2 F. Emmes, T. Enger, J. Giesl. *Proving non-looping non-termination automatically*. Proc. IJCAR, Springer LNCS 7364, pp. 225–240, 2012.

3 A. Geser, H. Zantema. *Non-looping string rewriting*. Informatique Théorique et Applications 33(3):279–302, 1999.

4 D. Hofbauer. *System description: KnockedForLoops*. Proc. WST, p. 51, 2009.

5 D. Hofbauer. *Synthesizing matrix interpretations via backward completion*. Proc. WST, pp. 56–58, 2013.

6 D. Hofbauer. *Embracing infinity – Termination of string rewriting by almost linear weight functions*. Proc. WST, this volume, 2018.

7 D. Hofbauer, J. Waldmann. *Termination of string rewriting with matrix interpretations*. Proc. RTA, Springer LNCS 4098, pp. 328–342, 2006.

8 C. Sternagel, A. Middeldorp. *Root-labeling*. Proc. RTA, pp. 336–350, 2008.

9 H. Zankl, C. Sternagel, D. Hofbauer, A. Middeldorp. *Finding and certifying loops*. Proc. SOFSEM, Springer LNCS 5901, pp. 755–766, 2010.

© Dieter Hofbauer;

Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

Ultimate Büchi Automizer

Matthias Heizmann, Daniel Dietsch, and Alexander Nutz

University of Freiburg

The tool Ultimate Büchi Automizer implements an automata-based termination analysis [5] for computer programs. This analysis can be explained using the notion of a *program trace* which we define as an infinite sequence of statements that occurs as a labeling along an infinite path in a control flow graph. In this analysis, we iteratively take a lasso-shaped (i.e., ultimately periodic) sample trace π from the program $\mathcal{P}$ that we analyze. Next, we apply the nontermination analysis [8] and the termination analysis [3, 7] of the LassoRanker tool in order to find out if there exists a nonterminating execution along the trace π. If LassoRanker proves termination of π, it returns a ranking function f_π for π. Ultimate Büchi Automizer then generalizes π to a set of traces S_π such that f_π is a ranking function for each of them. In order to make sure that elements of S_π are not considered in subsequent iterations we represent S_π as a Büchi automaton and construct an automata theoretic difference between the set of all traces whose termination is not yet known. This process is repeated until a nonterminating trace was found or termination of all traces has been proven.

In order to handle recursive programs, we use an interprocedural analysis [4] that is based on nested word automata. As optimizations, we apply an automata minimization [6] for nested word automata and try to construct [2] a semi-deterministic Büchi automaton for S_π because for this class of automata an efficient complementation [1] is available.

References

1 Frantisek Blahoudek, Matthias Heizmann, Sven Schewe, Jan Strejcek, and Ming-Hsien Tsai. Complementing semi-deterministic Büchi automata. In *TACAS*, volume 9636 of *Lecture Notes in Computer Science*, pages 770–787. Springer, 2016.

2 Yu-Fang Chen, Matthias Heizmann, Ondrej Lengál, Yong Li, Ming-Hsien Tsai, Andrea Turrini, and Lijun Zhang. Advanced automata-based algorithms for program termination checking. In *PLDI*, pages 135–150. ACM, 2018.

3 Matthias Heizmann, Jochen Hoenicke, Jan Leike, and Andreas Podelski. Linear ranking for linear lasso programs. In *ATVA*, volume 8172 of *Lecture Notes in Computer Science*, pages 365–380. Springer, 2013.

4 Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Nested interpolants. In *POPL*, pages 471–482. ACM, 2010.

5 Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Termination analysis by learning terminating programs. In *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 797–813. Springer, 2014.

6 Matthias Heizmann, Christian Schilling, and Daniel Tischner. Minimization of visibly pushdown automata using partial max-sat. In *TACAS (1)*, volume 10205 of *Lecture Notes in Computer Science*, pages 461–478, 2017.

7 Jan Leike and Matthias Heizmann. Ranking templates for linear loops. *Logical Methods in Computer Science*, 11(1), 2015. URL: `https://doi.org/10.2168/LMCS-11(1:16)2015`, `doi:10.2168/LMCS-11(1:16)2015`.

8 Jan Leike and Matthias Heizmann. Geometric nontermination arguments. In *TACAS (2)*, volume 10806 of *Lecture Notes in Computer Science*, pages 266–283. Springer, 2018.

© Matthias Heizmann;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

MU-TERM at the 2018 Termination Competition*

Raúl Gutiérrez and Salvador Lucas

DSIC, Universitat Politècnica de València, Spain

MU-TERM 5.18 is a tool which can be used to verify *termination* and *operational termination* of variants of *Term Rewriting Systems* (TRSs) using different variants of the *Dependency Pair* (DP) *Framework*:

- Termination and innermost termination of TRSs using the DP Framework for TRSs [4]. This framework is also used to prove termination of *String Rewriting Systems*.
- Termination and innermost termination of *context-sensitive rewriting* using the *Context-Sensitive DP Framework* for Context-Sensitive TRSs (CS-TRSs) [1, 5].
- Termination of *rewriting modulo associative/commutative theories* using the *A∨C-DP Framework* for TRSs with *associative*, *commutative* and/or *associative-commutative* axioms [2].
- Termination of *order-sorted rewriting* using the *Order-sorted DP Framework* for Order-Sorted TRSs (in Maude format) [7].
- Operational termination of Conditional TRSs (CTRSs) using the *2D DP Framework* [8].

MU-TERM is written in Haskell. More than 80 *processors* have been implemented for the aforementioned DP frameworks and used in the implemented proof strategies. In some of these processors, AGES [6] and Barcelogics [3] are used as a backend to synthesize logical models based on convex polytopic domains using values in $\mathbb{N}$, $\mathbb{Z}$, or $\mathbb{Q}$ for the polynomial and (piecewise) matrix interpretations of function and predicate symbols involved in the proofs.

References

1 B. Alarcón, R. Gutiérrez, and S. Lucas. Context-Sensitive Dependency Pairs. *Information and Computation*, 208:922–968, 2010.

2 B. Alarcón, S. Lucas, and J. Meseguer. A Dependency Pair Framework for A∨C-Termination. In *Proc. of WRLA'10*, volume 6381 of *LNCS*, pages 35–51. Springer, 2010.

3 M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. The Barcelogic SMT Solver. In *Proc. of CAV'08*, volume 5123 of *LNCS*, pages 294–298. Springer, 2008.

4 J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and Improving Dependency Pairs. *Journal of Automatic Reasoning*, 37(3):155–203, 2006.

5 R. Gutiérrez and S. Lucas. Proving Termination in the Context-Sensitive Dependency Pairs Framework. In *Proc of WRLA'10*, volume 6381 of *LNCS*, pages 19-35. Springer, 2010.

6 R. Gutiérrez, S. Lucas, and P. Reinoso. A Tool for the Automatic Generation of Logical Models of Order-Sorted First-Order Theories. In *Proc. of PROLE'16*, pages 215–230, 2016.

7 S. Lucas and J. Meseguer. Order-Sorted Dependency Pairs. In *Proc. of PPDP'08*, pages 108–119. ACM Press, 2008.

8 S. Lucas, J. Meseguer, and R. Gutiérrez. The 2D Dependency Pair Framework for conditional rewrite systems. Part I: Definition and basic processors. *Journal of Computer and System Sciences*, 96:74–106, 2018.

* Partially supported by the EU (FEDER), Spanish MINECO project TIN2015-69175-C4-1-R and GV project PROMETEOII/2015/013. Raúl Gutiérrez was also supported by INCIBE program "Ayudas para la excelencia de los equipos de investigación avanzada en ciberseguridad".

© Raúl Gutiérrez and Salvador Lucas;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas

iRankFinder

Jesús J. Doménech and Samir Genaim

Universidad Complutense de Madrid, Spain
{jdomenec,sgenaim}@ucm.es

iRankFinder is an open-source[1] termination analyzer for (integer) transition systems, namely control-flow graphs where edges are annotated with transition relations defined by linear constraints. It is written in Python (compatible with 2.7 and 3.X) and uses the Parma Polyhedra Library (PPL) [2] and Z3 [6] for manipulating (linear) constraints. It can be used via a web-interface[2] or from a command-line.

The underlying techniques for proving termination are based on finding lexicographic-linear ranking functions, by incrementally ranking transitions (at the level of strongly connect components) using quasi-ranking functions. These are functions that are required to decrease for some transitions and not to increase for the rest. In particular, it includes the following quasi-ranking functions: nested-QLRFs [4], BG-QLRFs [3], ADFG-QLRFs [1] and BMS-QLRFs [5]. It also supports ranking strongly connect components using linear ranking functions.

As a preprocessing step, iRankFinder can enrich the transitions with supporting invariants inferred using the polyhedra abstract domain, and apply control-flow refinement [7] to make implicit control-flow explicit.

References

1 Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis Symposium, SAS'10*, volume 6337 of *LNCS*, pages 117–133. Springer, 2010.

2 R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

3 Amir M. Ben-Amram and Samir Genaim. Ranking functions for linear-constraint loops. *Journal of the ACM*, 61(4):26:1–26:55, jul 2014.

4 Amir M. Ben-Amram and Samir Genaim. On multiphase-linear ranking functions. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification, CAV'17*, volume 10427 of *LNCS*, pages 601–620. Springer, 2017.

5 Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, CAV'05*, volume 3576 of *LNCS*, pages 491–504. Springer, 2005.

6 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

7 Jesús J. Doménech, Samir Genaim, and John P. Gallagher. Control-flow refinement via partial evaluation. In *International Workshop on Termination, WST'18*, 2018.

[1] http://github.com/jesusjda/pyRankFinder
[2] http://irankfinder.loopkiller.com

© Jesús J. Doménech and Samir Genaim;
Proceedings of the 16th International Workshop on Termination;
Editor: Salvador Lucas