18th International Workshop on Termination

WST 2022, August 11–12, 2022, affiliated with IJCAR-11 at FLoC 2022
https://sws.cs.ru.nl/WST2022

Edited by

Cynthia Kop

Preface

This report contains the proceedings of the 18th International Workshop on Termination (WST 2022), which was held in Haifa during August 11–12 as part of the Federated Logic Conferences (FLoC) 2022.

The Workshop on Termination traditionally brings 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. The 18th International Workshop on Termination continues the successful workshops held 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), Obergurgl (2016), and Oxford (2018), and the virtual event in 2021.

The WST 2022 program included an invited talk by René Thiemann on *Efficient Formalization of Simplification Orders*. WST 2022 received 7 regular submissions and 10 abstracts for tool presentations, 3 of which were accompanied by a system description. After light reviewing the program committee decided to accept all submissions. The 10 contributions are contained in these proceedings.

I would like to thank the program committee members for their dedication and effort, and the workshop chairs of FLoC 2022 for the invaluable help in the organization.

Nijmegen, July 2022 Cynthia Kop

Organization

Program Committee

José Divason	Universidad de La Rioja
Florian Frohn	AbsInt GmbH
Jera Hensel	RWTH Aachen
Dieter Hofbauer	ASW Saarland
Sebastiaan Joosten	Darthmouth College
Cynthia Kop (chair)	Radboud University Nijmegen
Akihisa Yamada	AIST, Japan
Hans Zantema	Eindhoven, University of Technology

Contents

Efficient Formalization of Simplification Orders

René Thiemann
University of Innsbruck, Austria

Akihisa Yamada
National Institute of Advanced Industrial Science and Technology, Japan

Abstract

The weighted path order (WPO) can simulate several simplification orders that are known in term rewriting. By integrating multiset comparisons into WPO, we show that also the recursive path ordering is covered. Moreover, we investigate how refinements of the classical simplification orders can efficiently be integrated: we formally prove the refinements within WPO once and then get them for free for the other simplification orders by the simulation property. Here, the most challenging part was to show that a refined version of the Knuth–Bendix order can actually be simulated by WPO. All of our proofs have been formalized in Isabelle/HOL.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Equational logic and rewriting

Keywords and phrases formalization, Isabelle/HOL, simplification order, termination analysis

Category invited paper

1 Introduction

Automatically proving termination of *term rewrite systems* has been an active field of research for half a century. A number of *simplification orders* [2, 3] are classic methods for proving termination, and these are still integrated in several current termination tools. Classical simplification orders are *Knuth–Bendix orders (KBO)* and *lexicographic* and *recursive path orders (LPO and RPO)*. The *weighted path order (WPO)* [6] was introduced as a simplification order that unifies and extends classical ones.

When switching from theory to implementations in termination tools, limitations of the applied simplification orders become visible while studying non-successful termination proofs. Therefore several refinements of the original definitions of the orders have been developed to make them more applicable and hence more powerful. At this point the question of soundness arises, in particular whether the main properties of a simplification order are still maintained after the integration of the refinements.

To solve this problem we propose to use formal verification, i.e., one should define the orders within a proof assistant such as Coq or Isabelle and then perform the proofs within that system. The advantage is that then re-checking of proofs is quite simple, and in particular a change of a definition (e.g., triggered by some refinement) will immediately point to those parts of the proof which need an adjustment.

The price of using formal verification is its overhead in comparison to a pure proof on paper. In this work we present our approach to perform verification efficiently, namely by exploiting the property that WPO subsumes several simplification orders:

- Instead of formally verifying that KBO, LPO, RPO and WPO are simplification orders, we just prove this fact for WPO and we formally verify that KBO, LPO and RPO are instances of WPO. To this end, we slightly refine WPO itself by permitting multiset comparisons.

- We further show that several refinements of simplification orders are sound for WPO, and hence only have to integrate these refinements into one order, and automatically get the refinements for the other orders, too.

We perform our formalization using Isabelle/HOL, based on IsaFoR, the **Isa**belle **Fo**rmalization of **R**ewriting [5]. As a result of this work we were able to completely remove the formal proofs within IsaFoR that RPO is a simplification order (which entails that LPO is a simplification) order, and we could also remove several formal proofs regarding KBO.

2 Preliminaries

We assume familiarity with term rewriting [1], but briefly recall notions that are used in the following. A *term* built from *signature* $\mathcal{F}$ and set $\mathcal{V}$ of variables is either $x \in \mathcal{V}$ or of form $f(t_1, \ldots, t_n)$, where $f \in \mathcal{F}$ is n-ary and $t_1, \ldots, t_n$ are terms. A *context* C is a term with one hole, and $C[t]$ is the term where the hole is replaced by t. The *subterm relation* $\trianglerighteq$ is defined by $C[t] \trianglerighteq t$. A *substitution* is a function σ from variables to terms, and we write $t\sigma$ for the *instance* of term t in which every variable x is replaced by $\sigma(x)$.

A *reduction pair* is a pair $(\succ, \succsim)$ of two relations on terms that satisfies the following requirements: $\succ$ is well-founded, $\succsim$ and $\succ$ are compatible (i.e., $\succsim \circ \succ \circ \succsim \subseteq \succ$), both are closed under substitutions, and $\succsim$ is closed under contexts. If additionally $\succ$ is transitive, closed under contexts, and contains the strict subterm relation $\triangleright$, then $\succ$ is a *simplification order*. The *trivial reduction pair* is the one where $\succ = \emptyset$ and $\succsim$ relates all terms.

A *quasi-precedence* is a preorder $\geqslant$ on $\mathcal{F}$, such that $> := \geqslant \setminus \leqslant$ is well-founded. A *precedence* is a quasi-precedence where $\geqslant$ is the reflexive closure of $>$.

We use the following notation for common extensions of a pair of relations over terms to pairs of relations over lists of terms.

- $\succ^{\mathsf{mul}}$ and $\succsim^{\mathsf{mul}}$ are the strict- and non-strict order of the *multiset extension* of $(\succ, \succsim)$, where the lists are interpreted as multisets;
- There are two variants of the lexicographic extension: the *unbounded lexicographic extension* is defined as $[s_1, \ldots, s_i, \ldots] \succ^{\mathsf{lex}} [t_1, \ldots, t_i, \ldots]$ iff $s_i \succ t_i$ and $s_j \succsim t_j$ for all $j < i$. The *bounded lexicographic extension* is parametrized by some $b \in \mathbb{N}$, the bound, and it is defined as $[s_1, \ldots, s_n] \succ^{\mathsf{lex},b} [t_1, \ldots, t_m]$ iff $[s_1, \ldots, s_n] \succ^{\mathsf{lex}} [t_1, \ldots, t_m] \land (n = m \lor m \leq b)$. There are similar definitions for $\succsim^{\mathsf{lex}}$ and $\succsim^{\mathsf{lex},b}$. We sometimes write $\succ^{\mathsf{lex}}$ and $\succsim^{\mathsf{lex}}$ also for the bounded lexicographic extension if the bound is clear from the context.

3 Structure of Simplification Orders

In this section we first define some quite generic relation (a simplified version of WPO) that is a template of several simplification orders, and we will then see how KBO, LPO, RPO and WPO fit into this framework. Moreover, we will also discuss refinements and their soundness.

Let $\geqslant$ be some quasi-precedence. Let $b \in \mathbb{N}$ be some bound which will be used as parameter for bounded lexicographic comparisons in the upcoming definition. Let $\tau : \mathcal{F} \to \{\mathsf{lex}, \mathsf{mul}\}$ be a status. Let *minimal* be some property of constants, such that whenever c is minimal then $f \geqslant c$ for all $f \in \mathcal{F}$. Let $(\succ, \succsim)$ be some reduction pair such that $\succ$ is transitive, $\succsim$ is a preorder, and $C[t] \succsim t$ for all terms t. We define a strict and a non-strict relation on terms ($\succ_{\mathsf{RoT}}$ and $\succsim_{\mathsf{RoT}}$) as follows: $s \succ_{\mathsf{RoT}} t$ iff

1. $s \succ t$, or
2. $s \succsim t$ and

a. $s = f(s_1, \ldots, s_n)$ and $\exists i \in \{1, \ldots, n\}.\ s_i \succsim_{\mathsf{RoT}} t$, or
b. $s = f(s_1, \ldots, s_n)$, $t = g(t_1, \ldots, t_m)$ and
  i. $\forall j \in \{1, \ldots, m\}.\ s \succ_{\mathsf{RoT}} t_j$ and
  ii. A. $f \succ g$ or
    B. $f \succsim g$ and $\tau(f) = \tau(g)$ and $[s_1, \ldots, s_n] \succ_{\mathsf{RoT}}^{\tau(f)} [t_1, \ldots, t_m]$.

The relation $s \succsim_{\mathsf{RoT}} t$ is defined in the same way, where $\succ_{\mathsf{RoT}}^{\tau(f)}$ in case **2.b.ii.B** is replaced by $\succsim_{\mathsf{RoT}}^{\tau(f)}$, and there are two additional subcases in case **2**:

c. $s = x = t$ for some $x \in \mathcal{V}$, or
d. $s = x \in \mathcal{V}$ and $t = c$ for some constant c which is minimal.

Using this generic relation on terms we can now define common instances:

- Classical LPO is obtained by
  - using the trivial reduction pair, so that **1** never applies and the condition in line **2** is always satisfied;
  - requiring a precedence so that **2.b.ii.B** is only applicable if $f = g$, i.e., only lists of the same length are compared;
  - no constant is minimal, so case **2.d** is just dropped;
  - $\tau(f) = \mathsf{lex}$ for all $f \in \mathcal{F}$.
- Classical RPO is like LPO without the requirement on τ.
- Classical KBO is similar to the setup of LPO, but
  - instead of the trivial reduction pair one defines $(\succ, \succsim)$ with the help of weight functions and the multisets of variables of terms;
  - the structure of KBO and of $(\succ_{\mathsf{RoT}}, \succsim_{\mathsf{RoT}})$ is slightly different since in KBO, condition **2.b.i** is not present and case **2.a** is also dropped; moreover in KBO there is one additional case, namely whenever $s \notin \mathcal{V}$, $x \in \mathcal{V}$ and $s \succsim x$, then both $s \succ_{\mathsf{KBO}} x$ and $s \succsim_{\mathsf{KBO}} x$.

▶ Remark. The relation $\succ_{\mathsf{RoT}}$ defined above looks like a simplified form of WPO, e.g., the status function π of WPO (for selecting arguments of each individual function symbol) has been omitted. However, the original WPO does not completely subsume $\succ_{\mathsf{RoT}}$, since the status function τ of $\succ_{\mathsf{RoT}}$ is not included in WPO and one would always compare lists of terms lexicographically in WPO.

Let us now regard two refinements of the classical simplification orders. The first refinement are quasi-precedences. When using quasi-precedences it becomes important to use the bounded version of the lexicographic extension, since otherwise one would be able to construct an infinite sequence $f_0(1) \succ_{\mathsf{RoT}} f_1(0,1) \succ_{\mathsf{RoT}} f_2(0,0,1) \succ_{\mathsf{RoT}} \ldots$ by using a quasi-precendence where $f_i \geqslant f_j$ for all i, j and $f_i > 1 > 0$ for all i. The second refinement are comparisons of the form $x \succsim c$ in case **2.d**. For LPO one requires that c is least in precedence among all symbols, in the same way as in $\succ_{\mathsf{RoT}}$. By contrast, for KBO $f \geqslant c$ is only required for those f which are constants and have weight w_0.

Note that activating both requirements – quasi-precedences and $x \succsim c$ comparisons – is sound for LPO, requires a special definition of lexicographic extensions for KBO, and is unsound for RPO.

▶ **Example 1.** Consider RPO with both refinements, i.e., $(\succ, \succsim)$ is the trivial reduction pair. Let $\geqslant = \mathcal{F} \times \mathcal{F}$ be the trivial precedence where all symbols are equivalent. Let $\tau(c) = \mathsf{lex}$ and $\tau(d) = \mathsf{mul}$ for two constants $c, d \in \mathcal{F}$. Then using case **2.d** we have $x \succsim_{\mathsf{RPO}} c$, but $d \succsim_{\mathsf{RPO}} c$ does not hold. Hence, closure under substitutions is violated.

▶ **Example 2.** Consider a KBO with precedence where all symbols are equivalent, a unary function symbol f with weight 0, and arbitrary symbols g_i with arity $i > 1$. Then $f(x) \succ_{\mathsf{KBO}} x$; however, for $f(g_i(t_1, \ldots, t_i)) \succ_{\mathsf{KBO}} g_i(t_1, \ldots, t_i)$ (closure under substitutions), only case **2.b.ii.B** is applicable, i.e., one needs lexicographic comparisons $[g_i(t_1, \ldots, t_i)] \succ_{\mathsf{KBO}}^{\mathsf{lex}} [t_1, \ldots, t_i]$ with lists of arbitrary lengths, i.e., unbounded lexicographic comparisons, which usually destroy well-foundedness in combination with unbounded arities.

The problem of Example 1 is easily fixed by just adding one more alternative to **2.b.ii**:

2. b. ii. C. $f \succsim g$ and $\tau(f) \neq \tau(g)$ and $m = 0$ (and $n > 0$ for $s \succ_{\mathsf{RoT}} t$)

That $(\succ_{\mathsf{RoT}}, \succsim_{\mathsf{RoT}})$ really forms a reduction pair with this fix has been formally proven. Actually, we have formalized an extended version of $\succ_{\mathsf{RoT}}$ that also includes the other features of WPO, i.e., a status function $\pi : \mathcal{F} \to \mathbb{N}^*$ and Refinements (2c) and (2d) of WPO [6, Section 4.2], and it is available in the archive of formal proofs [4]. It is the same definition as if one would take the WPO definition of [6], add multiset comparisons via a status $\tau : \mathcal{F} \to \{\mathsf{lex}, \mathsf{mul}\}$, and add case **2.b.ii.C** for symbols with different status.

▶ **Theorem 3.** $(\succ_{\mathsf{RoT}}, \succsim_{\mathsf{RoT}})$ *is a reduction pair and* $\succ_{\mathsf{RoT}}$ *is a simplification order.*

4 Simulating Classical Simplification Orders

In the previous section we have already seen that LPO and RPO are just instances of the WPO (assuming a definition of WPO that includes the status function τ). This covers quasi-precedences and the $x \succsim c$ refinement. However, such a relationship is not yet established for KBO with refinements. In particular there are three major differences:

1. minimal constants in KBO are defined differently than in WPO,
2. there is a different syntactic structure, and
3. WPO uses the bounded lexicographic extension, but KBO uses the unbounded extension.

We will address these problems and show how properties of WPO can be transferred to KBO.

1. Recall that in KBO a constant c is minimal if $f \geqslant c$ for all constants f of weight w_0, whereas in WPO $f \geqslant c$ is required for all $f \in \mathcal{F}$. We solve this problem by changing the quasi-precedence $\geqslant$ of KBO into some quasi-precedence $\geqslant'$ in a way that
   - KBO-minimal constants w.r.t. $\geqslant$ are WPO-minimal w.r.t. $\geqslant'$, and
   - $\succsim_{\mathsf{KBO}}$ and $\succ_{\mathsf{KBO}}$ are unmodified when switching from $\geqslant$ to $\geqslant'$.
2. For the syntactic differences, we prove that they do not affect the defined relations.
   - The additional case $f(C[x]) \succ_{\mathsf{KBO}} x$ of KBO can be simulated since $\succ_{\mathsf{RoT}}$ is a simplification order.
   - Assume that $f(s_1, \ldots, s_n) \succ_{\mathsf{KBO}} f(t_1, \ldots, t_m)$ was shown by **2.b**. Here we use some properties of KBO to conclude $f(s_1, \ldots, s_n) \succ_{\mathsf{KBO}} t_j$ for all $1 \leq j \leq m$. Hence, it does not matter whether the condition in **2.b.i** – which does not occur in the original KBO definition – is added to the KBO definition.

- The definition of KBO does not contain case **2.a**. However, as in the previous step we utilize the property of KBO that the corresponding inference rule $s_i \succsim_{\mathsf{KBO}} t \longrightarrow f(s_1, \ldots, s_n) \succ_{\mathsf{KBO}} t$ is still valid for all $i \in \{1, \ldots, n\}$.

Note that for the equivalence proof we already use some properties of KBO, i.e., these must be proven before we are able to transfer properties of $\succ_{\mathsf{RoT}}$ to KBO.

3. One cannot replace the unbounded lexicographic extension by a bounded one if function symbols of unbounded arity are considered. However, whenever terms s and t are compared, only finitely many symbols appear in s and t, and thus there is the maximum arity b among them. For these terms there is no difference in whether b-bounded or unbounded lexicographic extension is used.

We arrive at the following result.

▶ **Theorem 4.** *Let a KBO with quasi-precedence $\geqslant$ and some bound b be given. Then a reduction pair (encoding the weight-function) and quasi-precedence $\geqslant'$ can be constructed as parameters to $\succ_{\mathsf{RoT}}$ and $\succsim_{\mathsf{RoT}}$ (or to WPO), such that $(s \succ_{\mathsf{KBO}} t) \longleftrightarrow (s \succ_{\mathsf{RoT}} t)$ and $(s \succsim_{\mathsf{KBO}} t) \longleftrightarrow (s \succsim_{\mathsf{RoT}} t)$ for all terms s, t whose function symbols have arity below b.*

▶ **Corollary 5.** *For every KBO over a finite signature there exists an equivalent WPO.*

▶ **Corollary 6.** *KBO is transitive, closed under substitutions and well-founded.*

Proof. Consider the set of terms $\{s, t, u, s\sigma, t\sigma\}$, and define b as the maximum arity that occurs within these terms. From Theorem 3 we conclude $s \succ_{\mathsf{RoT}} t \succ_{\mathsf{RoT}} u \longrightarrow s \succ_{\mathsf{RoT}} u$ and $s \succ_{\mathsf{RoT}} t \longrightarrow s\sigma \succ_{\mathsf{RoT}} t\sigma$. By Theorem 4 and the choice of b, transitivity and closure under substitutions of KBO are proved.

For well-foundedness of KBO, consider an infinite sequence $t_1 \succ_{\mathsf{KBO}} t_2 \succ_{\mathsf{KBO}} \ldots$. Define b' as the weight of t_1. Hence b' is larger than the weight of all terms in the sequence. Since the weight is an upper bound for the arities, b' is also larger than the arities of all t_i. Thus, by Theorem 4 we know $t_1 \succ_{\mathsf{RoT}} t_2 \succ_{\mathsf{RoT}} \ldots$ in contradiction to Theorem 3. ◀

As future work it remains to be clarified whether the addition of multiset comparisons to WPO will improve the power of automated termination tools.

References

1 Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998. `doi:10.1017/CBO9781139172752`.

2 Nachum Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982. `doi:10.1016/0304-3975(82)90026-3`.

3 Donald E. Knuth and Peter Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, New York, 1970. `doi:10.1016/B978-0-08-012975-4.50028-X`.

4 Christian Sternagel, René Thiemann, and Akihisa Yamada. A formalization of weighted path orders and recursive path orders. *Archive of Formal Proofs*, 2021. `https://isa-afp.org/entries/Weighted_Path_Order.html`, Formal proof development.

5 René Thiemann and Christian Sternagel. Certification of termination proofs using CeTA. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468. Springer, 2009. `doi:10.1007/978-3-642-03359-9_31`.

6 Akihisa Yamada, Keiichirou Kusakari, and Toshiki Sakabe. A unified ordering for termination proving. *Sci. Comput. Program.*, 111:110–134, 2015. `doi:10.1016/j.scico.2014.07.009`.

Tuple Interpretations and Applications to Higher-Order Runtime Complexity

Cynthia Kop
Institute for Computation and Information Sciences, Radboud University, The Netherlands

Deivid Vale
Institute for Computation and Information Sciences, Radboud University, The Netherlands

Abstract

Tuple interpretations are a class of algebraic interpretation that subsumes both polynomial and matrix interpretations as it does not impose simple termination and allows non-linear interpretations. It was developed in the context of higher-order rewriting to study derivational complexity of algebraic functional systems. In this short paper, we continue our journey to study the complexity of higher-order TRSs by tailoring tuple interpretations to deal with innermost runtime complexity.

2012 ACM Subject Classification Theory of computation → Equational logic and rewriting

Keywords and phrases Complexity analysis, higher-order term rewriting, tuple interpretations

Funding The authors are supported by the NWO TOP project "ICHOR", NWO 612.001.803/7571 and the NWO VIDI project "CHORPE", NWO VI.Vidi.193.075.

1 Introduction

The step-by-step computational model induced by term rewriting naturally gives rise to a *complexity* notion. Here, complexity is understood as the number of rewriting steps needed to reach a normal form. In the rewriting setting, a *complexity function* bounds the length of the longest rewrite sequence parametrized by the size of the starting term. Two distinct complexity notions are commonly considered: derivational and runtime. In the former, the starting term is unrestricted which allows initial terms with nested function calls. The latter only considers rewriting sequences beginning with *basic* terms. Intuitively, basic terms are those where a single function call is performed with *data* objects as arguments.

There are many techniques to bound the runtime complexity of term rewriting [2, 4]. However, most of the literature focuses on the first-order case. We take a different approach and regard higher-order term rewriting. We present a technique that takes advantage of tuple interpretations [3] tailored to deal with an innermost rewriting strategy. The defining characteristic of tuple interpretations is to allow for a split of the complexity measure into abstract notions of *cost* and *size*. The former is usually interpreted as natural numbers, which accounts for the number of steps needed to reduce terms to normal forms. Meanwhile, the latter is interpreted as tuples over naturals carrying abstract notions of size.

2 Preliminaries

The Syntax of Terms and Rules We assume familiarity with the basics of term rewriting. We will here recall notation for *applicative simply-typed term rewriting systems*.

Let $\mathcal{B}$ be a set of base types (or sorts). The set $\mathcal{T}_\mathcal{B}$ of *simple types* is built using the right-associative $\Rightarrow$ as follows. Every $\iota \in \mathcal{B}$ is a type of order 0. If σ, τ are types of order n and m respectively, then $\sigma \Rightarrow \tau$ is a type of order $\max(n+1, m)$. A signature is a non-empty set $\mathcal{F}$ of function symbols together with a function $\mathtt{typeOf} : \mathcal{F} \longrightarrow \mathcal{T}_\mathcal{B}$. Additionally, we assume, for each $\sigma \in \mathcal{T}_\mathcal{B}$, a countable infinite set of type-annotated variables $\mathcal{X}_\sigma$ disjoint from $\mathcal{F}$. We will denote $\mathtt{f}, \mathtt{g}, \ldots$ for function symbols and $x, y, \ldots$ for variables.

This typing scheme imposes a restriction on the formation of terms which consists of those expressions s such that $s :: \sigma$ can be derived for some type σ using the following clauses: (i) $x :: \sigma$, if $x \in \mathcal{X}_\sigma$; (ii) $\mathsf{f} :: \sigma$, if $\mathtt{typeOf}(\mathsf{f}) = \sigma$; and (iii) $(s\,t) :: \tau$, if $s :: \sigma \Rightarrow \tau$ and $t :: \tau$. Application is left-associative. We denote $\mathtt{vars}(s)$ for the set of variables occurring in s and say s is *ground* if $\mathtt{vars}(s) = \emptyset$. A rewriting rule $\ell \to r$ is a pair of terms of the same type such that $\ell = \mathsf{f}\,\ell_1 \ldots \ell_m$ and $\mathtt{vars}(\ell) \supseteq \mathtt{vars}(r)$. An *applicative simply-typed term rewriting system* (shortly denoted TRS), is a set $\mathcal{R}$ of rules. The rewrite relation induced by $\mathcal{R}$ is the smallest monotonic relation that contains $\mathcal{R}$ and is stable under application of substitution. A term s is in *normal form* if there is no t such that $s \to t$. The *innermost rewrite relation* induced by $\mathcal{R}$ is defined as follows:

- $\ell\gamma \to^i r\gamma$, if $\ell \to r \in \mathcal{R}$ and all proper subterms of $\ell\gamma$ are in $\mathcal{R}$-normal form;
- $s\,t \to^i s'\,t$, if $s \to^i s'$; and $s\,t \to^i s\,t'$, if $t \to^i t'$.

In what follows we only allow for innermost reductions. So, we drop the i from the arrow, and $s \to t$ is to be read as $s \to^i t$. We shall use the explicit notation if confusion may arise.

▶ **Example 1.** We will use a system over the sorts nat (numbers) and list (lists of numbers). Let $\mathsf{0} :: \mathsf{nat}$, $\mathsf{s} :: \mathsf{nat} \Rightarrow \mathsf{nat}$, $\mathsf{nil} :: \mathsf{list}$, $\mathsf{cons} :: \mathsf{nat} \Rightarrow \mathsf{list} \Rightarrow \mathsf{list}$, and $F, G \in \mathcal{X}_{\mathsf{nat} \Rightarrow \mathsf{nat}}$; types of other function symbols and variables can be easily deduced.

$$
\begin{array}{rclcrcl}
\mathtt{map}\,F\,\mathsf{nil} & \to & \mathsf{nil} & \qquad & \mathtt{comp}\,F\,G\,x & \to & F\,(G\,x) \\
\mathtt{map}\,F\,(\mathsf{cons}\,x\,xs) & \to & \mathsf{cons}\,(F\,x)\,(\mathtt{map}\,F\,xs) & & \mathtt{app}\,F\,x & \to & F\,x \\
\mathtt{d}\,\mathsf{0} & \to & \mathsf{0} & & \mathtt{add}\,x\,\mathsf{0} & \to & x \\
\mathtt{d}\,(\mathsf{s}\,x) & \to & \mathsf{s}\,(\mathsf{s}\,(\mathtt{d}\,x)) & & \mathtt{add}\,x\,(\mathsf{s}\,y) & \to & \mathsf{s}\,(\mathtt{add}\,x\,y)
\end{array}
$$

Functions and orderings A quasi-ordered set $(A, \sqsupseteq)$ consists of a nonempty set A and a quasi-order $\sqsupseteq$ over A. A well-founded set $(A, >, \geq)$ is a nonempty set A together with a well-founded order $>$ and a compatible quasi-order $\geq$ on A, i.e., $> \circ \geq \subseteq >$. For quasi-ordered sets A and B, we say that a function $f : A \longrightarrow B$ is weakly monotonic if for all $x, y \in A$, $x \sqsupseteq_A y$ implies $f(x) \sqsupseteq_B f(y)$. If $(B, >, \geq)$ is a well-founded set, then $>$ and $\geq$ induce a point-wise comparison on $A \longrightarrow B$ as usual. If A, B are quasi-ordered, the notation $A \Longrightarrow B$ refers to the set of all weakly monotonic functions from A to B. Functional equality is extensional. The unit set is the quasi-ordered set defined by $\mathtt{unit} = (\{\mathtt{u}\}, \sqsupseteq)$, where $\mathtt{u} \sqsupseteq \mathtt{u}$.

3 Higher-Order Tuple Interpretations for Innermost Rewriting

To define interpretations, we will start by providing an interpretation of *types* (Def. 2). Types σ are interpreted by tuples $(\!|\sigma|\!)$ that carry information about cost and size. We will first show how application works in this newly defined cost-size domain (Def. 4). Interpretation of types will then set the domain for the tuple algebras we are interested in (Def. 7).

▶ **Definition 2** (Interpretation of Types). *For each type σ, we define the cost-size tuple interpretation of σ as $(\!|\sigma|\!) = \mathcal{C}_\sigma \times \mathcal{S}_\sigma$ where $\mathcal{C}_\sigma$ (respectively $\mathcal{S}_\sigma$) is defined as follows:*

$$
\begin{array}{llll}
\mathcal{C}_\sigma = \mathbb{N} \times \mathcal{F}^{\mathsf{c}}_\sigma & \qquad & \mathcal{S}_\iota = (\mathbb{N}^{K[\iota]}, \sqsupseteq), \ K[\iota] \geq 1 \\
\mathcal{F}^{\mathsf{c}}_\iota = \mathtt{unit} & & \mathcal{S}_{\sigma \Rightarrow \tau} = \mathcal{S}_\sigma \Longrightarrow \mathcal{S}_\tau \\
\mathcal{F}^{\mathsf{c}}_{\sigma \Rightarrow \tau} = (\mathcal{F}^{\mathsf{c}}_\sigma \times \mathcal{S}_\sigma) \Longrightarrow \mathcal{C}_\tau, & &
\end{array}
$$

where $\mathcal{F}^{\mathsf{c}}_{\sigma \Rightarrow \tau}$ ($\mathcal{S}_{\sigma \Rightarrow \tau}$) is the set of weakly monotonic functions from $\mathcal{F}^{\mathsf{c}}_\sigma \times \mathcal{S}_\sigma$ to $\mathcal{C}_\tau$ ($\mathcal{S}_\sigma$ to $\mathcal{S}_\tau$). The quasi-ordering on those sets is the induced point-wise comparison. The set $(\!|\sigma|\!)$ is ordered as follows: $((n, f), s) \succ ((m, g), t)$ if $n > m$, $f \geq g$ and $s \sqsupseteq t$; and $((n, f), s) \succcurlyeq ((m, g), t)$ if $n \geq m$, $f \geq g$ and $s \sqsupseteq t$.

The cost tuple $\mathcal{C}_\sigma = \mathbb{N} \times \mathcal{F}^{\mathsf{c}}_\sigma$ of $(\!|\sigma|\!)$ holds the cost information of reducing a term of type σ to its normal form. It is composed of a numeric and functional component. Base types, which are naturally not functional, have the unit set for $\mathcal{F}^{\mathsf{c}}_\iota$; the cost tuple of a base type is then $\mathcal{C}_\iota = \mathbb{N} \times \mathtt{unit}$. Functional types do possess an intrinsically functional component (the cost of *applying* the function), which in our setting is expressed by $\mathcal{F}^{\mathsf{c}}_{\sigma \Rightarrow \tau} = \mathcal{F}^{\mathsf{c}}_\sigma \times \mathcal{S}_\sigma \Longrightarrow \mathcal{C}_\tau$. For functional types the numeric component represents the cost of partial application.

To determine the number $K[\iota]$, associated to each sort ι, we use a semantic approach that takes the intuitive meaning of the sort we are interpreting into account. The sort nat for instance represents natural numbers, which we implement in unary format. Hence, it makes sense to reckon the number of successor symbols occurring in terms of the form $(\mathsf{s}^n\,\mathsf{0}) :: \mathsf{nat}$ as their *size*. This gives us $K[\mathsf{nat}] = 1$. Another example is the sort list (of natural numbers): it is natural to regard measures like *length* and *maximum element size*. This results in $K[\mathsf{list}] = 2$. Example 8 below shows how to interpret data constructors using this intuition.

The next lemma expresses the soundness of our approach, that is, cost-size tuples define a well-founded domain for the interpretation of types.

▶ **Lemma 3.** *For each type σ, the set $\mathcal{C}_\sigma$ is well-founded and $\mathcal{S}_\sigma$ quasi-ordered. Their product, that is, $((\!|\sigma|\!), \succ, \succcurlyeq)$, is well-founded.*

Semantic Application To interpret each term $s :: \sigma$ to an element of $(\!|\sigma|\!)$ (Def. 7), we will need a notion of application for cost-size tuples. Specifically, given a functional type $\sigma \Rightarrow \tau$, a cost-size tuple $\boldsymbol{f} \in (\!|\sigma \Rightarrow \tau|\!)$, and $\boldsymbol{x} \in (\!|\sigma|\!)$, our goal is to define the application $\boldsymbol{f} \cdot \boldsymbol{x}$ of $\boldsymbol{f}$ to $\boldsymbol{x}$. Let us illustrate the idea with a concrete example: consider the type $\sigma = (\mathsf{nat} \Rightarrow \mathsf{nat}) \Rightarrow \mathsf{list} \Rightarrow \mathsf{list}$, which is the type of $\mathtt{map}$ defined in Example 1. The function $\mathtt{map}$ takes as argument a function F of type $\mathsf{nat} \Rightarrow \mathsf{nat}$ and a list q, and applies F to each element of q. The cost interpretation of $\mathtt{map}$ is a functional in $\mathcal{C}_\sigma$ parametrized by functional arguments carrying the cost and size information of F and a cost-size tuple for q.

$$\mathbb{N} \times \overbrace{((\underbrace{(\mathtt{unit} \times \mathbb{N} \Longrightarrow \mathbb{N} \times \mathtt{unit})}_{\text{cost of } F} \times \underbrace{(\mathbb{N} \Longrightarrow \mathbb{N})}_{\text{size of } F} \Longrightarrow (\mathbb{N} \times (\underbrace{\mathtt{unit}}_{\text{cost of } q} \times \underbrace{\mathbb{N}^2}_{\text{size of } q} \Longrightarrow \mathbb{N} \times \mathtt{unit})))}^{\text{the functional cost of } \mathtt{map}},$$

Hence, we write an element of such space as the tuple (n, f^{c}). Size sets are somewhat simpler with $\underbrace{(\mathbb{N} \Longrightarrow \mathbb{N})}_{\text{size of } F} \Longrightarrow \underbrace{\mathbb{N}^2}_{\text{size of } q} \Longrightarrow \mathbb{N}^2$. Therefore, a functional cost-size tuple $\boldsymbol{f}$ is represented by $\boldsymbol{f} = \langle (n, f^{\mathsf{c}}), f^{\mathsf{s}} \rangle$. An argument to such a cost-size tuple is then an element in the domain of f^{c} and f^{s}, respectively. Therefore, we apply $\boldsymbol{f}$ to a cost-size tuple $\boldsymbol{x}$ of the form $\langle (m, g^{\mathsf{c}}), g^{\mathsf{s}} \rangle$ where g^{c} is the cost of computing F and g^{s} is the size of F. We proceed by applying the respective functions, so $f^{\mathsf{c}}(g^{\mathsf{c}}, g^{\mathsf{s}}) = (k, h)$ belongs to $\mathcal{C}_{\mathsf{list}}$, and add the numeric components together obtaining: $\boldsymbol{f} \cdot \boldsymbol{x} = \langle (n + m + k, f^{\mathsf{c}}(g^{\mathsf{c}}, g^{\mathsf{s}})), f^{\mathsf{s}}(g^{\mathsf{s}}) \rangle$. Notice that this gives us a new cost-size tuple with cost component in $\mathbb{N} \times (\mathcal{C}_{\mathsf{list}} \Longrightarrow \mathcal{C}_{\mathsf{list}})$ and size component in $\mathcal{S}_{\mathsf{list}} \Longrightarrow \mathcal{S}_{\mathsf{list}}$.

▶ **Definition 4.** *Let $\sigma \Rightarrow \tau$ be an arrow type, $\boldsymbol{f} = \langle (n, f^{\mathsf{c}}), f^{\mathsf{s}} \rangle \in (\!|\sigma \Rightarrow \tau|\!)$, and $\boldsymbol{x} = \langle (m, g^{\mathsf{c}}), g^{\mathsf{s}} \rangle \in (\!|\sigma|\!)$. The application of $\boldsymbol{f}$ to $\boldsymbol{x}$, denoted $\boldsymbol{f} \cdot \boldsymbol{x}$, is defined by:*

$$\text{let } f^{\mathsf{c}}(g^{\mathsf{c}}, g^{\mathsf{s}}) = (k, h)\text{; then } \langle (n, f^{\mathsf{c}}), f^{\mathsf{s}} \rangle \cdot \langle (m, g^{\mathsf{c}}), g^{\mathsf{s}} \rangle = \langle (n + m + k, h), f^{\mathsf{s}}(g^{\mathsf{s}}) \rangle$$

Semantic application is left-associative and respects a form of application rule.

▶ **Lemma 5.** *If $\boldsymbol{f}$ is in $(\!|\sigma \Rightarrow \tau|\!)$ and $\boldsymbol{x}$ is in $(\!|\sigma|\!)$, then $\boldsymbol{f} \cdot \boldsymbol{x}$ belongs to $(\!|\tau|\!)$.*

▶ Remark 6. In order to ease notation, we project sets $\pi_1 : A \times \texttt{unit} \longrightarrow A$ and $\pi_2 : \texttt{unit} \times A \longrightarrow A$ and compose functions with projections, so a function in $\texttt{unit} \times A \Longrightarrow B \times \texttt{unit}$ is lifted to a function in $A \Longrightarrow B$. The functional cost of map is then read as follows:

$$\mathbb{N} \times \overbrace{(\underbrace{(\mathbb{N} \Longrightarrow \mathbb{N})}_{\text{cost of } F} \times \underbrace{(\mathbb{N} \Longrightarrow \mathbb{N})}_{\text{size of } F} \Longrightarrow (\mathbb{N} \times (\underbrace{\mathbb{N}^2}_{\text{size of } q} \Longrightarrow \mathbb{N})))}^{\text{the functional cost of } \texttt{map}}$$

The $\mathbb{N}$ component of $C_{\sigma \Rightarrow \tau}$ is specific to innermost rewriting (it does not occur in [3]). We need this to handle rules of non-base type; for example, if $\texttt{add}\,0 \to \texttt{id}$, then the cost tuple of $\texttt{add}\ 0$ is $(1, \boldsymbol{\lambda} x.0)$. However, since in *most* cases the first component is 0, we will typically omit these zeroes and simply write for instance $\boldsymbol{\lambda} Fq.f^{\mathsf{c}}(F,q)$ instead of $(0, \boldsymbol{\lambda} F.\langle 0, \boldsymbol{\lambda} q.f^{\mathsf{c}}(F,q)\rangle)$. To compute using Definition 4 we still use the complete form.

Tuple algebras are higher-order weakly monotonic algebras [1] with cost-size tuples as interpretation domain.

▶ **Definition 7** (Higher-order tuple algebra). *A higher-order tuple algebra over a signature $(\mathcal{B}, \mathcal{F}, \texttt{typeOf})$ consists of: (i) a family of cost/size tuples $\{(\!|\sigma|\!)\}_{\sigma \in \mathcal{T}_{\mathcal{B}}}$ and (ii) an interpretation function $\mathcal{J}$ which maps each $\mathsf{f} \in \mathcal{F}$ of type σ to a cost-size tuple in $(\!|\sigma|\!)$.*

▶ **Example 8.** Following the semantics discussed previously, we interpret the constructors for both nat and list as follows. We call the first component of $\mathcal{S}_{\mathsf{list}}$ *length* and the second *maximum element size*. Those are abbreviated using the letters l and m, respectively.

$$\mathcal{J}_{0} = \langle 0, 0 \rangle \qquad \mathcal{J}_{\mathsf{s}} = \langle \boldsymbol{\lambda} x.0, \boldsymbol{\lambda} x.x + 1 \rangle$$
$$\mathcal{J}_{\mathsf{nil}} = \langle 0, \langle 0, 0 \rangle \rangle \qquad \mathcal{J}_{\mathsf{cons}} = \langle \boldsymbol{\lambda} xq.0, \boldsymbol{\lambda} xq.\langle q_{\mathsf{l}} + 1, \max(x, q_{\mathsf{m}}) \rangle \rangle$$

The cost-size tuples for 0 and nil are all 0s, as expected. The size components for s and cons describe the increase in size when new data is created. We interpret functions from Example 1 as follows:

$$\begin{array}{rcl}
\mathcal{J}_{\mathtt{app}} & = & \langle \boldsymbol{\lambda} Fx.F^{\mathsf{c}}(x) + 1, \boldsymbol{\lambda} Fx.F^{\mathsf{s}}(x) \rangle \\
\mathcal{J}_{\mathtt{d}} & = & \langle \boldsymbol{\lambda} x.x + 1, \boldsymbol{\lambda} x.2x \rangle \\
\mathcal{J}_{\mathtt{add}} & = & \langle \boldsymbol{\lambda} xy.y + 1, \boldsymbol{\lambda} xy.x + y \rangle \\
\mathcal{J}_{\mathtt{comp}} & = & \langle \boldsymbol{\lambda} FGx.F^{\mathsf{c}}(G^{\mathsf{s}}(x_{\mathsf{s}})) + 1, \boldsymbol{\lambda} FGx.F^{\mathsf{s}}(G^{\mathsf{s}}(x)) \rangle \\
\mathcal{J}_{\mathtt{map}} & = & \langle \boldsymbol{\lambda} Fq.q_{\mathsf{l}} F^{\mathsf{c}}(q_{\mathsf{m}}) + 1, \boldsymbol{\lambda} Fq.\langle q_{\mathsf{l}}, F^{\mathsf{s}}(q_{\mathsf{m}}) \rangle \rangle
\end{array}$$

A valuation α is a function that maps each $x :: \sigma$ to a cost-size tuple in $(\!|\sigma|\!)$. Due to innermost strategy, we can assume the interpretation of every variable $x :: \iota$ has zero cost. This is formalized by assigning $\alpha(x) = \langle (0, \mathtt{u}), x^s \rangle$, for all $x \in \mathcal{X}$ of base type. In this paper, we shall only consider valuations that satisfy this property. Variables of functional type, however, may carry cost information even though any instance of a redex needs to be normalized. Hence, we set $\alpha(F) = \langle (0, f^{\mathsf{c}}), f^{\mathsf{s}} \rangle$ when $F :: \sigma \Rightarrow \tau$.

▶ **Definition 9.** *We extend $\mathcal{J}$ to an interpretation $[\![\cdot]\!]_{\alpha,\mathcal{J}}$ of terms as follows:*

$$[\![x]\!]_{\alpha,\mathcal{J}} = \alpha(x) \qquad [\![\mathsf{f}]\!]_{\alpha,\mathcal{J}} = \langle (n, \mathcal{J}^{\mathsf{c}}_{\mathsf{f}}), \mathcal{J}^{\mathsf{s}}_{\mathsf{f}} \rangle, n \in \mathbb{N} \qquad [\![s\,t]\!]_{\alpha,\mathcal{J}} = [\![s]\!]_{\alpha,\mathcal{J}} \cdot [\![t]\!]_{\alpha,\mathcal{J}}$$

We are interested in interpretations satisfying a compatibility requirement:

▶ **Theorem 10** (Innermost Compatibility Theorem). *Let α be a valuation. If $[\![\ell]\!]_{\alpha,\mathcal{J}} \succ [\![r]\!]_{\alpha,\mathcal{J}}$ for all rules $\ell \to r \in \mathcal{R}$, then $[\![s]\!]_{\alpha,\mathcal{J}} \succ [\![t]\!]_{\alpha,\mathcal{J}}$, whenever $s \to^{i}_{\mathcal{R}} t$.*

One can check that the TRS from Example 1 interpreted as in Example 8 satisfy the compatibility requirement.

4 Higher-Order Innermost Runtime Complexity

In this section, we briefly limn how the cost-size tuple machinery allow us to reason about innermost runtime complexity. We start by reviewing basic definitions.

▶ **Definition 11.** *A symbol* $\mathsf{f} \in \mathcal{F}$ *is a defined symbol if it occurs at the head of a rule, i.e., there is a rule* $\mathsf{f}\,\ell_1\,\ldots\,\ell_k \to r \in \mathcal{R}$. *A symbol* c *of order at most 1 is a data constructor if it is not a defined symbol. A data term has the form* $\mathsf{c}\,d_1\,\ldots\,d_k$ *with* c *a constructor and each* d_i *a data term. A term* s *is basic if* $s :: \iota$ *and* s *is of the form* $\mathsf{f}\,d_1\,\ldots\,d_m$ *with* f *a defined symbol and all* $d_1, \ldots, d_m$ *data terms. The set* $T_{\mathcal{B}}(\mathcal{F})$ *collects all basic terms.*

▶ Remark 12. Notice that our notion of data is intrinsically first-order. This is motivated by applications of rewriting to full program analysis where even if higher-order functions are used a program has type $\iota_1 \Rightarrow \ldots \Rightarrow \iota_m \Rightarrow \kappa$. The sorts ι_i are the input data types and κ the output type of the program.

▶ **Definition 13.** *The innermost derivation height of* s *is* $\mathtt{dh}_{\mathcal{R}}(s) = \{n \mid \exists t : s \to^n t\}$. *The innermost runtime complexity function with respect to a TRS* $\mathcal{R}$ *is* $\mathtt{irc}_{\mathcal{R}}(n) = \max\{\mathtt{dh}_{\mathcal{R}}(s) \mid s \in T_{\mathcal{B}}(\mathcal{F}) \wedge |s| \leq n\}$.

To reasonably bound the innermost runtime complexity of a TRS $\mathcal{R}$, we require that size interpretations of constructors have their components bounded by an additive polynomial, that is, a polynomial of the form $\boldsymbol{\lambda} x_1 \ldots x_k. \sum_{i=1}^{k} x_i + a$, with $a \in \mathbb{N}$.

We can build programs by adding a new $\mathtt{main}$ function taking data variables as arguments and combine it with rules computing functions, including higher-order ones. For instance, using rules from Example 1, we can compute a program that adds a number x to every element in a list q as follows: $\mathtt{main}\,x\,q \to \mathtt{map}\,(\mathtt{add}\,x)\,q$. Hence, computing this program on inputs n and list q is equivalent to reducing the term $\mathtt{main}\,\mathsf{n}\,q$ to normal form. Its runtime complexity is therefore bounded by the cost-tuple of $[\![\mathtt{main}\,\mathsf{n}\,q]\!]$.

5 Conclusion

In this short paper, we shed light on how to use cost-size tuple interpretations to bound innermost runtime complexity of higher-order systems. We defined a new domain of interpretations that takes the intricacies of innermost rewriting into account and defined how application works in this setting. The compatibility result allows us to make use of interpretations as a way to bound the length of derivation chains, as it is expected from an interpretation method. As current, and future work, we are working on automation techniques to find interpretations and develop a completely rewriting-based automated tool for complexity analysis of functional programs.

References

1 C. Fuhs and C. Kop. Polynomial interpretations for higher-order rewriting. In *Proc. RTA*, 2012. doi:10.4230/LIPIcs.RTA.2012.176.
2 N. Hirokawa and G. Moser. Automated complexity analysis based on the dependency pair method. In *Proc. IJCAR*, 2008. doi:10.1007/978-3-540-71070-7_32.
3 C. Kop and D. Vale. Tuple interpretations for higher-order complexity. In *Proc. FSCD*, 2021. doi:10.4230/LIPIcs.FSCD.2021.31.
4 L. Noschinski, F. Emmes, and J. Giesel. Analysing innermost runtime complexity of term rewriting by dependency pairs. *JAR*, 2013. doi:10.1007/s10817-013-9277-6.

A transitive HORPO for curried systems

Liye Guo
Radboud University Nijmegen, Netherlands

Cynthia Kop
Radboud University Nijmegen, Netherlands

Abstract

The higher-order recursive path ordering is one of the oldest, but still very effective, methods to prove termination of higher-order TRSs. A limitation of this ordering is that it is not transitive (and its transitive closure is not computable). We will present a transitive variation of HORPO. Unlike previous HORPO definitions, this method can be used directly on terms in curried notation.

2012 ACM Subject Classification Theory of computation → Equational logic and rewriting

Keywords and phrases Higher-order term rewriting, termination, recursive path ordering

Funding The authors are supported by the NWO VIDI project "CHORPE", NWO VI.Vidi.193.075.

1 Introduction

Termination problems have been studied by the term rewriting community for decades. In *higher-order* termination, one of the earliest techniques was HORPO [2], a higher-order extension of the *recursive path ordering* [3]. This definition has seen a series of improvements over the years, culminating in the powerful *Computability Path Ordering* (CPO) [1].

Interestingly, the relations $\succ_{\text{horpo}}$ and $\succ_{\text{cpo}}$ are not transitive. To obtain a well-founded ordering, we must use the transitive closure $\succ_{\text{horpo}}^{+}$ (resp. $\succ_{\text{cpo}}^{+}$), but this is not computable. Hence, for many rules that can in theory be oriented by HORPO, this cannot be found in practice. This limitation is particularly problematic when HORPO is transposed to formalisms where lambda abstractions occur more often on the left-hand side of a rule, since we may for instance have $\mathtt{f}(\lambda x.\mathtt{g}(x), Y) \succ_{\text{cpo}} (\lambda x.\mathtt{g}(x)) \cdot Y \succ_{\text{cpo}} \mathtt{g}(Y)$, but not $\mathtt{f}(\lambda x.\mathtt{g}(x), Y) \succ_{\text{cpo}} \mathtt{g}(Y)$.

To address this issue, the second author explored an alternative HORPO in her PhD thesis [5]: following an idea from the *iterative* path ordering [4], we use an annotation $\star$ to mark an obligation to decrease a term. This can be harnessed to obtain a transitive definition. Like HORPO and CPO, *StarHorpo* was defined on a formalism with functional (uncurried) notation; application is encoded as a family of function symbols. Consequently, in curried specifications, the same few symbols occur over and over, making the method hard to apply.

In this paper, we adapt StarHorpo to a *curried* system. This is not just a notational matter: allowing function symbols to take a variable number of arguments poses new technical challenges. This is work in progress; we will focus on the core aspects of the method. For now, we omit lambda abstractions and type orderings as used in CPO. However, the eventual goal is to define a transitive ordering that strictly includes CPO for curried systems.

2 Preliminaries

2.1 Applicative TRS

For presentation, we shall consider an applicative term rewriting system. We assume that a set $\mathcal{S}$ of *base types* is given, and the set $\mathcal{T}$ of *simple types* is generated by the grammar $\mathcal{T} ::= \mathcal{S} \mid (\mathcal{T} \to \mathcal{T})$. Right-associativity is assigned to $\to$ so that some parentheses in types can be omitted. We postulate two disjoint sets $\mathcal{F}$ and $\mathcal{V}$, called the set of *function symbols* and the set of *variables*, respectively. We assume that every function symbol and variable has exactly one simple type, and we write $\mathfrak{a} : A$ for $\mathfrak{a}$ of type A. In this paper, we let f and g range over the set $\mathcal{F}$, x over the set $\mathcal{V}$ and $\mathfrak{a}$ over $\mathcal{F} \cup \mathcal{V}$.

The set $\mathbb{T}$ of *pre-terms* is generated by the grammar $\mathbb{T} ::= \mathcal{F}(\mathbb{T}, \ldots, \mathbb{T}) \mid \mathcal{V}(\mathbb{T}, \ldots, \mathbb{T})$. The set of *terms* consists of pre-terms which can be given a simple type by the following rule:

$$\frac{\mathfrak{a} : A_1 \to \cdots \to A_n \to B \qquad t_1 : A_1 \qquad \ldots \qquad t_n : A_n}{\mathfrak{a}(t_1, \ldots, t_n) : B} \ (\mathfrak{a} \in \mathcal{F} \cup \mathcal{V})$$

A term has only one type. When $n = 0$, we omit the parentheses and write $\mathfrak{a}$ instead of $\mathfrak{a}()$.

The *application* of a term $t = \mathfrak{a}(t_1, \ldots, t_n) : A \to B$ to another term $t_{n+1} : A$, denoted by $t \cdot t_{n+1}$, is defined to be $\mathfrak{a}(t_1, \ldots, t_n, t_{n+1})$. We assign to $\cdot$ left-associativity. Type-preserving functions from variables to terms are called *substitutions*. Every substitution σ extends to a type-preserving function $\bar{\sigma}$ from terms to terms. We write $t\sigma$ for $\bar{\sigma}(t)$ and define it as follows: $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$, and $x(t_1, \ldots, t_n)\sigma = \sigma(x) \cdot t_1\sigma \cdots t_n\sigma$.

A *rewrite rule* $\ell \to r$ is an ordered pair where ℓ and r are terms of the same type, variables occurring in r also occur in ℓ, and $\ell = f(t_1, \ldots, t_n)$. Given a set $\mathcal{R}$ of rewrite rules, $t \to_{\mathcal{R}} t'$ if and only if one of the following conditions is true:

- $t = \ell\sigma$, $t' = r\sigma$ and $\ell \to r \in \mathcal{R}$ for some substitution σ.
- $t = t_1 \cdot t_2$, $t' = t_1{}' \cdot t_2$ and $t_1 \to_{\mathcal{R}} t_1{}'$.
- $t = t_1 \cdot t_2$, $t' = t_1 \cdot t_2{}'$ and $t_2 \to_{\mathcal{R}} t_2{}'$.

$\to_{\mathcal{R}}$ is called the *rewrite relation*. This paper concerns the well-foundedness of $\to_{\mathcal{R}}$.

In the above definition, the application operator $\cdot$ is distinct from the function symbols, and terms are lists headed by a function symbol or a variable. An equivalent and commonly used alternative is to consider $\cdot$ as part of term formation and terms as binary trees. In this view, we would for instance write $f \cdot t_1 \cdot t_2$, or just $f\ t_1\ t_2$, instead of $f(t_1, t_2)$. We favor our current presentation to stress the similarities to the existing recursive path orderings, which are typically defined on formalisms with functional notation. We do *not* consider application as function symbols, as this would be detrimental to our method.

2.2 HORPO

We review a simple *higher-order recursive path ordering* [2] reformulated for the above formalism. Given a well-founded ordering $\blacktriangleright$ on $\mathcal{F}$, called the *precedence*, $s \succ_{\text{horpo}} t$ if and only if s has the same type as t and one of the following conditions is true:

(1) $s = f(s_1, \ldots, s_m)$ and $\exists i\ s_i \succeq_{\text{horpo}} t$.
(2) $s = f(s_1, \ldots, s_m)$, $t = t_1 \cdot t_2 \cdots t_n$ and $f(s_1, \ldots, s_m) \vdash \{t_1, \ldots, t_n\}$.
(3) $s = f(s_1, \ldots, s_m)$, $t = g(t_1, \ldots, t_n)$, $f \blacktriangleright g$ and $f(s_1, \ldots, s_m) \vdash \{t_1, \ldots, t_n\}$.
(4) $s = f(s_1, \ldots, s_m)$, $t = f(t_1, \ldots, t_m)$, $(s_1, \ldots, s_m) \succ_{\text{horpo}}^{\text{lex}} (t_1, \ldots, t_m)$ and $f(s_1, \ldots, s_m) \vdash \{t_1, \ldots, t_m\}$.
(5) $s = s_1 \cdot s_2$, $t = t_1 \cdot t_2$, $s_1 \succeq_{\text{horpo}} t_1$, $s_2 \succeq_{\text{horpo}} t_2$ and $s \neq t$.

In the above definition, $\succeq_{\text{horpo}}$ is the reflexive closure of $\succ_{\text{horpo}}$, $\succ_{\text{horpo}}^{\text{lex}}$ lexicographically compares lists of the same length by $\succ_{\text{horpo}}$, and $f(s_1, \ldots, s_m) \vdash \{t_1, \ldots, t_n\}$ stands for $\forall i\ (f(s_1, \ldots, s_m) \succ_{\text{horpo}} t_i \vee \exists j\ s_j \succeq_{\text{horpo}} t_i)$. We remark that the multiset extension in the definition [2] is omitted for simplicity's sake. The relation $\succ_{\text{horpo}}$ is *well-founded*, *monotonic* (i.e., $t_1 \succ_{\text{horpo}} t_1{}'$ implies $t_1 \cdot t_2 \succ_{\text{horpo}} t_1{}' \cdot t_2$, and $t_2 \succ_{\text{horpo}} t_2{}'$ implies $t_1 \cdot t_2 \succ_{\text{horpo}} t_1 \cdot t_2{}'$), and *stable* (i.e., $t \succ_{\text{horpo}} t'$ implies $t\sigma \succ_{\text{horpo}} t'\sigma$ for all substitutions σ). If in addition $\to_{\mathcal{R}}$ is *compatible* with $\succ_{\text{horpo}}$ (i.e., $\ell \succ_{\text{horpo}} r$ for all $\ell \to r \in \mathcal{R}$), then $\to_{\mathcal{R}}$ is well-founded.

As an example, consider the following definition of a recursor for the natural numbers:

$$\mathtt{rec}(\mathtt{0}, Y, F) \to Y \qquad \mathtt{rec}(\mathtt{s}(X), Y, F) \to F(X, \mathtt{rec}(X, Y, F))$$

where $\mathtt{0} : \mathtt{nat}$, $\mathtt{s} : \mathtt{nat} \to \mathtt{nat}$ and $\mathtt{rec} : \mathtt{nat} \to \mathtt{nat} \to (\mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}$ are the function symbols, and the types of the variables X, Y and F can be deduced. In order to show the well-foundedness of $\to_{\mathcal{R}}$, we need only to find a precedence $\blacktriangleright$ making $\to_{\mathcal{R}}$ compatible with the generated relation $\succ_{\text{horpo}}$. While $\mathtt{rec}(\mathtt{0}, Y, F) \succ_{\text{horpo}} Y$ follows from the first condition, $\mathtt{rec}(\mathtt{s}(X), Y, F) \succ_{\text{horpo}} F(X, \mathtt{rec}(X, Y, F))$ can be obtained as follows:

$$\dfrac{F \succeq_{\text{horpo}} F \qquad \dfrac{X \succeq_{\text{horpo}} X}{\mathtt{s}(X) \succeq_{\text{horpo}} X}\,1 \qquad \dfrac{\mathtt{s}(X) \succ_{\text{horpo}} X \qquad Y \succeq_{\text{horpo}} Y \qquad F \succeq_{\text{horpo}} F}{\mathtt{rec}(\mathtt{s}(X), Y, F) \succ_{\text{horpo}} \mathtt{rec}(X, Y, F)}\,4}{\mathtt{rec}(\mathtt{s}(X), Y, F) \succ_{\text{horpo}} F(X, \mathtt{rec}(X, Y, F))}\,2$$

The precedence can be any well-founded ordering on $\mathcal{F}$. The above process of finding the precedence can be automated by encoding the constraints $\ell \succ_{\text{horpo}} r$ in a propositional formula that is fed to a SAT solver, as demonstrated in [8] for the first-order RPO.

The usefulness of $\succ_{\text{horpo}}$ is limited by the type restriction—only terms of the same type can be compared. Let us extend the above example with the following rewrite rules:

$$\mathtt{add}(\mathtt{0}, Y) \to Y \qquad \mathtt{add}(\mathtt{s}(X), Y) \to \mathtt{s}(\mathtt{add}(X, Y)) \qquad \mathtt{sum}(X) \to \mathtt{rec}(X, \mathtt{0}, \mathtt{add})$$

where $\mathtt{add} : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$ and $\mathtt{sum} : \mathtt{nat} \to \mathtt{nat}$. If we ignore the rewrite rule on the right, we need only $\mathtt{add} \blacktriangleright \mathtt{s}$ to complete the proof. Since the rule on the right only removes occurrences of $\mathtt{sum}$, it seems harmless. However, $\mathtt{sum}(X) \succ_{\text{horpo}} \mathtt{rec}(X, \mathtt{0}, \mathtt{add})$ is not obtainable due to the type restriction: neither $\mathtt{sum}(X)$ nor X has the same type as $\mathtt{add}$ so the necessary premise $\mathtt{sum}(X) \Vdash \{\mathtt{add}\}$ does not hold. This problem is addressed in [2] by introducing *computable closures*. We will provide an alternative in the next section.

3 StarHorpo

Let $\mathcal{F}^\star$ be $\mathcal{F} \uplus (\mathcal{F} \times \mathcal{T})$, in which a function symbol is either a function symbol $f \in \mathcal{F}$, or an ordered pair (f, A), written as $f^\star_A$, where $f \in \mathcal{F}$ and $A \in \mathcal{T}$. We assume $f^\star_A : A$ and $\mathcal{F}^\star \cap \mathcal{V} = \emptyset$. With $\mathcal{F}^\star$, terms are generated and typed likewise. Given a term $f(t_1, \ldots, t_n)$ where $t_i : A_i$ for all i, the newly introduced function symbols allow us to have $f^\star(t_1, \ldots, t_n) : B$ for any B, where $f^\star$ stands for $f^\star_{A_1 \to \cdots \to A_n \to B}$. We will omit the type and write just $f^\star$ whenever the type can be deduced from the context. In the above translation from $f(t_1, \ldots, t_n)$ to $f^\star(t_1, \ldots, t_n)$, *marking* the head symbol serves two purposes:

- $f^\star(t_1, \ldots, t_n)$ can have a different type from the type of $f(t_1, \ldots, t_n)$.
- $f^\star(t_1, \ldots, t_n)$ encodes an obligation to make $f(t_1, \ldots, t_n)$ smaller [4, 7].

We further assume that every marked function symbol $f^\star$ in a term is followed by at least $\mathsf{minar}(f)$ arguments, where the function $\mathsf{minar} : \mathcal{F} \to \mathbb{N}$ is called the *minimal arity*.

A term is said to be *unmarked* if it does not contain any marked function symbol. Given minar and the precedence $\blacktriangleright$ on $\mathcal{F}$, $s \succ_\star t$ if and only if s has the same type as t, t is unmarked, and one of the following conditions is true:

Put	$s = f(s_1, \ldots, s_m)$, $m \geq \mathsf{minar}(f)$ and $f^\star(s_1, \ldots, s_m) \succ_\star t$.
Select	$s = f^\star(s_1, \ldots, s_m)$ and $\exists i\ s_i \cdot f^\star(s_1, \ldots, s_m) \cdots f^\star(s_1, \ldots, s_m) \succeq_\star t$.
Copy	$s = f^\star(s_1, \ldots, s_m)$, $t = g(t_1, \ldots, t_n)$, $f \blacktriangleright g$ and $\forall i\ f^\star(s_1, \ldots, s_m) \succ_\star t_i$.
Lex	$s = f^\star(s_1, \ldots, s_m)$, $t = f(t_1, \ldots, t_n)$, $(s_1, \ldots, s_{\mathsf{minar}(f)}) \succ_\star^{\text{lex}} (t_1, \ldots, t_n)$ and $\forall i\ f^\star(s_1, \ldots, s_m) \succ_\star t_i$.
Mono	$s = s_1 \cdot s_2$, $t = t_1 \cdot t_2$, $s_1 \succeq_\star t_1$, $s_2 \succeq_\star t_2$ and $s \neq t$.

In the above definition, $\succeq_\star$ is the union of $\succ_\star$ and the identity relation on unmarked terms, and $(s_1, \ldots, s_{\mathsf{minar}(f)}) \succ_\star^{\mathrm{lex}} (t_1, \ldots, t_n)$ if and only if $\exists i \leq \min(\mathsf{minar}(f), n)\ (s_i \succ_\star t_i \wedge \forall j < i\ s_j = t_j)$. Occurrences of $f^\star(s_1, \ldots, s_m)$ in the conditions always have an appropriate type, and the number of occurrences of $f^\star(s_1, \ldots, s_m)$ in Select is determined by the type of s_i.

Put allows us to mark a function symbol without changing its type. When we do so, it is required that the marked function symbol $f^\star$ takes at least $\mathsf{minar}(f)$ arguments. Without this restriction, the ordering, denoted by $>$, will end up allowing the derivation in Figure 1. By Mono, we get an infinite sequence $\mathtt{f}(\mathtt{g}) > \mathtt{e}(\mathtt{f}(\mathtt{g},\mathtt{h})) > \mathtt{e}(\mathtt{e}(\mathtt{f}(\mathtt{g},\mathtt{h}),\mathtt{h})) > \cdots$, which shows that $>$ is not well-founded. The "minimal arity" restriction is necessary because Select may cause function symbols to take extra arguments, which is not the case for HORPO, and taking into account the extra arguments in a Lex step can break well-foundedness, as shown in Figure 1.

$$\begin{aligned}
&\mathtt{e} : \mathsf{a} \to \mathsf{a} \to \mathsf{a}\\
&\mathtt{f} : ((\mathsf{a} \to \mathsf{a}) \to \mathsf{a} \to \mathsf{a}) \to \mathsf{a} \to \mathsf{a}\\
&\mathtt{g} : (\mathsf{a} \to \mathsf{a}) \to \mathsf{a} \to \mathsf{a}\\
&\mathtt{h} : \mathsf{a}
\end{aligned}$$

$$\dfrac{\dfrac{\dfrac{\mathtt{g} \blacktriangleright \mathtt{e} \qquad \dfrac{\dfrac{\dfrac{\mathtt{g} \blacktriangleright \mathtt{h}}{\mathtt{g}^\star(\mathtt{f}^\star(\mathtt{g})) > \mathtt{h}}\ \mathsf{Copy}}{\mathtt{f}^\star(\mathtt{g}, \mathtt{g}^\star(\mathtt{f}^\star(\mathtt{g}))) > \mathtt{f}(\mathtt{g},\mathtt{h})}\ \mathsf{Lex}}{\mathtt{g}^\star(\mathtt{f}^\star(\mathtt{g})) > \mathtt{f}(\mathtt{g},\mathtt{h})}\ \mathsf{Select}}{\dfrac{\mathtt{g}^\star(\mathtt{f}^\star(\mathtt{g})) > \mathtt{e}(\mathtt{f}(\mathtt{g},\mathtt{h}))}{\mathtt{g}(\mathtt{f}^\star(\mathtt{g})) > \mathtt{e}(\mathtt{f}(\mathtt{g},\mathtt{h}))}\ \mathsf{Put}}\ \mathsf{Copy}}{\mathtt{f}^\star(\mathtt{g}) > \mathtt{e}(\mathtt{f}(\mathtt{g},\mathtt{h}))}\ \mathsf{Select}}{\mathtt{f}(\mathtt{g}) > \mathtt{e}(\mathtt{f}(\mathtt{g},\mathtt{h}))}\ \mathsf{Put}$$

Figure 1 Non-well-foundedness from dropping the minar restriction

We note that marked function symbols play a role only in generating $\succ_\star$. Because $\to_{\mathcal{R}}$ is a relation on unmarked terms, we should consider the restriction of $\succ_\star$ to unmarked terms when showing the well-foundedness of $\to_{\mathcal{R}}$. Like $\succ_{\mathrm{horpo}}$, the restriction of $\succ_\star$ to unmarked terms is well-founded, monotonic and stable, which means we only need to find a combination of $\blacktriangleright$ and minar that makes $\to_{\mathcal{R}}$ compatible with the generated relation $\succ_\star$.

For example, $\mathtt{sum}(X) \succ_\star \mathtt{rec}(X, \mathtt{0}, \mathtt{add})$ can be obtained as follows, with $\mathsf{minar}(\mathtt{sum}) = 1$:

$$\dfrac{\dfrac{\mathtt{sum} \blacktriangleright \mathtt{rec} \qquad \dfrac{X \succeq_\star X}{\mathtt{sum}^\star(X) \succ_\star X}\ \mathsf{Select} \qquad \dfrac{\mathtt{sum} \blacktriangleright \mathtt{0}}{\mathtt{sum}^\star(X) \succ_\star \mathtt{0}}\ \mathsf{Copy} \qquad \dfrac{\mathtt{sum} \blacktriangleright \mathtt{add}}{\mathtt{sum}^\star(X) \succ_\star \mathtt{add}}\ \mathsf{Copy}}{\mathtt{sum}^\star(X) \succ_\star \mathtt{rec}(X, \mathtt{0}, \mathtt{add})}\ \mathsf{Copy}}{\mathtt{sum}(X) \succ_\star \mathtt{rec}(X, \mathtt{0}, \mathtt{add})}\ \mathsf{Put}$$

Unlike $\succ_{\mathrm{horpo}}$, $\succ_\star$ is necessarily transitive, which we exemplify with the rewrite sequence

$$\mathtt{rec}(\mathtt{s}(\mathtt{s}(X)), Y, \mathtt{add}) \to_{\mathcal{R}} \mathtt{add}(\mathtt{s}(X), \mathtt{rec}(\mathtt{s}(X), Y, \mathtt{add})) \to_{\mathcal{R}} \mathtt{s}(\mathtt{add}(X, \mathtt{rec}(\mathtt{s}(X), Y, \mathtt{add}))).$$

With either $\succ_{\mathrm{horpo}}$ or $\succ_\star$, we can see that in each of the rewrite steps, the term on the left-hand side is greater than the one on the right-hand side, using only $\mathtt{add} \blacktriangleright \mathtt{s}$ and $\mathsf{minar}(\mathtt{rec}) = \mathsf{minar}(\mathtt{add}) = 1$. If we skip the term in the middle and try to directly compare the first and the last in the sequence, $\succ_{\mathrm{horpo}}$ fails unless we further impose $\mathtt{rec} \blacktriangleright \mathtt{s}$. This shows that $\succ_{\mathrm{horpo}}$ is not transitive as imposing extra assumptions can be problematic when there are other rewrite rules in the system. On the other hand, with $\succ_\star$, we do have the following derivation (with irrelevant part omitted):

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathtt{add} \blacktriangleright \mathtt{s} \qquad \dfrac{\vdots}{\mathtt{add}^\star(\mathtt{rec}^\star(\ldots), \mathtt{rec}^\star(\ldots)) \succ_\star \mathtt{add}(X, \mathtt{rec}(\mathtt{s}(X), Y, \mathtt{add}))}\ \mathsf{Lex}}{\mathtt{add}^\star(\mathtt{rec}^\star(\ldots), \mathtt{rec}^\star(\ldots)) \succ_\star \mathtt{s}(\mathtt{add}(X, \mathtt{rec}(\mathtt{s}(X), Y, \mathtt{add})))}\ \mathsf{Copy}}{\mathtt{add}(\mathtt{rec}^\star(\ldots), \mathtt{rec}^\star(\ldots)) \succ_\star \mathtt{s}(\mathtt{add}(X, \mathtt{rec}(\mathtt{s}(X), Y, \mathtt{add})))}\ \mathsf{Put}}{\mathtt{rec}^\star(\mathtt{s}(\mathtt{s}(X)), Y, \mathtt{add}) \succ_\star \mathtt{s}(\mathtt{add}(X, \mathtt{rec}(\mathtt{s}(X), Y, \mathtt{add})))}\ \mathsf{Select}}{\mathtt{rec}(\mathtt{s}(\mathtt{s}(X)), Y, \mathtt{add}) \succ_\star \mathtt{s}(\mathtt{add}(X, \mathtt{rec}(\mathtt{s}(X), Y, \mathtt{add})))}\ \mathsf{Put}$$

In the above derivation, we "select" $\mathtt{add}$ in $\mathtt{rec}^{\star}(\mathtt{s}(\mathtt{s}(X)), Y, \mathtt{add})$ by Select. This reflects the derivation of $\mathtt{rec}(\mathtt{s}(X), Y, F) \succ_{\star} F(X, \mathtt{rec}(X, Y, F))$, in which F is selected. This step is not available with $\succ_{\text{horpo}}$. Also by Select, $\mathtt{add}$ gets two arguments, each with a marked head symbol. The arguments are later compared with some arguments on the right-hand side by Lex. This capacity to postpone comparison is vital to the transitivity of $\succ_{\star}$.

To automate StarHorpo, standard SAT encoding techniques (see, e.g., [8]) can be used. The only limitation is that, to ensure termination, the number of size-increasing applications of Select until the right-hand side is decreased should be bounded. This is implemented in the second author's termination tool WANDA [6], which features the original StarHorpo.

Finally, let us discuss what would happen if we encoded application as function symbols. In this perspective, we could ignore types and view a curried system as a first-order (uncurried) system with a single binary function symbol @. However, all complex terms are thus headed by the same binary function symbol, which sharply limits the applicability of recursive path orderings since we can rarely take advantage of the head symbol comparison, when we apply Copy. Consider the system with only one rewrite rule $\mathtt{f}(X) \to \mathtt{g}(X, X)$, where $\mathtt{f} : \mathtt{a} \to \mathtt{a}$ and $\mathtt{g} : \mathtt{a} \to \mathtt{a} \to \mathtt{a}$. This rule is easily oriented by $\succ_{\star}$ (or $\succ_{\text{horpo}}$) with $\mathtt{f} \blacktriangleright \mathtt{g}$, but its applicative first-order counterpart, $@(\mathtt{f}, X) \to @(@(\mathtt{g}, X), X)$, cannot be tackled. Even if we do not ignore types and introduce a separate symbol $@_{A,B} : (A \to B) \to A \to B$ for any types A and B, the same problem still arises: $@_{\mathtt{a},\mathtt{a}}(\mathtt{f}, X) \succ_{\star} @_{\mathtt{a},\mathtt{a}}(@_{\mathtt{a},\mathtt{a}\to\mathtt{a}}(\mathtt{g}, X), X)$ is not obtainable.

4 Conclusion

We have adapted StarHorpo to an applicative system. Changing the underlying formalism requires extra attention: the arity restriction of functional notation should not be dropped naively; instead, we impose the minimal arity, a weaker version of arity, on StarHorpo. Interestingly, on the same applicative system, our definition of HORPO does not seem in need of any kind of arity. While this definition is indeed more powerful in some cases, without Select, which can give function symbols extra arguments, HORPO is not necessarily transitive. We thus incorporate both Select and minar into StarHorpo to gain transitivity while maintaining well-foundedness.

In order to have CPO included in StarHorpo for curried systems, we still need to take into account lambda abstractions and type orderings, as well as the multiset extension, in our definition. Furthermore, another direction for future work is to apply StarHorpo to determine the termination of functional programs, which may require us to extend StarHorpo with support for real-world data types such as integers and floating-point numbers.

References

1 F. Blanqui, J. Jouannaud, and A. Rubio. The computability path ordering: The end of a quest. In *Proc. CSL*, 2008.
2 J. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *Proc. LICS*, 1999.
3 S. Kamin and J.-J. Lévy. Two generalizations of the recursive path ordering. Unpublished Manuscript, University of Illinois, 1980.
4 J.W. Klop, V. van Oostrom, and R. de Vrijer. Iterative lexicographic path orders. In *Algebra, Meaning, and Computation*. 2006.
5 C. Kop. *Higher Order Termination*. PhD thesis, VU Amsterdam, 2012.
6 C. Kop. WANDA – a higher-order termination tool. In *Proc. FSCD*, 2020.
7 C. Kop and F. van Raamsdonk. A higher-order iterative path ordering. In *Proc. LPAR*, 2008.
8 P. Schneider-Kamp, R. Thiemann, E. Annov, M. Codish, and J. Giesl. Proving termination using recursive path orders and SAT solving. In *Proc. FroCoS*, 2007.

Approximating Relative Match-Bounds

Alfons Geser
HTWK Leipzig

Dieter Hofbauer
ASW Saarland

Johannes Waldmann
HTWK Leipzig

Abstract

We present a simple method for obtaining automata that over-approximate match-heights for the set of right-hand sides of forward closures of a given string rewrite system. The method starts with an automaton that represents height-indexed copies of right-hand sides, and inserts epsilon transitions only. Rules that have no redex path in the highest level, are relatively match-bounded, and thus terminating relative to the others, on the starting language. For height bound 0, we obtain a generalisation of right barren string rewriting. An implementation of our method proves termination of 590 out of 595 benchmarks in `TPDB/SRS_Standard/ICFP_2010` quickly.

2012 ACM Subject Classification Theory of computation → Rewrite systems

Keywords and phrases Termination, String rewriting, Match-bounds, Right barren

1 Right Barren String Rewriting

Forward closures ("chains" in [10]) are pairs of strings that represent restricted derivations. The set of right-hand sides of forward closures is denoted by $\mathsf{RFC}(R)$. By a Theorem of Dershowitz [3], termination of a rewrite system R is equivalent to termination of R on $\mathsf{RFC}(R)$. For our purposes, we will employ the characterization $\mathsf{RFC}(R) = (\to_R \cup \rightharpoondown_{\mathsf{right}(R)})^*(\mathsf{rhs}(R))$, where $\rightharpoondown_{\mathsf{right}(R)}$ is the suffix rewrite relation induced by the system $\mathsf{right}(R) = \{\ell_1 \to r \mid (\ell_1\ell_2 \to r) \in R, \ell_1 \neq \epsilon \neq \ell_2\}$, cf. [6, Lemma 4] for an equivalent variant.

It might be the case that the relation $\to_R$ is never used when computing $\mathsf{RFC}(R)$. This was observed for one-rule systems by McNaughton [11] (for the non-overlapping case) and Geser [5] (for the general case). Such systems R are called right barren, and they are always terminating.

We generalize this approach to an arbitrary number of rules. A string rewrite system R is called *right barren* if there is no $\ell \in \mathsf{lhs}(R)$ such that ℓ is a factor of some string from $\mathsf{RFC}(R)$. Then R terminates on $\mathsf{RFC}(R)$, so R terminates everywhere.

The right barren property is decidable: we first compute the regular language $L = (\rightharpoondown_{\mathsf{right}(R)})^*(\mathsf{rhs}(R))$. By a result of Büchi on prefix rewriting [1, 2] (cf. [8, Sect. 6.1]), this can be effectively done via an automaton construction. Then we check that L does not contain any left-hand side of R as a factor, i. e., $L \cap \Sigma^* \cdot \mathsf{lhs}(R) \cdot \Sigma^*$ is empty.

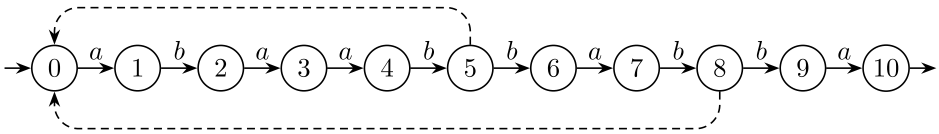
Our implementation computes the closure of $\mathsf{rhs}(R)$ under suffix rewriting w. r. t. $\mathsf{right}(R)$ as follows: We start with a finite state automaton that consists of isolated paths, one for each string in $\mathsf{rhs}(R)$. The set of initial states I (resp. set of final states F) contains all starting points (resp. end points) of these paths. We then add epsilon transitions as follows: For each $(\ell_1 \to r) \in \mathsf{right}(R)$, if the automaton contains a path $p \xrightarrow{\ell_1} q \in F$ for states p and q, then add an epsilon transition from p to the starting point of the path for r. In this case, we say that rule $\ell_1 \to r$ has a *suffix match* at state p. Note that this completion procedure always terminates, since the set of states is constant.

Our implementation keeps the set of epsilon transitions transitively closed after each addition. This means that when we trace a path, we need to do at most one epsilon step between real steps. In the diagrams below, we suppress transitive edges.

Unless noted otherwise, examples refer to directory `SRS_Standard` from the Termination Problems Database (TPDB) 11.0, see `https://termination-portal.org/wiki/TPDB`.

▶ **Example 1.** Consider the one-rule system $R = \{babbaba \to abaabbabba\}$, which is `Zantema_04/z033`. This system is right barren, as certified by the following automaton. Throughout all state diagrams, initial and final states are indicated by isolated in- or outgoing edges, respectively, and epsilon transitions are dashed. Here, states correspond to positions between letters, i. e., for right-hand side r we get states s with $0 \le s \le |r|$.

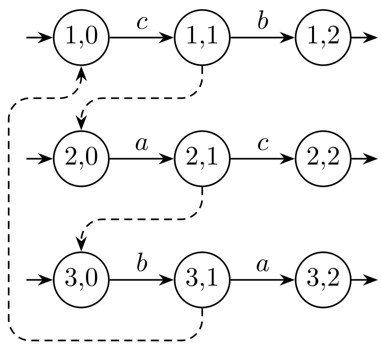
The completion procedure starts with one path from state 0 to state 10, labelled by the right-hand side of the rule. For $ba \in \mathsf{lhs}(\mathsf{right}(R))$ there is the path $8 \xrightarrow{ba} 10 \in F$, so an epsilon transition from 8 to the initial state 0 is added. Analogously, for $babba \in \mathsf{lhs}(\mathsf{right}(R))$ the path $5 \xrightarrow{babba} 10 \in F$ results in an epsilon transition from 5 to 0. No further epsilon transitions need to be added.



We observe that the left-hand side $\ell = babbaba$ is not a factor of any string in the accepted language of the resulting automaton, since no path $p \xrightarrow{\ell} q$ for states p and q exists, therefore R is right barren, thus terminating. ◀

▶ **Example 2.** Our approach also applies to systems with more than just one rule, as demonstrated by the example $R = \{aa \to cb, bb \to ac, cc \to ba\}$ (`Zantema_04/z087`). For $|R| > 1$, we choose states as pairs (n, s) with $1 \le n \le |R|$ and, as before, s with $0 \le s \le |r|$ for $r \in \mathsf{rhs}(R)$.

Here, completion starts with an automaton consisting of three paths, one for each right-hand side. For rule $(a \to cb) \in \mathsf{right}(R)$ there is a *suffix match* at state $(3, 1)$, so we add an epsilon transition from $(3, 1)$ to $(1, 0)$. Analogously, we get two more epsilon transitions, from $(1, 1)$ to $(2, 0)$ due to the suffix match of $(b \to ac) \in \mathsf{right}(R)$ at $(1, 1)$, and from $(2, 1)$ to $(3, 0)$ due to the suffix match of $(c \to ba) \in \mathsf{right}(R)$ at $(2, 1)$. The resulting automaton is closed w. r. t. $\mathsf{right}(R)$, and its language does not contain any left-hand side of R as a factor, so R is right barren, hence terminating.



◀

2 Removal of Relatively Right Barren Rules

The algorithm of Section 1 rejects if some left-hand side of R occurs as a factor of $\mathsf{RFC}(R)$. We now describe how to continue in this case.

We call a rule $\ell \to r$ from R *relatively right barren* w. r. t. the other rules, if ℓ does not occur as a factor of $\mathsf{RFC}(R)$. Relatively right barren rules can be removed from the termination problem: if all rules from $S \subseteq R$ are relatively right barren w. r. t. $R \setminus S$, and $R \setminus S$ is terminating, then R is terminating. This includes the previous concept as a special case: if a system is right barren, then all its rules are relatively right barren.

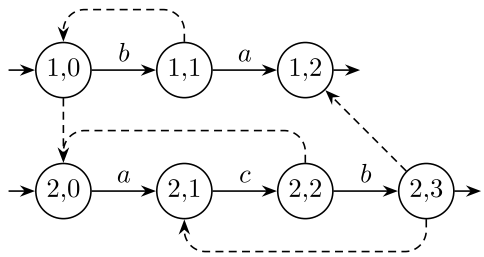
The definition of relative right barrenness uses $\mathsf{RFC}(R)$, and that set might be impossible to represent by a finite automaton. We therefore extend the previous algorithm to obtain an over-approximation: for a rule $\ell \to r$ in R and a redex path $p \xrightarrow{\ell} q$ in the automaton, we add one epsilon transition from p to the start of r, and one epsilon transition from the end of r to q.

Again, this procedure obviously terminates. The language $L(A)$ of the resulting automaton A contains $\mathsf{rhs}(R)$, and A is closed w. r. t. $\to_R$ and $\to_{\mathsf{right}(R)}$, so $L(A)$ over-approximates $\mathsf{RFC}(R)$. Rules without redex in $L(A)$ are relatively right barren, and can be removed. The approximation error comes from identifying all reduct paths of each rule.

Example 3 exhibits a system where the algorithm of this section allows to remove a rule, even though the system is not right barren.

▶ **Example 3.** Let $R = \{ab \to ba, ba \to acb\}$ (`Zantema_04/z006`). First we add two epsilon transitions, $(1,1) \xrightarrow{\epsilon} (1,0)$ due to the suffix match of $(a \to ba) \in \mathsf{right}(R)$ at $(1,1)$, and $(2,2) \xrightarrow{\epsilon} (2,0)$ due to the suffix match of $(b \to acb) \in \mathsf{right}(R)$ at $(2,2)$.

As there is a redex path $(1,0) \xrightarrow{ba} (1,2)$ for the second rule, we add the epsilon transitions $(1,0) \xrightarrow{\epsilon} (2,0)$ and $(2,3) \xrightarrow{\epsilon} (1,2)$. This creates a new redex path $(1,0) \xrightarrow{ba} (2,1)$ for the second rule (note that this path uses two epsilon transitions), resulting in $(1,0) \xrightarrow{\epsilon} (2,0)$ (already present) and $(2,3) \xrightarrow{\epsilon} (2,1)$ (a fresh transition). Note that the existence of these two redex paths entails that R is not right barren.



The resulting automaton is closed w. r. t. R and $\mathsf{right}(R)$, so the completion procedure stops. As there is no path labelled by ab, the rule $ab \to ba$ can be removed from R. The remaining system $\{ba \to acb\}$ is terminating (it is right barren, in fact), proving termination of R. ◀

3 Approximating Match-Bounds

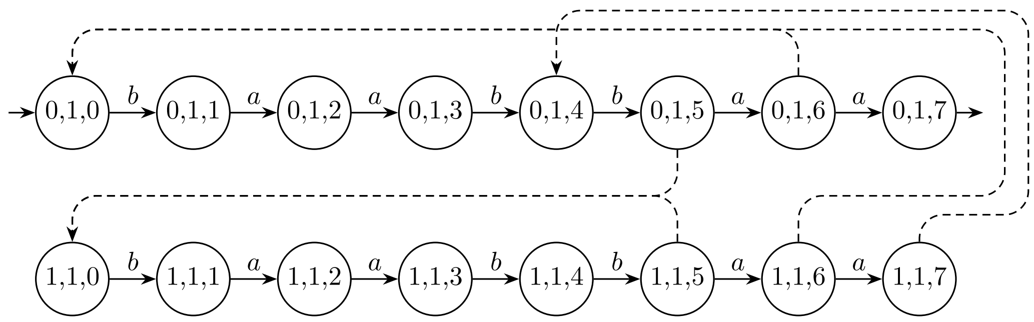
We refine the approximation of $\mathsf{RFC}(R)$ by match-heights [6]. We fix a number $B \in \mathbb{N}$, and let the initial automaton consist of layers $0, 1, \ldots, B$, where layer h contains disjoint paths for $\mathsf{rhs}(R)$ of match-height h. We put the height information not on the letters, but in the states: a state of the automaton now is a triple of number of layer, number of rule, and position in right-hand-side. The initial (final, resp.) states of the automaton are the initial (final, resp.) states of paths in layer 0.

For each suffix match for a rule $(\ell_1 \to r) \in \mathsf{right}(R)$, a new epsilon transition links to the starting point of r at height 0.

For each redex path for a rule from R, we compute its match-height h as the minimal layer of its letter transitions; the path also might contain epsilon transitions, these have no height. We reject if $h = B$. Else, we introduce epsilon transitions to, and from, the reduct path at height $h + 1$. If we succeed, we obtain an automaton certifying that R is match-bounded by B on $\mathsf{RFC}(R)$, thus R is terminating. The method of Section 1 is the special case of $B = 0$.

Example 4 illustrates this approach. This rewrite system is not right barren, and none of its rules is relatively right barren, so the criteria from Sections 1 and 2 are not successful.

▶ **Example 4.** Let $R = \{abaab \to baabbaa\}$, the reversal of `Zantema_04/z034`. Our algorithm produces a certificate for match-bound 1 on $\mathsf{RFC}(R)$, as follows.



Completion starts with an automaton consisting of the two paths $(0,1,0) \xrightarrow{baabbaa} (0,1,7)$ for height 0, and $(1,1,0) \xrightarrow{baabbaa} (1,1,7)$ for height 1. For the suffix match of $a \to baabbaa \in \mathsf{right}(R)$ at $(0,1,6)$ we add the transition $(0,1,6) \xrightarrow{\epsilon} (0,1,0)$. This creates the R-redex path $(0,1,5) \xrightarrow{a} (0,1,6) \xrightarrow{\epsilon} (0,1,0) \cdots (0,1,3) \xrightarrow{b} (0,1,4)$ of height 0, and we link to the reduct path of height 1 by adding $(0,1,5) \xrightarrow{\epsilon} (1,1,0)$ and $(1,1,7) \xrightarrow{\epsilon} (0,1,4)$. There is a suffix match at $(1,1,6)$. It is not for $a \to baabbaa$, since $(1,1,7)$ is not final, but for $abaa \to baabbaa$, via the path $(1,1,6) \xrightarrow{a} (1,1,7) \xrightarrow{\epsilon} (0,1,4) \xrightarrow{baa} (0,1,7)$. We add an edge $(1,1,6) \xrightarrow{\epsilon} (0,1,0)$. Now we have another R-redex path from $(1,1,5)$ to $(0,1,4)$. This path has minimal height 0 (only the first step has height 1), so we link to the reduct path at height 1 by the transitions $(1,1,5) \xrightarrow{\epsilon} (1,1,0)$ and $(1,1,7) \xrightarrow{\epsilon} (0,1,4)$ (the last transition already existing). The automaton is now closed. ◀

4 Removal of Relatively Match-Bounded Rules

Match-bounds can be used to remove rules [9]: A set of rules S is *match-bounded by* $B \in \mathbb{N}$ *relative to* R, *on a language* L, if in each (possibly infinite) height-annotated $(S \cup R)$-derivation starting with some zero-annotated string from L, each reduct of S has height $\leq B$. Then S is terminating relative to R on L. We extend the method of Section 3 accordingly: we let the highest layer B represent all higher layers as well: for a redex path with height B, we do no longer reject, but we use the reduct path at the same height B. We then remove the subset of rules where all redex heights are $< B$. They are match-bounded by B relative to R, on $\mathsf{RFC}(R)$, and thus, they terminate relatively to R. The method of Section 2 is the special case of $B = 0$.

▶ **Example 5.** For $R = \{aba \to baa, aac \to acab\}$, with $B = 2$, the algorithm removes the second rule, since its highest redex is at height 1 only. Methods of earlier sections do not apply: neither rule is relatively right-barren, and neither R nor its reversal is match-bounded on RFC. ◀

5 Experimental Evaluation

We focus on the 595 benchmarks in `ICFP_2010`. These benchmarks typically contain a large number of rules (average `ICFP`: 70, non-`ICFP`: 3.3) with large total size, i. e., sum of lhs and rhs lengths (average `ICFP`: 2340, non-`ICFP`: 21.5). By construction [4], each of these systems does admit a natural matrix interpretation. We will not make use of these matrices (they are stored, in some obfuscated format, in some obscure place) and we do not aim to reconstruct them. Termination Competitions show that these benchmarks are hard: in 2021, at a time-out of 5 minutes, 514 benchmarks were solved (86 percent). Over these solved benchmarks, the average CPU time of the "virtual best solver" was 90 seconds (median: 28 seconds). Of the 1056 non-ICFP benchmarks, 1017 were solved (96 percent), and the virtual best solver's average time was 51 seconds (median: 6 seconds).

The following table shows the number of `ICFP` benchmarks solved with methods from the present paper, with a time-out of 10 seconds only. In all cases, the method was tried for the reversed system as well, and rule removal by weights was also applied, with GLPK (GNU Linear Programming Kit) as a solver. The methods for rule removal (Sections 2 and 4) were iterated. Data is available on Starexec (Job 51953), and can be navigated at `https://termcomp.imn.htwk-leipzig.de/flexible-table/Query[]/51953`.

method (Section)	right bar. (1)	rel. right bar. (2)	rfc-mb (3)	rel. rfc-mb (4)
number of problems solved	370	568	588	590
percentage	62.2	95.5	98.8	99.2
average CPU time (seconds)	0.29	0.88	1.37	0.93

These methods are independently implemented in MultumNonMulta [7] and in Matchbox (`https://gitlab.imn.htwk-leipzig.de/waldmann/pure-matchbox`).

For non-`ICFP` benchmarks from TPDB, performance is not that strong. We guess the reason is that they have shorter rules that create more overlaps, increasing the approximation error that comes from re-using right-hand sides.

References

1 J. Richard Büchi. Regular canonical systems. *Archiv Math. Logik und Grundlagenforschung*, 6:91–111, 1964.

2 J. Richard Büchi. *Finite Automata, Their Algebras and Grammars – Towards a Theory of Formal Expressions (Dirk Siefkes, editor).* Springer, New York, 1989.

3 Nachum Dershowitz. Termination of linear rewriting systems. In Shimon Even and Oded Kariv, editors, *Automata, Languages and Programming, 8th Colloquium, Acre (Akko), Israel, July 13-17, 1981, Proceedings*, volume 115 of *LNCS*, pages 448–458. Springer, 1981.

4 Bertram Felgenhauer and Johannes Waldmann. The ICFP 2010 Programming Contest. `https://www.imn.htwk-leipzig.de/~waldmann/talk/10/icfp/`, 2010.

5 Alfons Geser. *Is Termination Decidable for String Rewriting with only One Rule?* Habilitationsschrift, Eberhard-Karls-Universität, Tübingen, Germany, 2001.

6 Alfons Geser, Dieter Hofbauer, and Johannes Waldmann. Match-bounded string rewriting systems. *Appl. Algebra Eng. Commun. Comput.*, 15(3-4):149–171, 2004.

7 Dieter Hofbauer. MultumNonMulta at TermComp 2018. In Salvador Lucas, editor, *16th Intl. Workshop on Termination, WST 2018, Oxford, U. K., 2018, Proceedings*, page 80, 2018.

8 Dieter Hofbauer and Johannes Waldmann. Deleting string rewriting systems preserve regularity. *Theor. Comput. Sci.*, 327(3):301–317, 2004.

9 Dieter Hofbauer and Johannes Waldmann. Match-bounds for relative termination. In Peter Schneider-Kamp, editor, *11th Intl. Workshop on Termination, WST 2010, Edinburgh, U. K.*, 2010.

10 Dallas S. Lankford and David R. Musser. A finite termination criterion. Technical report, Information Sciences Institute, Univ. of Southern California, Marina-del-Rey, CA, 1978.

11 Robert McNaughton. The uniform halting problem for one-rule Semi-Thue Systems. Technical Report 94-18, Dept. of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, August 1994. See also "Correction to 'The Uniform Halting Problem for One-rule Semi-Thue Systems'", unpublished paper, August, 1996.

Hydra Battles and AC Termination

Nao Hirokawa
School of Information Science, JAIST, Japan

Aart Middeldorp
Department of Computer Science, University of Innsbruck, Austria

Abstract

We encode the Battle of Hercules and Hydra as a rewrite system with AC symbols. Its termination is proved by type introduction and a new termination criterion for AC rewriting.

2012 ACM Subject Classification Theory of computation → Equational logic and rewriting; Theory of computation → Rewrite systems; Theory of computation → Computability

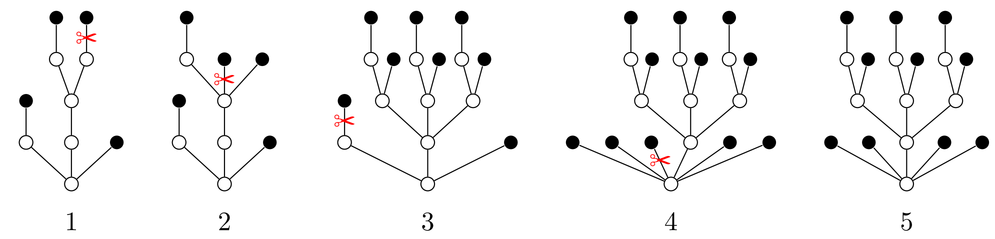
Keywords and phrases battle of Hercules and Hydra, term rewriting, termination

Digital Object Identifier 10.4230/LIPIcs.WST.2022.1

Funding *Nao Hirokawa*: JSPS KAKENHI Grant Numbers 22K11900
Aart Middeldorp: Part of this work was performed when the second author was employed at the Future Value Creation Research Center of Nagoya University, Japan.

1 Introduction

The mythological monster Hydra is a dragon-like creature with multiple heads. Whenever Hercules in his fight chops off a head, more and more new heads can grow instead, since the beast gets increasingly angry. Here we model a Hydra as an unordered tree. If Hercules cuts off a leaf corresponding to a head, the tree is modified in the following way: If the cut-off node h has a grandparent n, then the branch from n to the parent of h gets multiplied, where the number of copies depends on the number of decapitations so far. Hydra dies if there are no heads left, in that case Hercules wins. The following sequence shows an example fight:



Though the number of heads can grow considerably in one step, it turns out that the fight always terminates, and Hercules will win independent of his strategy. This can be shown by an argument based on ordinals [5]. Starting with [2, p. 271], several TRS encodings of the Battle of Hercules and Hydra have been proposed and studied [1, 3, 4, 7, 8]. Touzet [8] was the first to give a rigorous termination proof and in [9] the automation of ordinal interpretations is discussed. In this note we present yet another encoding. In contrast to earlier TRS encodings that model a specific strategy, it uses AC matching to represent *arbitrary* battles.

▶ **Definition 1.** *To represent Hydras, we use a signature containing a constant symbol* h *representing a head, a binary symbol* $\mid$ *for siblings, and a unary function symbol* i *representing the internal nodes. We use infix notation for* $\mid$ *and declare it to be an* AC *symbol.*

18th International Workshop on Termination (WST 2022).
Editor: Cynthia Kop; Article No. 1; pp. 1:1–1:5
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

▶ **Example 2.** The Hydras in the above example fight are represented by the terms

$$H_1 = \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{i}(\mathsf{h}))) \mid \mathsf{h})$$
$$H_2 = \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h} \mid \mathsf{h})) \mid \mathsf{h})$$
$$H_3 = \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h}) \mid \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h}) \mid \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h})) \mid \mathsf{h})$$
$$H_4 = \mathsf{i}(\mathsf{h} \mid \mathsf{h} \mid \mathsf{h} \mid \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h}) \mid \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h}) \mid \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h})) \mid \mathsf{h} \mid \mathsf{h})$$
$$H_5 = \mathsf{i}(\mathsf{h} \mid \mathsf{h} \mid \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h}) \mid \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h}) \mid \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h})) \mid \mathsf{h} \mid \mathsf{h})$$

▶ **Definition 3.** *The* TRS $\mathcal{H}$ *consists of the following 14 rewrite rules:*

$$\begin{aligned}
\mathsf{A}(n, \mathsf{i}(\mathsf{h})) &\xrightarrow{1} \mathsf{h} & \mathsf{D}(n, \mathsf{i}(\mathsf{i}(x))) &\xrightarrow{8} \mathsf{i}(\mathsf{D}(n, \mathsf{i}(x))) \\
\mathsf{A}(n, \mathsf{i}(\mathsf{h} \mid x)) &\xrightarrow{2} \mathsf{A}(\mathsf{s}(n), \mathsf{i}(x)) & \mathsf{D}(n, \mathsf{i}(\mathsf{i}(x) \mid y)) &\xrightarrow{9} \mathsf{i}(\mathsf{D}(n, \mathsf{i}(x)) \mid y) \\
\mathsf{A}(n, \mathsf{i}(x)) &\xrightarrow{3} \mathsf{B}(n, \mathsf{D}(\mathsf{s}(n), \mathsf{i}(x))) & \mathsf{D}(n, \mathsf{i}(\mathsf{i}(\mathsf{h} \mid x) \mid y)) &\xrightarrow{10} \mathsf{i}(\mathsf{C}(n, \mathsf{i}(x)) \mid y) \\
\mathsf{C}(\mathsf{0}, x) &\xrightarrow{4} \mathsf{E}(x) & \mathsf{D}(n, \mathsf{i}(\mathsf{i}(\mathsf{h} \mid x))) &\xrightarrow{11} \mathsf{i}(\mathsf{C}(n, \mathsf{i}(x))) \\
\mathsf{C}(\mathsf{s}(n), x) &\xrightarrow{5} x \mid \mathsf{C}(n, x) & \mathsf{D}(n, \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid y)) &\xrightarrow{12} \mathsf{i}(\mathsf{C}(n, \mathsf{h}) \mid y) \\
\mathsf{i}(\mathsf{E}(x) \mid y) &\xrightarrow{6} \mathsf{E}(\mathsf{i}(x \mid y)) & \mathsf{D}(n, \mathsf{i}(\mathsf{i}(\mathsf{h}))) &\xrightarrow{13} \mathsf{i}(\mathsf{C}(n, \mathsf{h})) \\
\mathsf{i}(\mathsf{E}(x)) &\xrightarrow{7} \mathsf{E}(\mathsf{i}(x)) & \mathsf{B}(n, \mathsf{E}(x)) &\xrightarrow{14} \mathsf{A}(\mathsf{s}(n), x)
\end{aligned}$$

The Battle is started with the term $\mathsf{A}(\mathsf{0}, t)$ where t is the term representation of the initial Hydra. Rule 1 takes care of the dying Hydra $\circ\!\!-\!\!\!-\!\!\bullet$. Rule 2 cuts a head without grandparent node, and so no copying takes place. Due to the power of AC matching, the removed head need not be the leftmost one. With rule 3, the search for locating a head with grandparent node starts. The search is performed with the auxiliary symbol D and involves rules 8–13. When the head to be cut is located (in rules 10–13), copying begins with the auxiliary symbol C and rules 4 and 5. The end of the copying phase is signaled with E, which travels upwards with rules 6 and 7. Finally, rule 14 creates the next stage of the Battle. Note that we make extensive use of AC matching to simplify the search process.

▶ **Theorem 4.** *If H and H' are the encodings in $\mathcal{T}(\{\mathsf{h}, \mathsf{i}, \mid\}) \setminus \{\mathsf{h}\}$ of successive Hydras in an arbitrary battle then $\mathsf{A}(n, H)\ (=_{\mathsf{AC}} \cdot \to \cdot =_{\mathsf{AC}})^{+}\ \mathsf{A}(\mathsf{s}(n), H')$ for some $n \in \mathcal{T}(\{\mathsf{0}, \mathsf{s}\})$.*

Rather than presenting a proof, we content ourselves with an example.

▶ **Example 5.** The following sequence simulates the first step in the example fight:

$$\begin{aligned}
\mathsf{A}(\mathsf{0}, H_1) &\xrightarrow{3} \mathsf{B}(\mathsf{0}, \mathsf{D}(\mathsf{s}(\mathsf{0}), H_1)) \\
=_{\mathsf{AC}} \cdot &\xrightarrow{9} \mathsf{B}(\mathsf{0}, \mathsf{i}(\mathsf{D}(\mathsf{s}(\mathsf{0}), \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{i}(\mathsf{h})))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h})) \\
&\xrightarrow{8} \mathsf{B}(\mathsf{0}, \mathsf{i}(\mathsf{i}(\mathsf{D}(\mathsf{s}(\mathsf{0}), \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{i}(\mathsf{h})))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h})) \\
&\xrightarrow{12} \mathsf{B}(\mathsf{0}, \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{C}(\mathsf{s}(\mathsf{0}), \mathsf{h}) \mid \mathsf{i}(\mathsf{h}))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h})) \\
&\xrightarrow{5} \mathsf{B}(\mathsf{0}, \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{C}(\mathsf{0}, \mathsf{h}) \mid \mathsf{i}(\mathsf{h}))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h})) \\
&\xrightarrow{4} \mathsf{B}(\mathsf{0}, \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{E}(\mathsf{h}) \mid \mathsf{i}(\mathsf{h}))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h})) \\
=_{\mathsf{AC}} \cdot &\xrightarrow{6} \mathsf{B}(\mathsf{0}, \mathsf{i}(\mathsf{i}(\mathsf{E}(\mathsf{i}(\mathsf{h} \mid \mathsf{h} \mid \mathsf{i}(\mathsf{h})))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h})) \\
&\xrightarrow{7} \mathsf{B}(\mathsf{0}, \mathsf{i}(\mathsf{E}(\mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{h} \mid \mathsf{i}(\mathsf{h})))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h})) \\
&\xrightarrow{6} \mathsf{B}(\mathsf{0}, \mathsf{E}(\mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{h} \mid \mathsf{i}(\mathsf{h}))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h}))) \\
&\xrightarrow{14} \mathsf{A}(\mathsf{s}(\mathsf{0}), \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{h} \mid \mathsf{i}(\mathsf{h}))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h})) =_{\mathsf{AC}} \mathsf{A}(\mathsf{s}(\mathsf{0}), H_2)
\end{aligned}$$

2 Termination Criterion

We present a new AC termination criterion based on weakly monotone algebras.

▶ **Definition 6.** *Let A be a set equipped with a strict order $>$ and let $A_1, \ldots, A_n$ be subsets of A. An n-ary function $\phi : A_1 \times \cdots \times A_n \to A$ is*

- strictly monotone *if $\phi(a_1, \ldots, a_i, \ldots, a_n) > \phi(a_1, \ldots, b, \ldots, a_n)$ for all $1 \leqslant i \leqslant n$, $(a_1, \ldots, a_n) \in A_1 \times \cdots \times A_n$, and $b \in A_i$ with $a_i > b$,*
- weakly monotone *if $\phi(a_1, \ldots, a_i, \ldots, a_n) \geqslant \phi(a_1, \ldots, b, \ldots, a_n)$ for all $1 \leqslant i \leqslant n$, $(a_1, \ldots, a_n) \in A_1 \times \cdots \times A_n$, and $b \in A_i$ with $a_i > b$, and*
- simple *if $\phi(a_1, \ldots, a_i, \ldots, a_n) \geqslant a_i$ for all $1 \leqslant i \leqslant n$ and $(a_1, \ldots, a_n) \in A_1 \times \cdots \times A_n$.*

▶ **Definition 7.** *An $\mathcal{S}$-sorted $\mathcal{F}$-algebra $\mathcal{A} = (\{S_{\mathcal{A}}\}_{S \in \mathcal{S}}, \{f_{\mathcal{A}}\}_{f \in \mathcal{F}})$ equipped with a strict order $>$ on the union of all carriers $S_{\mathcal{A}}$ is* simple monotone *if*

- *every carrier $S_{\mathcal{A}}$ is non-empty,*
- *$(S_i)_{\mathcal{A}} \subseteq S_{\mathcal{A}}$ for all $f : S_1 \times \cdots \times S_n \to S$ in $\mathcal{F}$ and $1 \leqslant i \leqslant n$, and*
- *every interpretation function $f_{\mathcal{A}}$ is simple and weakly monotone.*

A simple monotone algebra is totally ordered *if $>$ is a total order. Let $(\mathcal{A}, >)$ be an algebra equipped with a strict order on A. The order $>_{\mathcal{A}}$ induced from $\mathcal{A}$ is defined on terms as usual: $s >_{\mathcal{A}} t$ if $[\alpha]_{\mathcal{A}}(s) > [\alpha]_{\mathcal{A}}(t)$ for all assignments α for $\mathcal{A}$.*

In general the order induced from a totally ordered simple monotone algebra is not a reduction order as it is not closed under contexts. Nevertheless, its compatibility entails AC termination.

▶ **Theorem 8.** *A TRS $\mathcal{R}$ over a finite many-sorted signature $\mathcal{F}$ is AC terminating if there exists a totally ordered simple monotone $\mathcal{F}$-algebra $(\mathcal{A}, >)$ such that $\mathcal{R} \subseteq >_{\mathcal{A}}$, $\mathsf{AC} \subseteq =_{\mathcal{A}}$, and $f_{\mathcal{A}}$ is strictly monotone for all AC symbols f.*

Touzet [8] proved total termination of a non-AC version of $\mathcal{H}$ by devising a termination criterion based on totally ordered simple monotone algebras. Related results are presented in Zantema [11, Section 4]. The above theorem is a proper extension to AC termination as well as a generalization to many-sorted rewrite systems. The proof is based on a variant of Touzet's original proof method.

▶ **Example 9.** Consider the one-rule TRS $\mathcal{R}$ over the single-sorted signature $\{\mathsf{f}^{(1)}, \mathsf{g}^{(1)}, |^{(2)}\}$:

$$\mathsf{f}(\mathsf{g}(x \mid y)) \to \mathsf{g}(\mathsf{f}(\mathsf{f}(x \mid y)))$$

We designate $|$ as an AC symbol. So AC consists of the equations $(x \mid y) \mid z \approx x \mid (y \mid z)$ and $x \mid y \approx y \mid x$. Consider the algebra $\mathcal{A}$ on the carrier $\mathbb{O}$ of ordinals with the following interpretation functions:

$$\mathsf{f}_{\mathcal{A}}(x) = x + 1 \qquad \mathsf{g}_{\mathcal{A}}(x) = x + \omega \qquad x \mid_{\mathcal{A}} y = x \oplus y$$

Here $\oplus$ is natural addition on ordinals. Since $\oplus$ is strictly monotone and $+$ is weakly monotone in its first argument and strictly monotone in its second argument, the interpretation functions are weakly monotone. As $\oplus$ is associative and commutative, $\mathsf{AC} \subseteq =_{\mathcal{A}}$ holds. Moreover, $\mathcal{R} \subseteq >_{\mathcal{A}}$ is verified by the inequality

$$\mathsf{f}_{\mathcal{A}}(\mathsf{g}_{\mathcal{A}}(x \mid_{\mathcal{A}} y)) = (x \oplus y) + \omega + 1 > (x \oplus y) + \omega = (x \oplus y) + 2 + \omega = \mathsf{g}_{\mathcal{A}}(\mathsf{f}_{\mathcal{A}}(\mathsf{f}_{\mathcal{A}}(x \mid_{\mathcal{A}} y)))$$

where $\omega + 1 > \omega$ and $2 + \omega = \omega$ are used. Hence $\mathcal{R}$ is AC terminating.

It is essential for Theorem 8 to demand $f_{\mathcal{A}}(x_1,\ldots,x_i,\ldots,x_n) \geqslant x_i$ even if f has the sort declaration $S_1 \times \cdots \times S_i \times \cdots \times S_n \to S$ with $S_i \neq S$.

▶ **Example 10.** Consider the non-terminating TRS $\mathcal{R} = \{\mathsf{a} \to \mathsf{g}(\mathsf{f}(\mathsf{a}))\}$ over the signature $\{\mathsf{a} : \mathsf{A}, \mathsf{f} : \mathsf{A} \to \mathsf{B}, \mathsf{g} : \mathsf{B} \to \mathsf{A}\}$ and the algebra $\mathcal{A}$ with carriers $\mathsf{A}_{\mathcal{A}} = \mathsf{B}_{\mathcal{A}} = \mathbb{N}$ and interpretation functions $\mathsf{a}_{\mathcal{A}} = 1$, $\mathsf{f}_{\mathcal{A}}(x) = 0$ and $\mathsf{g}_{\mathcal{A}}(x) = x$. Observe that the argument sort (A) and output sort (B) of f are different. If the requirement of $\mathsf{f}_{\mathcal{A}}(x) \geqslant x$ is dropped from Theorem 8, the termination of $\mathcal{R}$ would be wrongly concluded.

Theorem 8 imposes strict monotonicity on the interpretation functions of AC symbols and totality on the order of the simple monotone algebra. We do not know whether these conditions are essential. In the absence of AC symbols, totality of the order can be dropped [11, Theorem 11].

3 Termination Proof

We show that $\mathcal{H}$ is AC terminating. In order to ease its proof we employ *type introduction* [10]. The following theorem is a special case of [6, Corollary 3.9].

▶ **Theorem 11.** *A non-collapsing* TRS *over a many-sorted signature is* AC *terminating if and only if the corresponding* TRS *over the unsorted version of the signature is* AC *terminating.*

The TRS $\mathcal{H}$ can be seen as a TRS over the many-sorted signature $\mathcal{F}'$:

$$\mathsf{h} : \mathsf{O} \qquad \mathsf{i}, \mathsf{E} : \mathsf{O} \to \mathsf{O} \qquad | : \mathsf{O} \times \mathsf{O} \to \mathsf{O} \qquad \mathsf{0} : \mathsf{N} \qquad \mathsf{s} : \mathsf{N} \to \mathsf{N} \qquad \mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D} : \mathsf{N} \times \mathsf{O} \to \mathsf{O}$$

where O and N are sort symbols. Since $\mathcal{H}$ is non-collapsing, Theorem 11 guarantees that AC termination of $\mathcal{H}$ follows from AC termination of well-sorted terms over $\mathcal{F}'$. We show the latter by constructing a suitable simple monotone algebra.

Consider the many-sorted algebra $\mathcal{A}$ with carriers $\mathsf{O}_{\mathcal{A}} = (\mathbb{O} \setminus \{0,1\}) \times \mathbb{N} \times \mathbb{N}$ and $\mathsf{N}_{\mathcal{A}} = (\mathbb{N} \setminus \{0,1\}) \times \mathbb{N} \times \mathbb{N}$ and the following interpretation functions, where we write $\vec{n}$ for $(n_1,n_2,n_3) \in \mathsf{N}_{\mathcal{A}}$ and $\vec{x}$ and $\vec{y}$ for $(x_1,x_2,x_3),(y_1,y_2,y_3) \in \mathsf{O}_{\mathcal{A}}$:

$$\begin{aligned}
\mathsf{0}_{\mathcal{A}} = \mathsf{h}_{\mathcal{A}} &= (2,0,0) & \mathsf{A}_{\mathcal{A}}(\vec{n},\vec{x}) &= (n_1 + x_1, n_2 + 2x_2 + 2, 0) \\
\mathsf{s}_{\mathcal{A}}(\vec{n}) &= (n_1 + 2, 0, 0) & \mathsf{B}_{\mathcal{A}}(\vec{n},\vec{x}) &= (2 + n_1 + x_1, n_2 + 2x_2 + 1, 0) \\
\mathsf{i}_{\mathcal{A}}(\vec{x}) &= (\omega^{x_1}, x_2 + 1, x_3 + 1) & \mathsf{C}_{\mathcal{A}}(\vec{n},\vec{x}) &= (x_1 \cdot n_1, 0, 0) \\
\vec{x} \mathrel{|_{\mathcal{A}}} \vec{y} &= (x_1 \oplus y_1, x_2 + y_2, x_3 + y_3) & \mathsf{E}_{\mathcal{A}}(\vec{x}) &= (x_1, x_2 + 1, 0) \\
\mathsf{D}_{\mathcal{A}}(\vec{n},\vec{x}) &= (n_1 + x_1, n_2 + x_2, n_2 + n_3 + x_2 + x_3)
\end{aligned}$$

The carriers $\mathsf{O}_{\mathcal{A}}$ and $\mathsf{N}_{\mathcal{A}}$ are equipped with the lexicographic order $>_{\mathsf{O}}$ on $\mathsf{O}_{\mathcal{A}}$. We write $>_{\mathsf{N}}$ for the restriction of $>_{\mathsf{O}}$ to $\mathsf{N}_{\mathcal{A}}$. The first component in the interpretation is used to represent the ordinal value of the Hydra that is encoded in a term. Since natural addition ($\oplus$) on ordinals is associative and commutative, AC equivalent term representations of Hydras have the same interpretation. The third component in vectors keeps track of the number of i symbols that do not occur below E. First we argue that $\mathcal{A}$ is simple monotone and $|_{\mathcal{A}}$ is strictly monotone. Due to lack of space, we only treat $\mathsf{C}_{\mathcal{A}}$.

- $\mathsf{C}_{\mathcal{A}}(\vec{n},\vec{x}) \geqslant_{\mathsf{O}} \mathsf{C}_{\mathcal{A}}(\vec{m},\vec{x})$ holds if $\vec{n} >_{\mathsf{N}} \vec{m}$, because $(x_1 \cdot n_1, 0, 0) \geqslant_{\mathsf{O}} (x_1 \cdot m_1, 0, 0)$ follows from $x_1 \cdot n_1 \geqslant_{\mathsf{O}} x_1 \cdot m_1$.

- $\mathsf{C}_{\mathcal{A}}(\vec{n},\vec{x}) \geqslant_{\mathsf{O}} \mathsf{C}_{\mathcal{A}}(\vec{n},\vec{y})$ holds if $\vec{x} >_{\mathsf{O}} \vec{y}$, because $(x_1 \cdot n_1, 0, 0) \geqslant_{\mathsf{O}} (y_1 \cdot n_1, 0, 0)$ follows from $x_1 \cdot n_1 >_{\mathsf{O}} y_1 \cdot n_1$. Note that the condition $n_1 \in \mathbb{N} \setminus \{0\}$ is essential as seen by $3 \cdot 0 = 0 \not>_{\mathsf{O}} 0 = 2 \cdot 0$.
- $\mathsf{C}_{\mathcal{A}}(\vec{n},\vec{x}) >_{\mathsf{O}} \vec{n}$ and $\mathsf{C}_{\mathcal{A}}(\vec{n},\vec{x}) >_{\mathsf{O}} \vec{x}$ hold, because $(x_1 \cdot n_1, 0, 0) \geqslant_{\mathsf{O}} (n_1, n_2, n_3)$ and $(x_1 \cdot n_1, 0, 0) \geqslant_{\mathsf{O}} (x_1, x_2, x_3)$ follow from $x_1 \cdot n_1 > n_1$ and $x_1 \cdot n_1 > x_1$. Note that the conditions $x_1 \notin \{0,1\}$ and $n_1 \in \mathbb{N} \setminus \{0,1\}$ are essential as seen by $1 \cdot 1 = 1 \not>_{\mathsf{O}} 1$ and $2 \cdot \omega = \omega \not>_{\mathsf{O}} \omega$.

Next we argue that $\mathcal{A}$ is compatible with $\mathcal{H}$. For brevity we treat only rules 3, 5, 7, 13 and omit unimportant elements in vectors.

- $\mathsf{A}_{\mathcal{A}}(\vec{n},\mathsf{i}(\vec{x})) = (\omega^{x_1}, n_2 + 2x_2 + 4, -) >_{\mathsf{O}} (\omega^{x_1}, n_2 + 2x_2 + 3, -) = \mathsf{B}_{\mathcal{A}}(\vec{n}, \mathsf{D}_{\mathcal{A}}(\mathsf{s}_{\mathcal{A}}(\vec{n}), \mathsf{i}_{\mathcal{A}}(\vec{x})))$
  Here ω^{x_1} is obtained from $n + \omega^{x_1} = \omega^{x_1}$ and $n + 2 + \omega^{x_1} = \omega^{x_1}$ by exploiting the condition $n_1 \in \mathbb{N}$.
- $\mathsf{C}_{\mathcal{A}}(\mathsf{s}_{\mathcal{A}}(\vec{n}),\vec{x}) = (x_1 \cdot (n_1+2), -, -) >_{\mathsf{O}} (x_1 \cdot (n_1+1), -, -) = \vec{x} \,|_{\mathcal{A}}\, \mathsf{C}_{\mathcal{A}}(\vec{n},\vec{x})$
  Remark that $\mathsf{s}_{\mathcal{A}}(\vec{n})$ is defined as $(n_1+2, 0, 0)$ rather than $(n_1+1, 0, 0)$.
- $\mathsf{i}_{\mathcal{A}}(\mathsf{E}_{\mathcal{A}}(\vec{x})) = (\omega^{x_1}, x_2 + 2, 1) >_{\mathsf{O}} (\omega^{x_1}, x_2 + 2, 0) = \mathsf{E}_{\mathcal{A}}(\mathsf{i}_{\mathcal{A}}(\vec{x}))$
- $\mathsf{D}_{\mathcal{A}}(\vec{n}, \mathsf{i}_{\mathcal{A}}(\mathsf{i}_{\mathcal{A}}(\mathsf{h}_{\mathcal{A}}))) = (\omega^{\omega^2}, -, -) >_{\mathsf{O}} (\omega^{2n_1}, -, -) = \mathsf{i}_{\mathcal{A}}(\mathsf{C}_{\mathcal{A}}(\vec{n}, \mathsf{h}_{\mathcal{A}}))$
  It follows from $\omega^2 >_{\mathsf{O}} 2n_1$ due to $n_1 \in \mathbb{N}$. This is the reason why we rely on Theorem 11.

▶ **Theorem 12.** *The* TRS $\mathcal{H}$ *is* AC *terminating.*

From Theorems 4 and 12 we conclude that Hercules eventually beats Hydra in any battle.

References

1. W. Buchholz. Another rewrite system for the standard Hydra battle. In *Proc. Mini-Workshop: Logic, Combinatorics and Independence Results*, volume 3(4) of *Oberwolfach Reports*, pages 3099–3102. European Mathematical Society, 2006.
2. N. Dershowitz and J.-P. Jouannaud. *Rewrite Systems. Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, 1990.
3. N. Dershowitz and G. Moser. The Hydra battle revisited. In *Rewriting, Computation and Proof, Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of his 60th Birthday*, volume 4600 of *Lecture Notes in Computer Science*, pages 1–27, 2007. `doi:10.1007/978-3-540-73147-4_1`.
4. M. C. F. Ferreira and H. Zantema. Total termination of term rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 7(2):133–162, 1996. `doi:110.1007/BF0119138`.
5. L. Kirby and J. Paris. Accessible independency results for Peano arithmetic. *Bulletin of the London Mathematical Society*, 14:285–325, 1982. `doi:10.1112/blms/14.4.285`.
6. A. Middeldorp and H. Ohsaki. Type introduction for equational rewriting. *Acta Informatica*, 36(12):1007–1029, 2000. `doi:10.1007/PL00013300`.
7. G. Moser. The Hydra battle and Cichon's principle. *Applicable Algebra in Engineering, Communication and Computing*, 20(2):133–158, 2009. `doi:10.1007/s00200-009-0094-4`.
8. H. Touzet. Encoding the Hydra battle as a rewrite system. In *Proc. 23rd International Symposium on Mathematical Foundations of Computer Science*, volume 1450 of *Lecture Notes in Computer Science*, pages 267–276, 1998. `doi:10.1007/BFb0055776`.
9. H. Zankl, S. Winkler, and A. Middeldorp. Beyond polynomials and Peano arithmetic—Automation of elementary and ordinal interpretations. *Journal of Symbolic Computation*, 69:129–158, 2015. `doi:10.1016/j.jsc.2014.09.033`.
10. H. Zantema. Termination of term rewriting: Interpretation and type elimination. *Journal of Symbolic Computation*, 17(1):23–50, 1994. `doi:10.1006/jsco.1994.1003`.
11. H. Zantema. The termination hierarchy for term rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 12(1-2):3–19, 2001. `doi:10.1007/s002000100061`.

A Calculus for Modular Non-Termination Proofs by Loop Acceleration

Florian Frohn[1]
LuFG Informatik 2, RWTH Aachen University, Germany

Carsten Fuhs
Department of Computer Science and Information Systems, Birkbeck, University of London, London, UK

Abstract

Recently, a calculus to combine various techniques for *loop acceleration* in a modular way has been introduced [5]. We show how to adapt this calculus for proving non-termination. An empirical evaluation demonstrates the applicability of our approach.

1 Introduction

In the last years, loop acceleration techniques have successfully been used to build static analyses for programs operating on integers. Essentially, such techniques extract a quantifier-free first-order formula ψ from a single-path loop $\mathcal{T}$, i.e., a loop without branching in its body, such that ψ under-approximates (or is equivalent to) $\mathcal{T}$. Recently, a calculus which allows for combining several acceleration techniques modularly in order to accelerate a single loop has been introduced [5]. As already observed in [7], certain properties of loops – in particular monotonicity of (parts of) the loop condition w.r.t. the loop body – are crucial for both acceleration and proving non-termination. In this paper, we take the next step by adapting the calculus from [5] for proving non-termination. To this end, we identify acceleration techniques that, if applied in isolation, give rise to non-termination proofs. Furthermore, we show that the combination of such non-termination techniques via the calculus from [5] gives rise to non-termination proofs, too. In this way, we obtain a modular framework for combining several different non-termination techniques in order to prove non-termination of a single loop. See [6] for an extended version of the current paper, including all proofs.

2 Preliminaries

We use the notation $\vec{x}$, $\vec{y}$, $\vec{z}$, ... for vectors. We consider loops of the form

$$\textbf{while } \varphi \textbf{ do } \vec{x} \leftarrow \vec{a} \qquad (\mathcal{T}_{loop})$$

where $\vec{x}$ contains d pairwise different variables over $\mathbb{Z}$, the loop condition $\varphi \in Conj(\vec{x})$ is a conjunction of polynomial inequations $p(\vec{x}) > 0$ over $\vec{x}$, and $\vec{a} \in \mathbb{Z}[\vec{x}]^d$. *Loop* denotes the set of all such loops.

We identify $\mathcal{T}_{loop}$ and the pair $\langle \varphi, \vec{a} \rangle$. Moreover, we identify $\vec{a}$ and the function $\vec{x} \mapsto \vec{a}$, where we sometimes write $\vec{a}(\vec{x})$ to make the variables $\vec{x}$ explicit. We use the same convention for other (vectors of) expressions. Similarly, we identify the formula $\varphi(\vec{x})$ (or just φ) with the predicate $\vec{x} \mapsto \varphi$. Our goal is to prove *non-termination* of $\mathcal{T}_{loop}$.

▶ **Definition 1** ((Non-)Termination). *We call a vector $\vec{x} \in \mathbb{Z}^d$ a* witness of non-termination *for $\mathcal{T}_{loop}$ (denoted $\vec{x} \longrightarrow^{\infty}_{\langle \varphi, \vec{a} \rangle} \perp$) if $\varphi(\vec{a}^n(\vec{x}))$ holds for all $n \in \mathbb{N}$. Here, $\vec{a}^n$ is the n-fold application of $\vec{a}$, i.e., $\vec{a}^0(\vec{x}) = \vec{x}$ and $\vec{a}^{n+1}(\vec{x}) = \vec{a}(\vec{a}^n(\vec{x}))$. If there is such a witness, then $\mathcal{T}_{loop}$ is* non-terminating. *Otherwise, $\mathcal{T}_{loop}$* terminates.

To find a witness of non-termination, we search for a *certificate of non-termination*.

[1] This work has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2) and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 389792660 as part of TRR 248.

▶ **Definition 2** (Certificate of Non-Termination). *We call a formula $\eta \in Conj(\vec{x})$ a* certificate of non-termination *for $\mathcal{T}_{loop}$ if it is satisfiable and the implication $\eta(\vec{x}) \implies \vec{x} \longrightarrow^{\infty}_{\langle \varphi, \vec{a} \rangle} \perp$ is valid.*

3 Proving Non-Termination via Loop Acceleration

In [5], several techniques for *loop acceleration* were discussed. For example, *Acceleration via Monotonic Increase* applies if $\varphi(\vec{x}) \implies \varphi(\vec{a}(\vec{x}))$ is valid and *Acceleration via Eventual Increase* applies if $e(\vec{x}) \leq e(\vec{a}(\vec{x})) \implies e(\vec{a}(\vec{x})) \leq e(\vec{a}^2(\vec{x}))$ holds for each inequation $e(\vec{x}) > 0$ in φ. It is not difficult to see that loops where these acceleration techniques apply are usually non-terminating, i.e., these techniques give rise to certificates of non-termination. More interestingly, the same holds if a loop can be accelerated using the calculus from [5], as long as all steps use one of these acceleration techniques. Thus, we obtain a novel, modular technique for proving non-termination of loops $\mathcal{T}_{loop}$.

Attempts to prove non-termination operate on a variation of the *acceleration problems* from [5], which we call *non-termination problems*.

▶ **Definition 3** (Non-Termination Problem). *A tuple $\|\psi \mid \check{\varphi} \mid \widehat{\varphi}\|_{\vec{a}}$ where $\psi, \check{\varphi}, \widehat{\varphi} \in Conj(\vec{x})$ and $\vec{a} : \mathbb{Z}^d \to \mathbb{Z}^d$ is a* non-termination problem. *It is* consistent *if every model of ψ is a witness of non-termination for $\langle \check{\varphi}, \vec{a} \rangle$ and* solved *if it is consistent and $\widehat{\varphi} \equiv \top$. The* canonical non-termination problem *of a loop $\mathcal{T}_{loop}$ is $\|\top \mid \top \mid \varphi\|_{\vec{a}}$.*

▶ **Example 4.** Consider the loop

$$\textbf{while } x_1 > 0 \textbf{ do } \left(\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right) \leftarrow \left(\begin{smallmatrix} x_1 + x_2 \\ x_2 + 1 \end{smallmatrix}\right). \qquad (\mathcal{T}_{ev\text{-}inc})$$

Its canonical non-termination problem is $\|\top \mid \top \mid x_1 > 0\|_{\left(\begin{smallmatrix} x_1 + x_2 \\ x_2 + 1 \end{smallmatrix}\right)}$, which is consistent as $\langle \check{\varphi}, \vec{a} \rangle = \langle \top, \left(\begin{smallmatrix} x_1 + x_2 \\ x_2 + 1 \end{smallmatrix}\right) \rangle$ diverges for all valuations of x_1 and x_2, but not solved as $\widehat{\varphi} \equiv x_1 > 0 \not\equiv \top$.

The first component ψ of a non-termination problem $\|\psi \mid \check{\varphi} \mid \widehat{\varphi}\|_{\vec{a}}$ is the partial result that has been computed so far. The second component $\check{\varphi}$ corresponds to the part of the loop condition that has already been processed successfully. As our calculus preserves consistency, ψ is always a certificate of non-termination for $\langle \check{\varphi}, \vec{a} \rangle$. The third component is the part of the loop condition that remains to be processed, i.e., we still need to find a certificate of non-termination for the loop $\langle \widehat{\varphi}, \vec{a} \rangle$. The goal of our calculus is to transform a canonical into a solved non-termination problem.

More specifically, when we have simplified a canonical non-termination problem $\|\top \mid \top \mid \varphi\|_{\vec{a}}$ to $\|\psi_1 \mid \check{\varphi} \mid \widehat{\varphi}\|_{\vec{a}}$, then $\varphi \equiv \check{\varphi} \land \widehat{\varphi}$ and $\psi_1 \implies \vec{x} \longrightarrow^{\infty}_{\langle \check{\varphi}, \vec{a} \rangle} \perp$. Then it suffices to find some $\psi_2 \in Conj(\vec{x})$ such that $\left(\vec{x} \longrightarrow^{\infty}_{\langle \check{\varphi}, \vec{a} \rangle} \perp \land \psi_2\right) \implies \vec{x} \longrightarrow^{\infty}_{\langle \widehat{\varphi}, \vec{a} \rangle} \perp$. The reason is that we have $\longrightarrow_{\langle \check{\varphi}, \vec{a} \rangle} \cap \longrightarrow_{\langle \widehat{\varphi}, \vec{a} \rangle} = \longrightarrow_{\langle \check{\varphi} \land \widehat{\varphi}, \vec{a} \rangle} = \longrightarrow_{\langle \varphi, \vec{a} \rangle}$ and thus $(\psi_1 \land \psi_2) \implies \vec{x} \longrightarrow^{\infty}_{\langle \varphi, \vec{a} \rangle} \perp$, i.e., $\psi_1 \land \psi_2$ is a certificate of non-termination for $\mathcal{T}_{loop}$.

We use a variation of the *conditional acceleration techniques* from [5], which we call *conditional non-termination techniques*, to simplify the canonical non-termination problem of the analyzed loop.

▶ **Definition 5** (Conditional Non-Termination Technique). *Let $nt : Loop \times Conj(\vec{x}) \rightharpoonup Conj(\vec{x})$ be a partial function. We call nt a* conditional non-termination technique *if*

$$\vec{x} \longrightarrow^{\infty}_{\langle \check{\varphi}, \vec{a} \rangle} \perp \land\, nt(\langle \chi, \vec{a} \rangle, \check{\varphi}) \quad \textit{implies} \quad \vec{x} \longrightarrow^{\infty}_{\langle \chi, \vec{a} \rangle} \perp \qquad \textit{for all } (\langle \chi, \vec{a} \rangle, \check{\varphi}) \in \mathrm{dom}(nt), \vec{x} \in \mathbb{Z}^d.$$

Thus, we obtain the following variation of the calculus from [5].

▶ **Definition 6** (Non-Termination Calculus). *The relation $\rightsquigarrow_{nt}$ on non-termination problems is defined as follows, where we identify conjunctions $e_1 > 0 \land \cdots \land e_n > 0$ and finite sets $\{e_1 > 0, \ldots, e_n > 0\}$:*

$$\frac{\chi \subseteq \widehat{\varphi} \quad nt(\langle \chi, \vec{a} \rangle, \check{\varphi}) = \psi_2}{\|\psi_1 \mid \check{\varphi} \mid \widehat{\varphi}\|_{\vec{a}} \rightsquigarrow_{nt} \|\psi_1 \land \psi_2 \mid \check{\varphi} \land \chi \mid \widehat{\varphi} \setminus \chi\|_{\vec{a}}} \; nt \textit{ is a conditional non-termination technique}$$

Since $\rightsquigarrow_{nt}$ preserves consistency, we obtain the following theorem, which shows that our calculus is indeed suitable for proving non-termination.

▶ **Theorem 7** (Correctness of $\rightsquigarrow_{nt}$). *If* $\|\top \mid \top \mid \varphi\|_{\vec{a}} \rightsquigarrow_{nt}^* \|\psi \mid \check{\varphi} \mid \top\|_{\vec{a}}$*, and* ψ *is satisfiable, then* ψ *is a certificate of non-termination for* $\mathcal{T}_{loop}$*.*

Moreover, termination of $\rightsquigarrow_{nt}$ is trivial, as each step removes an inequation from $\widehat{\varphi}$. It remains to present conditional non-termination techniques that can be used with our novel calculus. We first derive a conditional non-termination technique from *Acceleration via Monotonic Increase* [5].

▶ **Theorem 8** (Non-Termination via Monotonic Increase). *If* $\check{\varphi}(\vec{x}) \wedge \chi(\vec{x}) \implies \chi(\vec{a}(\vec{x}))$*, then* $(\langle\chi, \vec{a}\rangle, \check{\varphi}) \mapsto \chi$ *is a conditional non-termination technique.*

▶ **Example 9.** Consider the following loop:

$$\textbf{while } x > 0 \textbf{ do } x \leftarrow x + 1 \qquad (\mathcal{T}_{inc})$$

Its canonical non-termination problem is $\|\top \mid \top \mid x > 0\|_{(x+1)}$. Thus, in order to apply $\rightsquigarrow_{nt}$ with Thm. 8, the only possible choice for the formula χ is $x > 0$. Furthermore, we have $\check{\varphi} := \top$ and $\vec{a} := (x + 1)$. Hence, Thm. 8 is applicable if the implication $x > 0 \implies x + 1 > 0$ is valid, which is clearly the case. Thus, we get $\|\top \mid \top \mid x > 0\|_{(x+1)} \rightsquigarrow_{nt} \|x > 0 \mid x > 0 \mid \top\|_{(x+1)}$. Since the latter non-termination problem is solved and $x > 0$ is satisfiable, $x > 0$ is a certificate of non-termination for $\mathcal{T}_{inc}$ due to Thm. 7.

Clearly, Thm. 8 is only applicable in very simple cases. To prove non-termination of more complex examples, we now derive a conditional non-termination technique from *Acceleration via Eventual Increase* [5].

▶ **Theorem 10** (Non-Termination via Eventual Increase). *If*

$$\check{\varphi}(\vec{x}) \wedge e(\vec{x}) \leq e(\vec{a}(\vec{x})) \implies e(\vec{a}(\vec{x})) \leq e(\vec{a}^2(\vec{x})),$$

holds for all $e(\vec{x}) > 0 \in \chi$*, then the following function is a conditional non-termination technique:*

$$(\langle\chi, \vec{a}\rangle, \check{\varphi}) \mapsto \bigwedge_{e(\vec{x})>0\in\chi} 0 < e(\vec{x}) \leq e(\vec{a}(\vec{x}))$$

▶ **Example 11.** We continue Ex. 4. To apply $\rightsquigarrow_{nt}$ with Thm. 10 to the canonical non-termination problem of $\mathcal{T}_{ev\text{-}inc}$, the only possible choice for the formula χ is $x_1 > 0$. Moreover, we again have $\check{\varphi} := \top$, and $\vec{a} := \left(\begin{smallmatrix} x_1+x_2 \\ x_2+1 \end{smallmatrix}\right)$. Thus, Thm. 10 is applicable if $x_1 \leq x_1 + x_2 \implies x_1 + x_2 \leq x_1 + 2 \cdot x_2 + 1$ is valid. Since we have $x_1 \leq x_1 + x_2 \iff x_2 \geq 0$ and $x_1 + x_2 \leq x_1 + 2 \cdot x_2 + 1 \iff x_2 + 1 \geq 0$, this is clearly the case. Hence, we get $\|\top \mid \top \mid x_1 > 0\|_{\vec{a}} \rightsquigarrow_{nt} \|0 < x_1 \leq x_1 + x_2 \mid x_1 > 0 \mid \top\|_{\vec{a}}$. Since $0 < x_1 \leq x_1 + x_2 \equiv x_1 > 0 \wedge x_2 \geq 0$ is satisfiable, $x_1 > 0 \wedge x_2 \geq 0$ is a certificate of non-termination for $\mathcal{T}_{ev\text{-}inc}$ due to Thm. 7.

Of course, some non-terminating loops do not behave (eventually) monotonically.

▶ **Example 12.** Consider the loop

$$\textbf{while } x_1 > 0 \textbf{ do } \left(\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right) \leftarrow \left(\begin{smallmatrix} x_2 \\ x_1 \end{smallmatrix}\right). \qquad (\mathcal{T}_{fixpoint})$$

Thm. 8 is inapplicable, since $x_1 > 0 \not\Longrightarrow x_2 > 0$. Furthermore, Thm. 10 is inapplicable, since $x_1 \leq x_2 \not\Longrightarrow x_2 \leq x_1$.

However, $\mathcal{T}_{fixpoint}$ has *fixpoints*, i.e., there are valuations such that $\vec{x} = \vec{a}(\vec{x})$. Therefore, it can be handled by existing approaches like [7, Thm. 12]. As the following theorem shows, such techniques can also be embedded into our calculus.

▶ **Theorem 13** (Non-Termination via Fixpoints). *For each expression* e*, let* $\mathcal{V}(e)$ *denote the set of variables occurring in* e*. Moreover, we define* $closure_{\vec{a}}(e) := \bigcup_{n\in\mathbb{N}} \mathcal{V}(\vec{a}^n(e))$*. Then*

$$(\langle\chi, \vec{a}\rangle, \check{\varphi}) \mapsto \bigwedge_{e(\vec{x})>0\in\chi} e(\vec{x}) > 0 \quad \wedge \bigwedge_{x_j \in closure_{\vec{a}}(\chi)} x_j = \vec{a}(\vec{x})_j$$

is a conditional non-termination technique.

▶ **Example 14.** We continue Thm. 12 by showing how to apply Thm. 13 to $\mathcal{T}_{\mathit{fixpoint}}$, i.e., we have $\chi := x_1 > 0$, $\breve{\varphi} := \top$, and $\vec{a} := \left(\begin{smallmatrix} x_2 \\ x_1 \end{smallmatrix}\right)$. Thus, we get $\mathit{closure}_{\vec{a}}(x_1 > 0) = \{x_1, x_2\}$. So starting with the canonical non-termination problem of $\mathcal{T}_{\mathit{fixpoint}}$, we get $\|\top \mid \top \mid x_1 > 0\|_{\left(\begin{smallmatrix} x_2 \\ x_1 \end{smallmatrix}\right)} \rightsquigarrow_{nt} \|x_1 > 0 \land x_1 = x_2 \mid x_1 > 0 \mid \top\|_{\left(\begin{smallmatrix} x_2 \\ x_1 \end{smallmatrix}\right)}$. Since the formula $x_1 > 0 \land x_1 = x_2$ is satisfiable, it is a certificate of non-termination for $\mathcal{T}_{\mathit{fixpoint}}$ by Thm. 7.

Clearly, the conditional non-termination techniques from Thms. 10 and 13 can yield unsatisfiable formulas. Thus, when integrating these techniques into our calculus, one needs to check the resulting formula for satisfiability after each step to detect fruitless derivations early.

We conclude this section with a more complex example, which shows how the presented conditional non-termination techniques can be combined to find certificates of non-termination.

▶ **Example 15.** Consider the following loop:

$$\textbf{while } x_1 > 0 \land x_3 > 0 \land x_4 + 1 > 0 \textbf{ do } \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} \leftarrow \begin{pmatrix} 1 \\ x_2 + x_1 \\ x_3 + x_2 \\ -x_4 \end{pmatrix}$$

So we have $\varphi := x_1 > 0 \land x_3 > 0 \land x_4 + 1 > 0$ and $\vec{a} := \begin{pmatrix} 1 \\ x_2 + x_1 \\ x_3 + x_2 \\ -x_4 \end{pmatrix}$. Then the canonical non-termination problem is $\|\top \mid \top \mid x_1 > 0 \land x_3 > 0 \land x_4 + 1 > 0\|_{\vec{a}}$. First, our implementation applies Thm. 8 to $x_1 > 0$ (as $x_1 > 0 \implies 1 > 0$), resulting in $\|x_1 > 0 \mid x_1 > 0 \mid x_3 > 0 \land x_4 + 1 > 0\|_{\vec{a}}$. Next, it applies Thm. 10 to $x_3 > 0$, which is possible since $x_1 > 0 \land x_3 \leq x_3 + x_2 \implies x_3 + x_2 \leq x_3 + 2 \cdot x_2 + x_1$ is valid. Note that this implication breaks if one removes $x_1 > 0$ from the premise, i.e., Thm. 10 does not apply to $x_3 > 0$ without applying Thm. 8 to $x_1 > 0$ before. This shows that our calculus is more powerful than "the sum" of the underlying conditional non-termination techniques. Hence, we obtain the non-termination problem $\|x_1 > 0 \land x_2 \geq 0 \land x_3 > 0 \mid x_1 > 0 \land x_3 > 0 \mid x_4 + 1 > 0\|_{\vec{a}}$ Here, we simplified $0 < x_3 \leq x_3 + x_2$ to $x_2 \geq 0 \land x_3 > 0$. Finally, neither Thm. 8 nor Thm. 10 applies to $x_4 + 1 > 0$, since x_4 does not behave (eventually) monotonically: Its value after n iterations is given by $(-1)^n \cdot x_4^{\mathit{init}}$, where x_4^{init} denotes the initial value of x_4. Thus, we apply Thm. 13 and we get $\|x_1 > 0 \land x_2 \geq 0 \land x_3 > 0 \land x_4 = 0 \mid \varphi \mid \top\|_{\vec{a}}$, which is solved. Here, we simplified the subformula $x_4 + 1 > 0 \land x_4 = -x_4$ that results from the last acceleration step to $x_4 = 0$.

This shows that our calculus allows for applying Thm. 13 to loops that do not have a fixpoint. The reason is that it suffices to require that a *subset* of the program variables remain unchanged, whereas the values of other variables may still change.

As $x_1 > 0 \land x_2 \geq 0 \land x_3 > 0 \land x_4 = 0$ is satisfiable, it is a certificate of non-termination by Thm. 7.

4 Experiments and Conclusion

We implemented our approach in our open-source tool LoAT.[2] It uses Z3 [10] and Yices2 [4] to check implications. To evaluate our approach, we used the examples from the evaluation of [5]. All tests have been run on StarExec [11]. To prove non-termination, our implementation applies the conditional non-termination techniques from Sec. 3 with the following priorities: Thm. 8 > Thm. 10 > Thm. 13. We compared our implementation in LoAT with several leading tools for proving non-termination of integer programs: AProVE [8], iRankFinder [1], RevTerm [2], Ultimate [3], and VeryMax [9]. We used a timeout of 60s for each tool.

	LoAT	AProVE	iRankFinder	RevTerm	Ultimate	VeryMax
NO	206	200	205	133	205	175
YES	0	1301	1298	0	965	1299
fail	1305	10	8	1378	341	37
avg rt	0.04s	16.09s	1.40s	39.31s	21.71s	3.17s
avg rt NT	0.02s	10.65s	1.34s	4.55s	8.27s	14.52s

[2] https://aprove-developers.github.io/LoAT/

The results can be seen in the table above. They show that our novel calculus is competitive with state-of-the-art tools. Both iRankFinder and Ultimate can prove non-termination of precisely the same 205 examples. LoAT can prove non-termination of these examples, too. In addition, it solves one benchmark that cannot be handled by any other tool:

$$\textbf{while } x > 9 \land x_1 \geq 0 \textbf{ do } \left(\begin{smallmatrix} x \\ x_1 \end{smallmatrix}\right) \leftarrow \left(\begin{smallmatrix} x_1^2+2\cdot x_1+1 \\ x_1+1 \end{smallmatrix}\right)$$

We conjecture that the other tools fail for this example due to the presence of non-linear arithmetic. Our calculus from Sec. 3 just needs to check implications, so as long as the underlying SMT-solver supports non-linearity, it can be applied to non-linear examples, too.

For more details on our experiments, our benchmark collection, more details about the results of our evaluation, and a pre-compiled binary (Linux, 64 bit) we refer to [6].

Conclusion

We showed how the calculus from [5] can be adapted for proving non-termination, and we presented three non-termination techniques that can be combined with our novel calculus. Our experiments show that our approach is competitive in practice.

References

1. Amir M. Ben-Amram, Jesús J. Doménech, and Samir Genaim. Multiphase-linear ranking functions and their relation to recurrent sets. In *SAS '19*, LNCS 11822, pages 459–480, 2019. doi:10.1007/978-3-030-32304-2_22.
2. Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotný, and Dorde Zikelic. Proving non-termination by program reversal. In *PLDI '21*, pages 1033–1048, 2021. doi:10.1145/3453483.3454093.
3. 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 '18*, pages 135–150, 2018. doi:10.1145/3192366.3192405.
4. Bruno Dutertre. Yices 2.2. In *CAV '14*, LNCS 8559, pages 737–744, 2014. doi:10.1007/978-3-319-08867-9_49.
5. Florian Frohn. A calculus for modular loop acceleration. In *TACAS '20*, LNCS 12078, pages 58–76, 2020. doi:10.1007/978-3-030-45190-5_4.
6. Florian Frohn and Carsten Fuhs. A calculus for modular loop acceleration and non-termination proofs, 2021. URL: https://arxiv.org/abs/2111.13952.
7. Florian Frohn and Jürgen Giesl. Proving non-termination via loop acceleration. In *FMCAD '19*, pages 221–230, 2019. doi:10.23919/FMCAD.2019.8894271.
8. 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. *J. Autom. Reasoning*, 58(1):3–31, 2017. doi:10.1007/s10817-016-9388-y.
9. Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving non-termination using max-SMT. In *CAV '14*, LNCS 8559, pages 779–796, 2014. doi:10.1007/978-3-319-08867-9_52.
10. Leonardo de Moura and Nikolay Bjørner. Z3: An efficient SMT solver. In *TACAS '08*, LNCS 4963, pages 337–340, 2008. doi:10.1007/978-3-540-78800-3_24.
11. Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A cross-community infrastructure for logic solving. In *IJCAR '14*, LNCS 8562, pages 367–373, 2014. doi:10.1007/978-3-319-08587-6_28.

Deciding Termination of Uniform Loops with Polynomial Parameterized Complexity

Marcel Hark
Florian Frohn
Jürgen Giesl
LuFG Informatik 2, RWTH Aachen University, Aachen, Germany

Abstract

In [3, 4] we showed that for so-called triangular weakly non-linear loops over rings $\mathcal{S}$ like $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$, the question of termination can be reduced to the existential fragment of the first-order theory of $\mathcal{S}$. For loops over $\mathbb{R}$, our reduction implies decidability of termination. For loops over $\mathbb{Z}$ and $\mathbb{Q}$, it proves semi-decidability of non-termination. In this paper, we show that there is an important class of linear loops where our decision procedure results in an *efficient* procedure for termination analysis, i.e., where the parameterized complexity of deciding termination is *polynomial*.

2012 ACM Subject Classification Theory of computation → Program analysis; Theory of computation → Parameterized complexity and exact algorithms

Keywords and phrases Termination, Linear Loops, Decision Procedure, Complexity, Closed Form

Funding funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)-235950644 (Project GI 274/6-2), by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)-389792660 as part of TRR 248, and by the DFG Research Training Group 2236 UnRAVeL

1 Introduction

In [3, 4], we showed that termination of linear loops whose update matrix only has rational eigenvalues is Co-NP-complete. In this paper, we present a special case of linear loops (so-called *uniform* loops) and show that for these loops deciding termination is *polynomial*, if one fixes the number of eigenvalues of the update matrix.

In Sect. 2, we introduce uniform loops and state our main result (Thm. 3). To prove it, we use our decision procedure from [3, 4] which instantiates the variables in the loop guard by *closed forms* for the iterated update of the loop (Sect. 3). For uniform loops, this results in formulas of a special structure that can be checked in polynomial time (Sect. 4).

2 Uniform Loops and the Parameterized Complexity Class XP

A *linear loop over a ring* $\mathcal{S}$ has the form **while** (φ) **do** $\vec{x} \leftarrow A \cdot \vec{x}$ (or $(\varphi, A \cdot \vec{x})$ for short). Here, $\vec{x}$ is a vector of $d \geq 1$ pairwise different variables that range over $\mathcal{S}$ and A is a $d \times d$ matrix over $\mathcal{S}$. For loops **while** (φ) **do** $\vec{x} \leftarrow A \cdot \vec{x} + \vec{c}$, note that $\vec{c} \in \mathcal{S}^d$ can be eliminated by introducing an additional variable. The guard φ is a quantifier-free formula over the atoms $\{f \triangleright 0 \mid f \in \mathcal{S}[\vec{x}]_{\text{lin}}, \triangleright \in \{\geq, >\}\}$, where $\mathcal{S}[\vec{x}]_{\text{lin}}$ contains all polynomials of degree at most 1.

▶ **Definition 1** (Uniform Loop). *A linear loop* $(\varphi, A \cdot \vec{x})$ *over* $\mathcal{S} \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}_{\mathbb{A}}\}$ *(where* $\mathbb{R}_{\mathbb{A}}$ *are the real algebraic numbers) is* uniform *if each eigenvalue* λ *of* A *is a* non-negative *number from* $\mathcal{S}$ *whose eigenspace w.r.t.* A *is one-dimensional, i.e.,* λ *has* geometric multiplicity 1.

The latter property is equivalent to requiring that there is exactly one Jordan block for each eigenvalue in A's Jordan normal form. Fig. 1 shows an example for a uniform loop. In contrast, a loop which updates each x_i to $\lambda \cdot x_i$ is not uniform. To give an intuition how hard the restriction to uniform loops is, the *TPDB* category for "Termination of Integer Transition

$$\textbf{while } (\varphi) \textbf{ do} \quad \vec{x} \leftarrow A \cdot \vec{x} \qquad A = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 2 \end{bmatrix} \qquad \vec{q} = \begin{bmatrix} x_1 + n \cdot x_2 \\ x_2 \\ x_3 \cdot 2^n + \left(\frac{x_4}{2} - \frac{x_5}{8}\right) \cdot n \cdot 2^n + \frac{x_5}{8} \cdot n^2 \cdot 2^n \\ x_4 \cdot 2^n + \frac{x_5}{2} \cdot n \cdot 2^n \\ x_5 \cdot 2^n \end{bmatrix}$$

Figure 1 Uniform Loop and its Normalized Closed Form

Systems" at the *Termination and Complexity Competition* contains 467 polynomial loops with non-constant guard, where 290 (62 %) are uniform. We show that termination of uniform loops is in the *parameterized complexity class* XP.

▶ **Definition 2** (Parameterized Decision Problem, XP [2]). *A* parameterized decision problem *is a language $\mathcal{L} \subseteq \Sigma^* \times \mathbb{N}$, where Σ is a finite alphabet. The second component is the* parameter *of the problem. $\mathcal{L}$ is an element of the complexity class* XP *if the time needed for deciding the question "$(x, k) \in \mathcal{L}$?" is in $\mathcal{O}(|x|^{f(k)})$ where f is a computable function depending only on k.*

In the remainder, we show that for any fixed $k \in \mathbb{N}$, termination of uniform loops with k eigenvalues is decidable in polynomial time. For the full version of our paper, we refer to [4].

▶ **Theorem 3** (Parameterized Complexity of k-Termination). *We define the parameterized decision problem* k-termination *as follows: $((\varphi, A \cdot \vec{x}), k) \in \mathcal{L}_{k\text{-termination}}$ iff the loop $(\varphi, A \cdot \vec{x})$ terminates over $\mathcal{S}$ and A has k eigenvalues. For uniform loops, k-termination is in* XP.

Our result is surprising as it shows that for these loops, termination is easier to decide than satisfiability of the guard (e.g., unsatisfiability of linear formulas over $\mathbb{R}_{\mathbb{A}}$ is Co-NP-complete). Intuitively, our class *prohibits* multiple updates like $x_i \leftarrow x_i$ where variables "stabilize", as termination is essentially equivalent to unsatisfiability of the guard for such loops.

3 Deciding Termination

To decide termination, we first transform the uniform loop $(\varphi, A \cdot \vec{x})$ such that the update matrix is in Jordan normal form. Let $\lambda_1 < \ldots < \lambda_k$ be A's eigenvalues, let Q be A's Jordan normal form where the Jordan blocks are ordered such that the numbers on the diagonal are weakly monotonically increasing, and let T be the corresponding transformation matrix, i.e., $A = T^{-1} \cdot Q \cdot T$. Moreover, let η be the automorphism defined by $\eta(\vec{x}) = T \cdot \vec{x}$. As shown in [3, 4], instead of termination of the original loop on $\mathcal{S}^d$, we can prove termination of the transformed loop on $(\eta^{-1}(\varphi), Q \cdot \vec{x}) = (\varphi', Q \cdot \vec{x})$ on $T \cdot \mathcal{S}^d$. Here, $\eta^{-1}(\varphi)$ results from φ by applying η^{-1} to all polynomials that occur in φ. For $\mathcal{S} \in \{\mathbb{Q}, \mathbb{R}_{\mathbb{A}}\}$, the transformation matrix T is an invertible matrix over $\mathcal{S}$. Therefore, we obtain $T \cdot \mathcal{S}^d = \mathcal{S}^d$, i.e., we now have to analyze termination of $(\varphi', Q \cdot \vec{x})$ over $\mathcal{S}$. In contrast, if $\mathcal{S} = \mathbb{Z}$, then the transformation matrix T or its inverse may contain non-integer rational numbers. Thus, we focus on uniform loops over $\mathcal{S} \in \{\mathbb{Q}, \mathbb{R}_{\mathbb{A}}\}$ and refer to [4] for $\mathcal{S} = \mathbb{Z}$.

So we now assume that the update matrix of our loop $(\varphi, A \cdot \vec{x})$ is in Jordan normal form. As shown in [4], one can easily compute a *normalized closed form* $\vec{q}$ of A, i.e., $\vec{q}$ is a vector of d arithmetic expressions over $\vec{x}$ and a designated variable n such that $\vec{q} = A^n \cdot \vec{x}$ for all large enough $n \in \mathbb{N}$ (see Fig. 1 for such a $\vec{q}$ in our example). Then $(\varphi, A \cdot \vec{x})$ is non-terminating iff

$$\exists \vec{x} \in \mathcal{S}^d,\, n_0 \in \mathbb{N}.\ \forall n \in \mathbb{N}_{>n_0}.\ \varphi(\vec{q}) \qquad \text{is valid,} \tag{1}$$

where $\varphi(\vec{q}) = \varphi[\vec{x}/\vec{q}]$. To check this, we examine the dominant terms in $\varphi(\vec{q})$'s inequations.

Let $p \triangleright 0$ occur in $\varphi(\vec{q})$, where $\triangleright \in \{\geq, >\}$. Then we order p's addends according to their asymptotic growth w.r.t. n. Here, let $\mathcal{Q}_{\mathcal{S}} = \left\{ \frac{r}{s} \;\middle|\; r \in \mathcal{S}, s \in \mathcal{S}_{>0} \right\}$ be the quotient field of $\mathcal{S}$.

▶ **Definition 4** (Ordering Coefficients). Marked coefficients *are of the form $\alpha^{(b,a)}$ where $\alpha \in \mathcal{Q}_{\mathcal{S}}[\vec{x}]_{\text{lin}}, b \in \mathcal{S}_{>0}$, and $a \in \mathbb{N}$. We define* $\text{unmark}(\alpha^{(b,a)}) = \alpha$ *and $\alpha_1^{(b_1,a_1)} \prec_{coef} \alpha_2^{(b_2,a_2)}$ if*

$b_1 < b_2$ *or* $b_1 = b_2 \land a_1 < a_2$. *Let* $p = \sum_{j=1}^{\ell} \alpha_j \cdot n^{a_j} \cdot b_j^n$, *where* $\alpha_j \neq 0$ *for all* $1 \leq j \leq \ell$. *Then the marked coefficients of* p *are* $\operatorname{coefs}(p) = \{0^{(1,0)}\}$, *if* $\ell = 0$, *and* $\operatorname{coefs}(p) = \{\alpha_j^{(b_j,a_j)} \mid 1 \leq j \leq \ell\}$ *otherwise. W.l.o.g., let* $\alpha_i^{(b_i,a_i)} \prec_{coef} \alpha_j^{(b_j,a_j)}$ *for all* $1 \leq i < j \leq \ell$.

▶ **Example 5.** In Fig. 1, let $\varphi = f > 0 \land f' > 0$ for $f = -x_1 + 3 \cdot x_3 + 4$ and $f' = 2 \cdot x_1 - 5$. So $\varphi(\vec{q}) = f(\vec{q}) > 0 \land f'(\vec{q}) > 0 = p > 0 \land p' > 0$. We have $p = f(q_1, \ldots, q_5) = -q_1 + 3 \cdot q_3 + 4 =$

$$(-x_1 + 4) - x_2 \cdot n + 3 \cdot x_3 \cdot 2^n + \left(\tfrac{3 \cdot x_4}{2} - \tfrac{3 \cdot x_5}{8}\right) \cdot n \cdot 2^n + \tfrac{3 \cdot x_5}{8} \cdot n^2 \cdot 2^n. \tag{2}$$

So $\operatorname{coefs}(p) = \{\alpha_1^{(1,0)}, \alpha_2^{(1,1)}, \alpha_3^{(2,0)}, \alpha_4^{(2,1)}, \alpha_5^{(2,2)}\}$, where $\alpha_1 = -x_1 + 4$, $\alpha_2 = -x_2$, $\alpha_3 = 3 \cdot x_3$, $\alpha_4 = \frac{3 \cdot x_4}{2} - \frac{3 \cdot x_5}{8}$, and $\alpha_5 = \frac{3 \cdot x_5}{8}$.

For $\vec{v} \in \mathcal{S}^d$, we have $p(\vec{v}) > 0$ for large enough values of n iff the coefficient of the asymptotically fastest growing addend $\alpha(\vec{v}) \cdot n^a \cdot b^n$ that does not vanish (i.e., where $\alpha(\vec{v}) \neq 0$) is *positive*. Similarly, we have $p(\vec{v}) < 0$ for large enough n iff $\alpha(\vec{v}) < 0$. If *all* addends of p vanish when instantiating $\vec{x}$ with $\vec{v}$, then $p(\vec{v}) = 0$. Thus, $\exists \vec{x} \in \mathcal{S}^d,\, n_0 \in \mathbb{N}.\ \forall n \in \mathbb{N}_{>n_0}.\ p \triangleright 0$ holds iff there is a $\vec{v} \in \mathcal{S}^d$ such that $\operatorname{unmark}\left(\max_{\succ_{coef}}(\operatorname{coefs}(p(\vec{v})))\right) \triangleright 0$. This is equivalent to satisfiability of $\operatorname{red}(p \triangleright 0)$, where

$$\operatorname{red}(p > 0) = \bigvee\nolimits_{i=1}^{\ell} \left(\alpha_i > 0 \land \bigwedge\nolimits_{j=i+1}^{\ell} \alpha_j = 0\right) \text{ and } \operatorname{red}(p \geq 0) = \operatorname{red}(p > 0) \lor \bigwedge\nolimits_{j=1}^{\ell} \alpha_i = 0.$$

▶ **Example 6.** We continue Ex. 5. Here, $\exists \vec{x} \in \mathcal{S}^d,\, n_0 \in \mathbb{N}.\ \forall n \in \mathbb{N}_{>n_0}.\ p \triangleright 0$ is valid iff $\exists x_1, x_2, x_3, x_4, x_5 \in \mathbb{Z}.\ \operatorname{red}(p > 0)$ is valid, where $\operatorname{red}(p > 0)$ is

$$\begin{aligned} & (\alpha_1 > 0 \land \alpha_2 = 0 \land \ldots \land \alpha_5 = 0) \lor (\alpha_2 > 0 \land \alpha_3 = 0 \land \ldots \land \alpha_5 = 0) \\ \lor\ & (\alpha_3 > 0 \land \alpha_4 = 0 \land \alpha_5 = 0) \lor (\alpha_4 > 0 \land \alpha_5 = 0) \lor \alpha_5 > 0. \end{aligned}$$

To lift our reduction to quantifier-free formulas ξ, let the formula $\operatorname{red}(\xi)$ result from ξ by replacing each atom $p \triangleright 0$ in ξ by $\operatorname{red}(p \triangleright 0)$. Then, we obtain the following decision procedure.

▶ **Theorem 7** (Deciding Termination). *A loop* $(\varphi, A \cdot \vec{x})$ *over* $\mathcal{S}$ *with normalized closed form* $\vec{q}$ *is non-terminating iff* $\exists \vec{x} \in \mathcal{S}^d.\ \operatorname{red}(\varphi(\vec{q}))$ *is valid.*

4 Interval Conditions

For uniform loops, $\operatorname{red}(p \triangleright 0)$ can be expressed as a disjunction of *interval conditions*, whose satisfiability is particularly easy to check. More precisely, for any $p = f(\vec{q})$ with $f \in \mathcal{S}[\vec{x}]_{\text{lin}}$ and any $1 \leq i \leq \ell$, in [4] we show how to construct an interval condition $\rho_{f,i}$ in polynomial time from $q_1, \ldots, q_d$ and f which is equivalent to $\alpha_i > 0 \land \bigwedge_{j=i+1}^{\ell} (\alpha_j = 0)$, and we also introduce an interval condition $\rho_{f,0}$ which is equivalent to $\bigwedge_{j=1}^{\ell} (\alpha_j = 0)$.

▶ **Example 8.** For $p = f(\vec{q}) =$ (2) from Ex. 5, $\operatorname{red}(p > 0)$ is equivalent to $\operatorname{ic}(p > 0) = \rho_{f,1} \lor \rho_{f,2} \lor \rho_{f,3} \lor \rho_{f,4} \lor \rho_{f,5} =$

$$\begin{aligned} & \left(-x_1 + 4 > 0 \land \bigwedge\nolimits_{j=2}^{5} x_j = 0\right) \lor \left(-x_2 > 0 \land \bigwedge\nolimits_{j=3}^{5} x_j = 0\right) \\ \lor\ & (x_3 > 0 \land x_4 = 0 \land x_5 = 0) \lor (x_4 > 0 \land x_5 = 0) \lor (x_5 > 0). \end{aligned}$$

The formulas $\rho_{f,s}$ are so-called *interval conditions*.

▶ **Definition 9** (Interval Condition). *For* $1 \leq i, i' \leq d$, $i \neq i'$, $I \subseteq \{1, \ldots, d\}$, $sg \in \{-1, 1\}$, *and* $0 \neq c \in \mathcal{Q}_{\mathcal{S}}$, *an* interval condition *has one of the following forms:*

$$\begin{array}{llll} (a) & & \bigwedge_{j \in I} & (x_j = 0) \\ (b) & sg \cdot x_i > 0 \ \land & \bigwedge_{j \in I \setminus \{i\}} & (x_j = 0) \end{array}$$

$$\begin{array}{llllll}
(c) & & & x_{i'} = c \;\wedge & \bigwedge_{j \in I \setminus \{i'\}} & (x_j = 0) \\
(d) & sg \cdot x_i & > 0 \;\wedge & x_{i'} = c \;\wedge & \bigwedge_{j \in I \setminus \{i,i'\}} & (x_j = 0) \\
(e) & sg \cdot x_i + c & > 0 \;\wedge & & \bigwedge_{j \in I \setminus \{i\}} & (x_j = 0)
\end{array}$$

To decide satisfiability of the formulas $\mathrm{ic}(p \triangleright 0)$, we only have to regard instantiations of the variables with values from $\{0, 1, \text{-}1, \star\}$, where $\star$ stands for one additional non-zero value.

▶ **Definition 10** (Evaluation). *Let ρ be a formula built from $\wedge$, $\vee$, and atoms of the form $sg \cdot x + c > 0$ and $x = c$ for $sg \in \{1, -1\}$, $c \in \mathcal{Q}_{\mathcal{S}}$, and $x \in \{x_1, \ldots, x_d\}$. Moreover, let $\vec{v} \in \{0, 1, \text{-}1, \star\}^d$. The* evaluation *of ρ w.r.t. $\vec{v}$ (written $\rho(\vec{v})\!\downarrow$) results from $\rho(\vec{v}) = \rho[\vec{x}/\vec{v}]$ by simplifying (in)equations without $\star$ to* true *or* false*, and by simplifying conjunctions and disjunctions with* true *resp.* false*. We write $\vec{v} \models^? \rho$ if $\rho(\vec{v})\!\downarrow \neq$ false.*

So if ρ is $(x_1 - \frac{5}{2} > 0) \wedge (x_2 = 0)$ and $\vec{v} = (\star, 0)$, then $\rho(\vec{v})\!\downarrow$ is $\star - \frac{5}{2} > 0$. Hence, $\vec{v} \models^? \rho$. Thus, we have $\vec{v} \models^? \rho$ whenever there *could* be a value w for $\star$ such that $\rho[\vec{x}/\vec{v}, \star/w]\!\downarrow =$ true.

Let $\nu_1, \ldots, \nu_k$ be the *algebraic multiplicities* of the eigenvalues $\lambda_1, \ldots, \lambda_k$. So in Fig. 1 we have $\lambda_1 = 1$, $\lambda_2 = 2$, $\nu_1 = 2$, and $\nu_2 = 3$. Then we define the *blocks* $B_{\lambda_1} = \{1, \ldots, \nu_1\}$, $B_{\lambda_2} = \{\nu_1 + 1, \ldots, \nu_1 + \nu_2\}$, …, and $B_{\lambda_k} = \{\nu_1 + \ldots + \nu_{k-1} + 1, \ldots, d\}$. Note that if $\lambda_1 = 0$, then all entries $q_1, \ldots, q_{\nu_1}$ of $\vec{q}$ are 0. Thus, we assume that 0 is not an eigenvalue of A.

Now we define candidate assignments $\mathrm{cndAssg}(\rho_{f,i})$ for the formulas $\rho_{f,i}$ which contain all $\vec{v} \in \{0, 1, \text{-}1, \star\}^d$ that *may* satisfy $\rho_{f,i}$ (if a suitable value for $\star$ is found). However, for each block B, *at most one* variable x_j with $j \in B$ may be assigned a non-zero value (i.e., 1, -1, or $\star$). Moreover, the value $\star$ may only be used in the block B_1 for the eigenvalue $\lambda = 1$.

▶ **Definition 11** (Sets of Candidate Assignments). *For all $0 \leq s \leq \ell$, we define* $\mathrm{cndAssg}(\rho_{f,s}) =$

$$\begin{array}{ll}\{\vec{v} \in \{0, 1, \text{-}1, \star\}^d \mid & \vec{v} \models^? \rho_{f,s}, \quad v_j = \star \implies j \in B_1, \\ & \forall\, B \in \{B_{\lambda_1}, \ldots, B_{\lambda_k}\}.\ \textit{there is at most one } j \in B \textit{ with } v_j \neq 0\}\end{array}$$

▶ **Example 12.** For $\rho_{f,4} = (x_4 > 0 \wedge x_5 = 0)$ in Ex. 8, $\vec{v} \models^? \rho_{f,4}$ implies $v_4 = 1$ and $v_5 = 0$. Here, $v_4 = \star$ is not possible, because 4 does not belong to the block $B_1 = \{1, 2\}$ for the eigenvalue 1. Since at most one value for each block may be non-zero, we have $v_3 = 0$. In contrast, v_1 and v_2 can be arbitrary, but at most one of them may be non-zero. Hence, we obtain the following for $\rho_{f,4}$ and for $\rho_{f,0} = (x_1 = 4) \wedge \bigwedge_{j=2}^{5} (x_j = 0)$:

$$\mathrm{cndAssg}(\rho_{f,4}) = \{ \begin{bmatrix}1\\0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}\star\\0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}\text{-}1\\0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\\star\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\\text{-}1\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\\0\end{bmatrix} \}, \qquad \mathrm{cndAssg}(\rho_{f,0}) = \{ \begin{bmatrix}\star\\0\\0\\0\\0\end{bmatrix} \}$$

We lift cndAssg to inequations by defining $\mathrm{cndAssg}(\mathrm{ic}(p > 0)) = \bigcup_{i=1}^{\ell} \mathrm{cndAssg}(\rho_{f,i})$ and $\mathrm{cndAssg}(\mathrm{ic}(p \geq 0)) = \mathrm{cndAssg}(\mathrm{ic}(p > 0)) \cup \mathrm{cndAssg}(\rho_{f,0})$. Then we have

$$|\mathrm{cndAssg}(\mathrm{ic}(p \triangleright 0))| \leq (d + 2) \cdot (3 \cdot \max\{\nu_i \mid 1 \leq i \leq k\} + 1)^k. \tag{3}$$

To analyze termination of uniform loops, we use Alg. 1 to decide whether (1) is valid (i.e., whether the uniform loop is non-terminating). Our algorithm computes $\mathrm{ic}(\varphi(\vec{q}))$ by replacing each atom $f(\vec{q}) \triangleright 0$ in $\varphi(\vec{q})$ (where $f \in \mathcal{S}[\vec{x}]_{\mathrm{lin}}$) by $\mathrm{ic}(f(\vec{q}) \triangleright 0)$, and then checks each candidate assignment from $\mathrm{cndAssg}(\mathrm{ic}(\varphi(\vec{q}))) = \bigcup_{f(\vec{q}) \triangleright 0 \text{ atom in } \varphi(\vec{q})} \mathrm{cndAssg}(\mathrm{ic}(f(\vec{q}) \triangleright 0))$.

To this end, it calls a method $\mathrm{SMT}(\psi, \mathcal{V}, \mathcal{S})$ which checks whether the linear formula ψ in the variables $\mathcal{V}$ is satisfiable. In our case, $\mathcal{V} = \{\star\}$ and thus, $|\mathcal{V}| = 1$. With this restriction, the method SMT has *polynomial* runtime (see [1, 5]). More precisely, SMT is called in Alg. 1 to determine whether $\star$ can be assigned a non-zero value such that $\psi(\vec{v})\!\downarrow$ is satisfiable. Here, we have to assign *all* occurrences of $\star$ in the formula $\psi(\vec{v})\!\downarrow$ the same value.

Input: a formula φ over the atoms $\{f \triangleright 0 \mid f \in \mathcal{S}[\vec{x}]_{\text{lin}}, \triangleright \in \{>, \geq\}\}$,
a normalized closed form $\vec{q}$, and a ring $\mathcal{S} \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}_{\mathbb{A}}\}$
Result: $\top$ if $\exists \vec{x} \in \mathcal{S}^d.\ \text{ic}(\varphi(\vec{q}))$ is valid, $\bot$ otherwise
$\psi \leftarrow \text{ic}(\varphi(\vec{q}))$
foreach $\vec{v} \in \text{cndAssg}(\psi)$ **do**
  $\psi' \leftarrow \psi(\vec{v})\!\downarrow$
  if $\text{SMT}((\psi' \land \star \neq 0), \{\star\}, \mathcal{S})$ **then return** $\top$
return $\bot$

Algorithm 1 Checking Interval Conditions

The formula $\text{ic}(\varphi(\vec{q}))$ and each element of $\text{cndAssg}(\text{ic}(\varphi(\vec{q})))$ can be computed in polynomial time. By (3), $\text{cndAssg}(\text{ic}(\varphi(\vec{q})))$ has at most $|\varphi| \cdot (d+2) \cdot (3 \cdot \max\{\nu_i \mid 1 \leq i \leq k\} + 1)^k$ elements, where $|\varphi|$ is the number of atoms in φ and $\nu_i \leq d$ for all $1 \leq i \leq k$. Thus, $\text{cndAssg}(\text{ic}(\varphi(\vec{q})))$ can be computed in polynomial time for fixed k. Moreover, evaluating a formula w.r.t. $\vec{v}$ according to Def. 10 is possible in polynomial time, too. So the runtime of Alg. 1 is polynomial when regarding k as a parameter.

▶ **Example 13.** Reconsider the loop in Fig. 1 and φ, p, p' in Ex. 5. Here, $\psi = \text{ic}(\varphi(\vec{q})) = \text{ic}(p > 0) \land \text{ic}(p' > 0)$, where $\text{ic}(p > 0) = \bigvee_{i=1}^{5} \rho_{f,i}$ as in Ex. 8. Note that $\text{coefs}(p') = \{\alpha_1'^{(1,0)}, \alpha_2'^{(1,1)}\}$ with $\alpha_1' = 2 \cdot x_1 - 5$ and $\alpha_2' = 2 \cdot x_2$. Moreover, $\text{ic}(p' > 0) = \rho_{f',1} \lor \rho_{f',2}$ with $\rho_{f',1} = (x_1 - \frac{5}{2} > 0) \land (x_2 = 0)$ and $\rho_{f',2} = (x_2 > 0)$. Consider $\vec{v} = (\star, 0, 0, 0, 0)$. Then $(\rho_{f,1} \land \rho_{f',1})(\vec{v})\!\downarrow \;=\; (-\star + 4 > 0) \land (\star - \frac{5}{2} > 0)$ is satisfiable with the model $\star = 3$. Hence, this model also satisfies $\psi(\vec{v})\!\downarrow \land \star \neq 0$. Thus, Alg. 1 proves validity of $\exists \vec{x} \in \mathcal{S}^d.\ \text{ic}(\varphi(\vec{q}))$ and therefore, non-termination of the uniform loop over $\mathcal{S}$ for both $\mathcal{S} = \mathbb{Q}$ and $\mathcal{S} = \mathbb{R}_{\mathbb{A}}$.

5 Conclusion

We presented an approach to decide termination of uniform loops over $\mathcal{S} \in \{\mathbb{Q}, \mathbb{R}_{\mathbb{A}}\}$ in polynomial time, if the number of eigenvalues of the update matrix is fixed. To this end, we first transform the uniform loop such that the update matrix is in Jordan normal form and then compute its normalized closed form. Afterwards, Alg. 1 can decide its termination.

The approach also works for uniform loops over $\mathbb{Z}$ if the update matrix is already in Jordan normal form. Otherwise, we have to analyze termination of the transformed loop $(\varphi', Q \cdot \vec{x})$ over $T \cdot \mathbb{Z}^d$. As shown in [4], this is also possible by a slight modification of Alg. 1. Thus, termination of uniform loops is in XP for all rings $\mathcal{S} \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}_{\mathbb{A}}\}$, i.e., our decision procedure can indeed be used as an efficient technique for termination analysis.

References

1 Saugata Basu, Richard Pollack, and Marie-Françoise Roy. *Algorithms in Real Algebraic Geometry*. Algorithms and Comp. in Math. 10. Springer, 2006. doi:10.1007/3-540-33099-2.
2 Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999. doi:10.1007/978-1-4612-0515-9.
3 Florian Frohn, Marcel Hark, and Jürgen Giesl. Termination of polynomial loops. In *Proc. SAS*, LNCS 12389, pages 89–112, 2020. doi:10.1007/978-3-030-65474-0_5.
4 Marcel Hark, Florian Frohn, and Jürgen Giesl. Termination of triangular polynomial loops. *CoRR*, abs/1910.11588, 2022. URL: https://arxiv.org/abs/1910.11588.
5 Hendrik W. Lenstra Jr. Integer programming with a fixed number of variables. *Math. Oper. Res.*, 8(4):538–548, 1983. doi:10.1287/moor.8.4.538.

Improved Automatic Complexity Analysis of Integer Programs

Jürgen Giesl
Nils Lommen
Marcel Hark
Fabian Meyer
LuFG Informatik 2, RWTH Aachen University, Aachen, Germany

Abstract

In former work [4], we developed an approach for automatic complexity analysis of integer programs, based on an alternating modular inference of upper runtime and size bounds for program parts. In this paper, we show how recent techniques to improve automated termination analysis of integer programs (like the generation of multiphase-linear ranking functions and control-flow refinement) can be integrated into our approach for the inference of runtime bounds. Our approach is implemented in the complexity analysis tool KoAT.

2012 ACM Subject Classification Theory of computation → Complexity classes; Theory of computation → Program analysis; Software and its engineering → Automated static analysis

Keywords and phrases Automatic Complexity Analysis, Integer Programs, Ranking Functions, Control-Flow Refinement

Funding funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2) and the DFG Research Training Group 2236 UnRAVeL

1 Introduction

There are many techniques and tools for automated complexity analysis of programs. Most of them infer variants of (mostly linear) polynomial ranking functions which are then combined to get a runtime bound for the overall program. However, linear ranking functions are incomplete for complexity analysis, even for loops with only linear arithmetic. For example, consider the loop from [2, 8], which terminates, but does not admit a linear ranking function:

$$\textbf{while } (x > 0) \textbf{ do } (x, y) \leftarrow (x + y, y - 1) \tag{1}$$

Its runtime is linear in the initial values of x and y, if they are positive initially. The reason is that if $y > 0$, then x grows first but it is decreased with the same "speed" once y has become negative. By *multiphase-linear ranking functions* (MΦRFs, see, e.g., [2, 3, 8]), one detects that the program has two phases: First y is decremented until it is negative. Afterwards, x is decremented until it is negative and the loop terminates. In [2], it is shown that the existence of an MΦRF for a loop implies linear runtime complexity. In this paper, we show how to integrate MΦRFs into our modular approach for complexity analysis of integer programs from [4]. In contrast to [2], we infer MΦRFs for parts of the program and combine the so-obtained bounds to an overall runtime bound. In this way, we obtain a powerful technique which can infer finite runtime bounds for programs containing loops like (1).

Different forms of *control-flow refinement* are used for program analysis (see, e.g., [5, 6, 10]) and such refinements have also been used to improve the automatic termination and complexity analysis of programs further. The basic idea is to gain "more information" on the values of variables to sort out certain paths in the program. We show how to integrate the technique for control-flow refinement from [5] into our modular analysis in a non-trivial way.

See [7] for the full version of our paper and [9] for a further improvement which integrates a special handling for sub-programs that correspond to *triangular weakly non-linear loops*.

2 Improving Runtime Bounds by Multiphase-Linear Ranking Functions

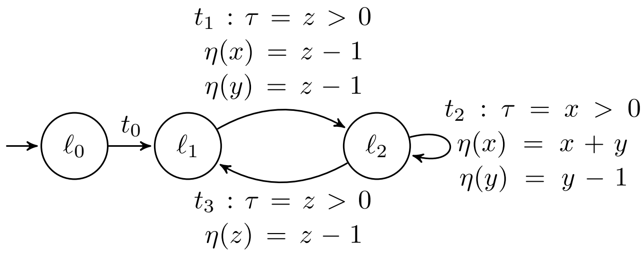


Figure 1 Integer Program with two Nested Loops

Instead of **while** loops as in (1), we use a formalism based on transitions. Fig. 1 depicts an *integer program* with locations $\mathcal{L} = \{\ell_0, \ell_1, \ell_2\}$ and variables $\mathcal{V} = \{x, y, z\}$. $\mathcal{T}$ is the set of transitions $t = (\ell, \tau, \eta, \ell')$, i.e., directed edges from a location ℓ to ℓ' which are labeled with a guard τ and an update function $\eta : \mathcal{V} \to \mathbb{Z}[\mathcal{V}]$. In Fig. 1 we omitted trivial guards, i.e., $\tau = \texttt{true}$, and trivial updates of the form $\eta(v) = v$. Note that transition t_2 corresponds to the loop in (1).

When evaluating the transition t, one moves from location ℓ to ℓ' if the guard τ is fulfilled, and the current state $\sigma : \mathcal{V} \to \mathbb{Z}$ is updated according to η. We denote such an evaluation step by $(\ell, \sigma) \to_t (\ell', \sigma')$, where $\sigma'(v) = \sigma(\eta(v))$ for all $v \in \mathcal{V}$. Moreover, $\to_{\mathcal{T}}$ is $\bigcup_{t \in \mathcal{T}} \to_t$.

To over-approximate the runtime of programs, we compute a *runtime bound* $\mathcal{RB}(t)$ for every transition $t \in \mathcal{T}$. Here, $\mathcal{RB}(t)$ is an arithmetic expression over the variables $\mathcal{V}$ such that $|\sigma|(\mathcal{RB}(t))$ over-approximates the number of applications of t in any evaluation starting with the initial state σ, where $|\sigma|(v) = |\sigma(v)|$ for all $v \in \mathcal{V}$.

Our approach is *modular*, i.e., program parts are analyzed as standalone programs and the results are then lifted to contribute to the overall analysis. To lift local to global runtime bounds, we also compute *size bounds* $\mathcal{SB}(t, v)$. The arithmetic expression $\mathcal{SB}(t, v)$ over-approximates the absolute value that the variable v may have *after* the transition t in any possible run. Since size bounds are needed to compute runtime bounds and vice versa, we compute and improve them in an alternating way.

To compute runtime bounds, we use *ranking functions*. Essentially, a ranking function must decrease by at least one in every evaluation step when a specific transition is applied. Moreover, the ranking function has to be non-negative before we apply a transition. Thus, if the function becomes negative, then the program terminates. An MΦRF extends this idea and uses a ranking function f_i for every "phase" $1 \leq i \leq d$ of a program. When the phases 1 to $i - 1$ are finished, the functions $f_1, \ldots, f_{i-1}$ remain negative and decreasing, but now f_i becomes decreasing as well. If all functions are negative, then the program terminates.

The following definition corresponds to so-called nested MΦRFs from [2, 8]. Here, the sum of f_{i-1} and f_i must be larger than the updated function f_i for all i. We set f_0 to 0. Then $f_0 + f_1 = f_1$ must be decreasing with each update. If f_1 becomes negative, then $f_1 + f_2 < f_2$ and thus, f_2 has to be decreasing with every update, and so on until f_d becomes decreasing. The program eventually terminates, since f_d must be non-negative whenever the program can be executed further. In contrast to [2, 8], we define MΦRFs for sub-programs $\mathcal{T}'_> \subseteq \mathcal{T}' \subseteq \mathcal{T}$ which is crucial for our modular approach (see Thm. 3). Let $\mathbb{Z}[\mathcal{V}]_{\text{lin}}$ denote the set of linear polynomials (i.e., of degree at most 1) over $\mathbb{Z}$ in the variables $\mathcal{V}$.

▶ **Definition 1** (MΦRFs for Sub-Programs). *Let $\varnothing \neq \mathcal{T}'_> \subseteq \mathcal{T}' \subseteq \mathcal{T}$ and $d \geq 1$. A tuple $f = (f_1, \ldots, f_d)$ of functions $f_1, \ldots, f_d : \mathcal{L} \to \mathbb{Z}[\mathcal{V}]_{\text{lin}}$ is an* MΦRF of depth d *for $\mathcal{T}'_>$ and $\mathcal{T}'$ if for all evaluation steps $(\ell, \sigma) \to_t (\ell', \sigma')$:*

(a) If $t \in \mathcal{T}'_>$, then $\sigma(f_{i-1}(\ell)) + \sigma(f_i(\ell)) \geq \sigma'(f_i(\ell')) + 1$ for all $1 \leq i \leq d$ and $\sigma(f_d(\ell)) \geq 0$.

(b) If $t \in \mathcal{T}' \setminus \mathcal{T}'_>$, then we have $\sigma\left(f_i(\ell)\right) \geq \sigma'\left(f_i(\ell')\right)$ for all $1 \leq i \leq d$.

For instance, (f_1, f_2) is an MΦRF for $\mathcal{T}'_> = \{t_2\}$ and $\mathcal{T}' = \{t_2, t_3\}$ in the program of Fig. 1, where $f_1(\ell_1) = f_1(\ell_2) = y + 1$ and $f_2(\ell_1) = f_2(\ell_2) = x$. So f_1, f_2 correspond to the two phases of the program, i.e., f_2 decreases once y has become negative.

We define the set of *entry transitions* of $\ell \in \mathcal{L}$ as $\mathcal{T}_\ell = \{t \mid t = (\ell', \tau, \eta, \ell) \wedge t \in \mathcal{T} \setminus \mathcal{T}'\}$ and the set of *entry locations* is $\mathcal{E}_{\mathcal{T}'} = \{\ell_{in} \mid \mathcal{T}_{\ell_{in}} \neq \varnothing \wedge \exists \ell' : (\ell_{in}, \tau, \eta, \ell') \in \mathcal{T}'\}$. Moreover, the *entry transitions of* $\mathcal{T}'$ are $\mathcal{ET}_{\mathcal{T}'} = \bigcup_{\ell \in \mathcal{E}_{\mathcal{T}'}} \mathcal{T}_\ell$. So for the program in Fig. 1 and $\mathcal{T}' = \{t_2, t_3\}$, we have $\mathcal{T}_{\ell_1} = \{t_0\}$, $\mathcal{T}_{\ell_2} = \{t_1\}$, $\mathcal{E}_{\mathcal{T}'} = \{\ell_2\}$, and $\mathcal{ET}_{\mathcal{T}'} = \{t_1\}$.

Now Lemma 2 gives rise to a runtime bound β_ℓ. In any run through $\mathcal{T}'$ starting in (ℓ, σ), all functions f_i of the MΦRF become negative after at most $|\sigma|(\beta_\ell)$ many $\mathcal{T}'_>$-evaluations. To use the bound β_ℓ in our modular approach, it must be weakly monotonically increasing. To transform polynomials into such bounds, we replace their coefficients by their absolute values (and denote this transformation by $\lceil \cdot \rceil$). So for example, $\lceil -x + 2 \rceil = |-1| \cdot x + |2| = x + 2$.

▶ **Lemma 2** (Local Runtime Bound for Sub-Program)**.** *Let $\varnothing \neq \mathcal{T}'_> \subseteq \mathcal{T}' \subseteq \mathcal{T}$ and let $f = (f_1, \ldots, f_d)$ be an MΦRF for $\mathcal{T}'_>$ and $\mathcal{T}'$. For all $1 \leq i \leq d$ we define the constants $\gamma_i \in \mathbb{Q}$, and for all $\ell \in \mathcal{E}_{\mathcal{T}'}$ we define the arithmetic expression β_ℓ:*

- *$\gamma_1 = 1$ and $\gamma_i = 2 + \frac{\gamma_{i-1}}{i-1} + \frac{1}{(i-1)!}$ for $i > 1$*
- *$\beta_\ell = 1 + d! \cdot \gamma_d \cdot \left(\lceil f_1(\ell) \rceil + \ldots + \lceil f_d(\ell) \rceil\right)$*

Then for any run $(\ell, \sigma) \left(\rightarrow^*_{\mathcal{T}' \setminus \mathcal{T}'_>} \circ \rightarrow_{\mathcal{T}'_>}\right)^n (\ell', \sigma')$ with $n \geq |\sigma|(\beta_\ell)$ and any $1 \leq i \leq d$, we have $\sigma'(f_i(\ell')) < 0$.

We have $\gamma_1 = 1$ and $\gamma_2 = 2 + \frac{1}{1} + \frac{1}{1} = 4$. So if $\mathcal{T}' = \{t_2, t_3\}$ in Fig. 1 is interpreted as a standalone program, then t_2 can be executed at most $|\sigma|(\beta_{\ell_2}) = |\sigma|(1 + 2! \cdot \gamma_2 \cdot (\lceil f_1(\ell_2) \rceil + \lceil f_2(\ell_2) \rceil)) = 8 \cdot |\sigma|(x) + 8 \cdot |\sigma|(y) + 9$ many times when starting in the configuration (ℓ_2, σ).

Note that β_ℓ in Lemma 2 is only a (linear) *local* bound w.r.t. the values of the variables at the start of the sub-program $\mathcal{T}'$. In an evaluation of the full program, we enter $\mathcal{T}'$ by an entry transition $t \in \mathcal{T}_\ell$ to an entry location $\ell \in \mathcal{E}_{\mathcal{T}'}$. Thus, to lift β_ℓ to a (possibly non-linear) global bound, we have to instantiate the variables in β_ℓ by (over-approximations of) the values that the variables have when reaching the sub-program $\mathcal{T}'$, i.e., after the transition t. To this end, we use *size bounds* $\mathcal{SB}(t, v)$ which over-approximate the largest absolute value of v after the transition t. We also use the notation $\mathcal{SB}(t, b)$ for arithmetic expressions b, where $\mathcal{SB}(t, b)$ results from b by replacing each variable v in b by $\mathcal{SB}(t, v)$. Hence, $\mathcal{SB}(t, \beta_\ell)$ is a (global) bound on the number of applications of transitions from $\mathcal{T}'_>$ if $\mathcal{T}'$ is entered once via the entry transition t. Here, weak monotonic increase of β_ℓ ensures that the over-approximation of the variables v in β_ℓ by $\mathcal{SB}(t, v)$ indeed leads to an over-approximation of $\mathcal{T}'_>$'s runtime.

However, for every entry transition t we also have to take into account how often the sub-program $\mathcal{T}'$ may be entered via t. We over-approximate this value by $\mathcal{RB}(t)$. This leads to Thm. 3. The analysis starts with a runtime bound $\mathcal{RB}$ and a size bound $\mathcal{SB}$ which map all transitions to ω, except for the transitions t outside of strongly connected components (SCCs), where $\mathcal{RB}(t) = 1$. Afterwards, $\mathcal{RB}$ and $\mathcal{SB}$ are refined repeatedly (see [4] for the computation of size bounds). Instead of using a single ranking function for the refinement of $\mathcal{RB}$ as in [4], Thm. 3 now allows us to replace $\mathcal{RB}$ by a refined bound $\mathcal{RB}'$ based on an MΦRF.

▶ **Theorem 3** (Refining Runtime Bounds Based on MΦRFs)**.** *Let $\mathcal{RB}$ be a runtime bound, $\mathcal{SB}$ a size bound, $\varnothing \neq \mathcal{T}'_> \subseteq \mathcal{T}' \subseteq \mathcal{T}$ such that $\mathcal{T}'$ does not contain any initial transitions, and β_ℓ be as in Lemma 2. Then $\mathcal{RB}'$ is also a runtime bound, where $\mathcal{RB}'(t) = \mathcal{RB}(t)$ for $t \notin \mathcal{T}'_>$ and*

$$\mathcal{RB}'(t_>) = \sum\nolimits_{\ell \in \mathcal{E}_{\mathcal{T}'}} \sum\nolimits_{t \in \mathcal{T}_\ell} \mathcal{RB}(t) \cdot \mathcal{SB}(t, \beta_\ell) \quad \textit{for all } t_> \in \mathcal{T}'_>.$$

▶ **Example 4.** In Fig. 1, t_1 and t_3 are evaluated at most z times (this can be shown by

while $(x < 0)$ **do**
  if $y < z$ **then**
    $y \leftarrow y - x$
  else
    $x \leftarrow x + 1$

Figure 2 Original Loop

while $(x < 0 \land y < z)$ **do**
  $y \leftarrow y - x$
while $x < 0 \land y \geq z$ **do**
  $x \leftarrow x + 1$

Figure 3 After Control-Flow Refinement

ranking functions for $\mathcal{T}' = \{t_1, t_2, t_3\}$ and $\mathcal{T}'_> = \{t_1\}$ resp. $\mathcal{T}'_> = \{t_3\}$). Hence, $\mathcal{RB}(t_0) = 1$ and $\mathcal{RB}(t_1) = \mathcal{RB}(t_3) = z$ is a runtime bound. So by Thm. 3 and $\mathcal{SB}(t_1, v) = z$ for all $v \in \mathcal{V}$, we get $\mathcal{RB}(t_2) = \mathcal{RB}(t_1) \cdot \mathcal{SB}(t_1, \beta_{\ell_2}) = z \cdot (8 \cdot \mathcal{SB}(t_1, x) + 8 \cdot \mathcal{SB}(t_1, y) + 9) = 16 \cdot z^2 + 9 \cdot z$. Thus, the runtime of the full program is bounded by $\sum_{i=0}^{3} \mathcal{RB}(t_i) = 16 \cdot z^2 + 11 \cdot z + 1$.

3 Improving Runtime Bounds by Control-Flow Refinement

Now we discuss another technique to improve the automated complexity analysis of integer programs, so-called *control-flow refinement*. The idea is to transform a program $\mathcal{P}$ into a new program $\mathcal{P}'$ which is "easier" to analyze. Of course, we ensure that the runtime of $\mathcal{P}'$ is *at least* the runtime of $\mathcal{P}$. Then it is sound to infer upper runtime bounds for $\mathcal{P}'$ instead of $\mathcal{P}$. Our approach is based on the *partial evaluation* technique of [5]. For example, this technique transforms the program in Fig. 2 into the equivalent one in Fig. 3. Clearly, Fig. 3 is easier to analyze as the two consecutive loops do not interfere with each other: x and z are constants in its first loop, while y and z are constants in its second loop. We integrated the technique for control-flow refinement from [5] into our modular analysis in a non-trivial way.

More precisely, we improved the "locality" of control-flow refinement and use partial evaluation as an SCC-based refinement technique. We refine a non-trivial SCC $\mathcal{T}_{SCC}$ of an integer program into multiple SCCs by considering "abstract" evaluations which do not operate on concrete states but on sets of states. These sets of states are characterized by constraints, i.e., a constraint φ stands for all states σ with $\sigma(\varphi) = \texttt{true}$. To formalize this, we label every location ℓ in the SCC by a constraint φ which describes (a superset of) those states σ which can occur in this location. So for any location ℓ, all reachable configurations have the form (ℓ, σ) such that $\sigma(\varphi) = \texttt{true}$.

We begin with labeling the entry locations of $\mathcal{T}_{SCC}$ by the constraint $\texttt{true}$. The constraints for the other locations in the SCC are obtained by considering how the updates of the transitions affect the constraints of their source locations and their guards. The resulting pairs of locations and constraints then become the new locations in the refined program.

Nevertheless, control-flow refinement may lead to an exponential blow-up of the program. Therefore, we heuristically minimize the strongly connected part of the program on which we apply partial evaluation, i.e., it is only applied *on-demand* on those transitions where our current runtime bounds are "not yet good enough". Such a sub-SCC-based partial evaluation results in considerably shorter runtimes than the SCC-based partial evaluation.

4 Evaluation of our Complexity Analysis Tool KoAT

In our evaluation we consider all 484 programs from the category for complexity analysis of C programs in the *Termination Problems Data Base (TPDB)* which is used in the annual *Termination and Complexity Competition* (where at most 366 of them *might* have finite runtime). In Fig. 4, we compare our implementation in the tool KoAT to the main other tools for complexity analysis of integer programs: CoFloCo [6], KoAT1 (the old version of KoAT from [4]), Loopus [11], and MaxCore [1]. For example, there are $24 + 228 = 252$ programs where KoAT + MΦRF5 + CFR shows that the program has at most linear runtime w.r.t. the

	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{>2})$	$\mathcal{O}(\text{EXP})$	$< \infty$	AVG$^+$(s)	AVG(s)
KoAT+MΦRF5+CFR	24	228	65	11	0	328	4.77	16.40
KoAT+MΦRF5+CFRSCC	24	228	65	11	0	328	5.72	16.53
KoAT + CFR	25	216	68	11	0	320	5.14	11.67
KoAT + CFRSCC	28	216	66	10	0	320	6.00	11.93
MaxCore	23	214	66	7	0	310	1.94	5.24
KoAT + MΦRF5	23	204	71	12	0	310	2.11	5.16
CoFloCo	22	195	66	5	0	288	0.81	2.95
KoAT1	25	168	74	12	6	285	2.36	2.97
KoAT	23	176	70	12	0	281	2.05	2.76
Loopus	17	169	49	4	0	239	0.84	0.72

Figure 4 Evaluation on the Category Complexity_C_Integer from the *TPDB*

initial values, and "$< \infty$" is the number of examples where a finite (possibly non-polynomial) bound on the runtime could be computed within the time limit of 5 minutes. "AVG$^+$(s)" is the average runtime of the tool on successful runs in seconds, whereas "AVG(s)" considers all runs including timeouts. Both MaxCore and KoAT + MΦRF5 (which applies MΦRFs of depth 5) solve 310 examples. In contrast, KoAT + CFRSCC (which uses control-flow refinement on the complete SCC) solves 320 examples, which makes KoAT the strongest tool on the benchmark set. KoAT + CFR uses our local sub-SCC approach which improves the runtime without reducing the number of solved examples. When enabling both control-flow refinement and multiphase-linear ranking functions then KoAT is even stronger, as KoAT + MΦRF5 + CFR solves 328 examples (i.e., 90 % of the potentially terminating ones). Moreover, it is faster than the equally powerful configuration KoAT + MΦRF5 + CFRSCC. A detailed evaluation, a web interface of KoAT, and its source code, binary, and a Docker image are available at `https://aprove-developers.github.io/ComplexityMprfCfr/`.

References

1 E. Albert, M. Bofill, C. Borralleras, E. Martín-Martín, and A. Rubio. Resource Analysis driven by (Conditional) Termination Proofs. *TPLP*, 19(5-6):722–739, 2019.

2 A. M. Ben-Amram and S. Genaim. On Multiphase-Linear Ranking Functions. In *Proc. CAV*, LNCS 10427, pages 601–620, 2017.

3 A. M. Ben-Amram, J. J. Doménech, and S. Genaim. Multiphase-Linear Ranking Functions and Their Relation to Recurrent Sets. In *Proc. SAS*, LNCS 11822, pages 459–480, 2019.

4 M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Analyzing Runtime and Size Complexity of Integer Programs. *ACM TOCL*, 38(4), 2016.

5 J. J. Doménech, J. P. Gallagher, and S. Genaim. Control-Flow Refinement by Partial Evaluation, and its Application to Termination and Cost Analysis. *TPLP*, 19:990–1005, 2019.

6 A. Flores-Montoya and R. Hähnle. Resource Analysis of Complex Programs with Cost Equations. In *Proc. APLAS*, LNCS 8858, pages 275–295, 2014.

7 J. Giesl, N. Lommen, M. Hark, and F. Meyer. Improving Automatic Complexity Analysis of Integer Programs. In *The Logic of Software: A Tasting Menu of Formal Methods*, LNCS 13360. 2022.

8 J. Leike and M. Heizmann. Ranking Templates for Linear Loops. *LMCS*, 11(1), 2015.

9 N. Lommen, F. Meyer, and J. Giesl. Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops. In *Proc. IJCAR*, LNCS 13385, 2022. To appear.

10 K. L. McMillan. Lazy Abstraction with Interpolants. In *Proc. CAV*, LNCS 4144, pages 126–136, 2006.

11 M. Sinn, F. Zuleger, and H. Veith. Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints. *J. Autom. Reason.*, 59(1):3–45, 2017.

Automatic Complexity Analysis of (Probabilistic) Integer Programs via KoAT

Nils Lommen
Fabian Meyer
Marcel Hark
Jürgen Giesl
LuFG Informatik 2, RWTH Aachen University, Aachen, Germany

Abstract

In former work [3], we developed an approach for automatic complexity analysis of integer programs, based on an alternating modular inference of upper runtime and size bounds for program parts. Recently, we extended and reimplemented this approach in a new version of our open-source tool KoAT (see [7, 10]). In order to compute runtime bounds, we analyze subprograms in topological order, i.e., in case of multiple consecutive loops, we start with the first loop and propagate knowledge about the resulting values of variables to subsequent loops. By inferring runtime bounds for one subprogram after the other, in the end we obtain a bound on the runtime complexity of the whole program. We first try to compute runtime bounds for subprograms by means of multiphase linear ranking functions (MΦRFs [1, 2, 7, 9]). If MΦRFs do not yield a finite runtime bound for the respective subprogram, then we apply a technique to handle so-called *triangular weakly non-linear loops* (twn-loops [5, 6, 8, 10]) on the unsolved parts of the subprogram. Moreover, we integrated control-flow refinement via partial evaluation [4] to improve the automatic complexity analysis of programs further. Additionally, in [11] we introduced the notion of expected size which allowed us to extend our approach to the computation of upper bounds on the expected runtimes of *probabilistic* programs.

2012 ACM Subject Classification Theory of computation → Complexity classes; Theory of computation → Program analysis; Software and its engineering → Automated static analysis

Keywords and phrases Automatic Complexity Analysis, (Probabilistic) Integer Programs, Ranking Functions, Decidable Subclasses, Control-Flow Refinement

Funding funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2) and the DFG Research Training Group 2236 UnRAVeL

References

1. Amir M. Ben-Amram and Samir Genaim. On Multiphase-Linear Ranking Functions. In *Proc. CAV*, LNCS 10427, pages 601–620, 2017. doi:10.1007/978-3-319-63390-9_32.
2. Amir M. Ben-Amram, Jesús J. Doménech, and Samir Genaim. Multiphase-Linear Ranking Functions and Their Relation to Recurrent Sets. In *Proc. SAS*, LNCS 11822, pages 459–480, 2019. doi:10.1007/978-3-030-32304-2_22.
3. Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Analyzing Runtime and Size Complexity of Integer Programs. *ACM Transactions on Programming Languages and Systems*, 38, 2016. doi:10.1145/2866575.
4. Jesús J. Doménech, John P. Gallagher, and Samir Genaim. Control-Flow Refinement by Partial Evaluation, and its Application to Termination and Cost Analysis. *Theory Pract. Log. Program.*, 19(5-6):990–1005, 2019. doi:10.1017/S1471068419000310.
5. Florian Frohn and Jürgen Giesl. Termination of triangular integer loops is decidable. In *Proc. CAV*, LNCS 11562, pages 426–444, 2019. doi:10.1007/978-3-030-25543-5_24.
6. Florian Frohn, Marcel Hark, and Jürgen Giesl. Termination of Polynomial Loops. In *Proc. SAS*, LNCS 12389, pages 89–112, 2020. Full version available at https://arxiv.org/abs/1910.11588. doi:10.1007/978-3-030-65474-0_5.

7 Jürgen Giesl, Nils Lommen, Marcel Hark, and Fabian Meyer. Improving Automatic Complexity Analysis of Integer Programs. In *The Logic of Software: A Tasting Menu of Formal Methods*, LNCS 13360. 2022. To appear. Also in *CoRR*, abs/2202.01769. URL: `https://arxiv.org/abs/2202.01769`.

8 Marcel Hark, Florian Frohn, and Jürgen Giesl. Polynomial Loops: Beyond Termination. In *Proc. LPAR*, EPiC 73, pages 279–297, 2020. `doi:10.29007/nxv1`.

9 Matthias Heizmann and Jan Leike. Ranking Templates for Linear Loops. *Logical Methods in Computer Science*, 11(1), 2015. `doi:10.2168/LMCS-11(1:16)2015`.

10 Nils Lommen, Fabian Meyer, and Jürgen Giesl. Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops. In *Proc. IJCAR*, LNCS, 2022. To appear.

11 Fabian Meyer, Marcel Hark, and Jürgen Giesl. Inferring Expected Runtimes of Probabilistic Integer Programs Using Expected Sizes. In *Proc. TACAS*, LNCS 12651, pages 250–269, 2021. `doi:10.1007/978-3-030-72016-2_14`.

CeTA – A certifier for termCOMP 2022

Christina Kohl
University of Innsbruck, Austria

René Thiemann
University of Innsbruck, Austria

Abstract

CeTA is a certifier that can be used to validate automatically generated termination and complexity proofs during the termination competition 2022. We briefly highlight some features of CeTA that have been recently added.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Automated reasoning

Keywords and phrases Certification, Isabelle/HOL, Termination, Complexity, Confluence

CeTA is a certifier for automatically generated proofs. Its soundness – if CeTA accepts a proof of a certain property, then the property holds – is proven in the Isabelle/HOL [2] formalization IsaFoR [3]. A complete list of supported proof techniques as well as IsaFoR and CeTA itself are available at `http://cl-informatik.uibk.ac.at/software/ceta/`. We highlight some recent extensions of CeTA for validating termination proofs.

Improved Support for Confluence of TRS Certain termination technique are only valid for confluent TRSs, e.g., the switch between termination and innermost termination in (non)termination proofs [1]. Here, the improvement consists of adding development closedness [4] as new confluence criterion to IsaFoR and CeTA.

Improved Support of Weighted Path Order The weighted path order (WPO) [5] unifies several existing reduction orders. However, the original definition of WPO does not generalize the recursive path order (RPO), since it does not contain the status of RPO that can select between lexicographic or multiset comparison of arguments. We generalized WPO by adding such a status and further proved that RPO is an instance of this generalized WPO within IsaFoR. So far, the certification problem format (CPF) does not specify how such a generalized WPO should be specified. We are looking forward to collaborate with tool authors that would like to exploit this more general WPO, i.e., we can negotiate the design of the generalized WPO within CPF and will extend CeTA accordingly.

Improved Efficiency of Parsing In Isabelle 2018 the modeling of characters was completely changed, where from that point onwards only ASCII symbols (characters 0–127) have been allowed, since target languages differ in their treatment of characters outside the ASCII range. However, CeTA was also getting slower because of that change since characters have been encoded as tuples of Booleans, an effect which in particular materializes during parsing CPFs. Since often the processing time after parsing is much higher than the parsing time itself, this effect got unnoticed, until Johannes Waldmann recently gave us a detailed problem report with CPFs of a different style, namely large proofs using rather easy-to-check (non)termination-techniques. As a consequence, IsaFoR and CeTA now include a more efficient implementation of characters (that avoids the tuple representation, and is logically equivalent), so that parsing is now roughly 4 times faster than before.

References

1 Bernhard Gramlich. Abstract relations between restricted termination and confluence properties of rewrite systems. *Fundam. Informaticae*, 24(1/2):2–23, 1995. doi:10.3233/FI-1995-24121.

2 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-45949-9.

3 René Thiemann and Christian Sternagel. Certification of termination proofs using CeTA. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468. Springer, 2009. doi:10.1007/978-3-642-03359-9_31.

4 Vincent van Oostrom. Developing developments. *Theor. Comput. Sci.*, 175(1):159–181, 1997. doi:10.1016/S0304-3975(96)00173-9.

5 Akihisa Yamada, Keiichirou Kusakari, and Toshiki Sakabe. A unified ordering for termination proving. *Sci. Comput. Program.*, 111:110–134, 2015. doi:10.1016/j.scico.2014.07.009.

Certified Matchbox

Johannes Waldmann
HTWK Leipzig

Abstract

We describe the Matchbox termination prover that takes part in the category of certified termination of string rewriting in the Termination Competition 2022.

1 Introduction

Matchbox originally (2003) [7] computed RFC-matchbound certificates for termination of string rewriting, via completion of automata. It was extended (2006) to compute matrix interpretations [4], via constraint solving. A recent addition (2019) is sparse tiling [2]. For participation in Termination Competition 2022, Matchbox will produce termination certificates to be checked with CeTA [6]. Therefore, the range of methods is restricted (no RFC matchbounds, no sparse tiling). Still, with an efficient implementation of available methods, performance comes near non-certified Matchbox of last year.

2 Interpretations

Matchbox uses matrix interpretations over natural and over arctic numbers. Conditions for interpretations are formulated with the ersatz library (Kmett 2010, `https://hackage.haskell.org/package/ersatz`). In particular, we use a representation of unary and binary numbers of fixed bit width, with an extra overflow bit. Constraints are solved with Kissat (Biere 2020, `http://fmv.jku.at/kissat/`). Kissat is accessed via a Haskell API (`https://github.com/jwaldmann/ersatz-kissatapi`) that, in turn, uses Kissat's C API that conforms to the IPASIR standard (Balyo 2017, `https://github.com/biotomas/ipasir`).

Matchbox looks for quasi-periodic interpretations (QPI) [9] as well, and presents them to CeTA as arctic matrix interpretations [5]. The constraint system for a QPI is smaller, and can often be solved faster, than the corresponding arctic matrix constraint. QPIs seem to be helpful for `Wenzel_16` and `Waldmann_07` systems.

Matchbox looks for weights, described by a system of linear inequalities, solved by GLPK (Makhorin 2000, `https://www.gnu.org/software/glpk/`), accessed via hmatrix-glpk (Ruiz and Steinitz 2018, `https://hackage.haskell.org/package/hmatrix-glpk`).

3 Non-Sparse Tiling

Since sparse tiling is not certified currently, Matchbox applies "full tiling": an R over Σ is transformed to an equivalent R' over the set of k-tiles Σ^k. This corresponds to semantic labeling in the k-shift algebra, for the $(k-1)$-fold left-context-closed system.

The resulting labeled system may be large, so Matchbox will only use weights (not matrices) to remove labeled rules. To keep the search space (and the certificates) small, we unlabel immediately. This method solves many `ICFP_2010` problems.

4 Loops and Transport Systems

Matchbox finds loops by enumerating forward closures. Closures are kept in a priority queue. The priority of a closure (l, r) with k rewrite steps is $\log_2 \log_2 k - 4 \log_2 \log_2 |l| - 0.5 \log_2 |r|$. That function was determined experimentally.

Matchbox searches for Transport Systems (TS) [8], and presents them to CeTA as loops. Enumeration of TS candidates uses tiering operations of Leancheck (Matela 2017, `https://hackage.haskell.org/package/leancheck`).

5 Strategy

Matchbox will first remove rules via tiling. Then it applies the dependency pairs transformation [1], for both the original and the mirrored system in parallel, with recursive decomposition of the estimated dependency graph [3]. It will then remove dependency pairs via interpretations, considering usable rules only.

Matchbox has a language for specifying the search strategy. It allows to control elementary steps (e.g., search for QPI), to restrict searches (e.g., only if number of rules is below some bound), and to combine searches sequentially and concurrently.

In preparation for competition, we used an evolutionary algorithm that modifies parameters in strategy expressions

6 Performance

In a pre-competition test run (starexec job 52669, `https://termcomp.imn.htwk-leipzig.de/flexible-table/Query%20%5B%5D/47876/52669`), certified-matchbox 2022 obtains 1250 YES, 184 NO. That is 95 percent of uncertified-matchbox 2021.

Matchbox does write large proofs sometimes: full k-tiling will multiply system size by $|\Sigma|^{2(k-1)}$. Conversion of a transport system to a loop gives an exponential blow-up. CeTA's proof format is verbose: both from built-in redundancies, and from XML representation.

The (2-tiling) certificate for `ICFP_2010/26132` has size 325 MB. The certificate that represents a loop of length 1024 for `Wenzel_16/abaaaaa-aaaaaaabababab` has size 173 MB. The total certificate size over a test run was 24 GB. These certificates are highly compressible—down to 0.3 percent on average.

References

1. Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000.
2. Alfons Geser, Dieter Hofbauer, and Johannes Waldmann. Sparse tiling through overlap closures for termination of string rewriting. In Herman Geuvers, editor, *FSCD*, LIPIcs 131, 2019.
3. Nao Hirokawa and Aart Middeldorp. Dependency pairs revisited. In Vincent van Oostrom, editor, *RTA*, LNCS 3091, pages 249–268. Springer, 2004.
4. Dieter Hofbauer and Johannes Waldmann. Termination of string rewriting with matrix interpretations. In Frank Pfenning, editor, *RTA*, LNCS 4098, 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. René Thiemann and Christian Sternagel. Certification of termination proofs using ceta. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, LNCS 5674, pages 452–468. Springer, 2009.
7. Johannes Waldmann. Matchbox: A tool for match-bounded string rewriting. In Vincent van Oostrom, editor, *RTA*, LNCS 3091, pages 85–94. Springer, 2004.
8. Johannes Waldmann. Non-termination. Austro-Japanese Rewriting Workshop, `https://www.imn.htwk-leipzig.de/~waldmann/talk/07/ajrw/`, 2007.
9. Hans Zantema and Johannes Waldmann. Termination by quasi-periodic interpretations. In Franz Baader, editor, *RTA*, LNCS 4533, pages 404–418. Springer, 2007.