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 Dartmouth College
Cynthia Kop (chair) Radboud University Nijmegen
Akihisa Yamada    AIST, Japan
Hans Zantema       Eindhoven, University of Technology
## Contents

Preface ................................................................. i
Organization ......................................................... iii

### Invited Talks

Efficient Formalization of Simplification Orders
   René Thiemann and Akihisa Yamada ........................ 1

### Regular Papers

Tuple Interpretations and Applications to Higher-Order Runtime Complexity
   Cynthia Kop and Deivid Vale ................................. 6

A transitive HORPO for curried systems
   Liye Guo and Cynthia Kop ................................ 11

Approximating Relative Match-Bounds
   Alfons Geser, Dieter Hofbauer and Johannes Waldmann ........ 16

Hydra Battles and AC Termination
   Nao Hirokawa and Aart Middeldorp ......................... 21

A Calculus for Modular Non-Termination Proofs by Loop Acceleration
   Florian Frohn and Carsten Fuhs .......................... 26

Deciding Termination of Uniform Loops with Polynomial Parameterized Complexity
   Marcel Hark, Florian Frohn and Jürgen Giesl ................ 31

Improved Automatic Complexity Analysis of Integer Programs
   Jürgen Giesl, Nils Lommen, Marcel Hark and Fabian Meyer ......... 36

### System Descriptions

Automatic Complexity Analysis of (Probabilistic) Integer Programs via KoAT
   Nils Lommen, Fabian Meyer, Marcel Hark and Jürgen Giesl .......... 41

CeTA – A certifier for termCOMP 2022
   Christina Kohl and René Thiemann ........................ 43

Certified Matchbox
   Johannes Waldmann ........................................ 45
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 Isabelle Formalization of Rewriting [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 \( \triangleright \) is defined by \( C[t] \triangleright t \).

We define a strict and a non-strict relation on terms (\( \triangleright_{\text{RoT}} \) and \( \triangleright_{\text{RoT}} \)) as follows:

\[
\begin{align*}
  s &\triangleright_{\text{RoT}} t, \text{ iff } \quad 1. \ s \triangleright t, \text{ or } \\
  s &\triangleright_{\text{RoT}} t, \text{ and } \quad 2. \ s \triangleright_{\text{RoT}} t.
\end{align*}
\]

2.1 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 \( \triangleright \) 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 \( \tau : \mathcal{F} \rightarrow \{\text{lex, mul}\} \) be a status.

The bounded lexicographic extension is parametrized by some \( b \in \mathbb{N} \), the bound, and it is defined as

\[
[s_1, \ldots, s_n] \triangleright_{\text{lex,b}} [t_1, \ldots, t_m] \iff [s_1, \ldots, s_n] \triangleright_{\text{lex}} [t_1, \ldots, t_m] \land (n = m \lor m \leq b).
\]

There are similar definitions for \( \triangleright_{\text{lex}} \) and \( \triangleright_{\text{lex,b}} \). We sometimes write \( \triangleright_{\text{lex}} \) and \( \triangleright_{\text{lex,b}} \) 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 \( \triangleright \) 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 \( \tau : \mathcal{F} \rightarrow \{\text{lex, mul}\} \) be a status.

The bounded lexicographic extension is parametrized by some \( b \in \mathbb{N} \), the bound, and it is defined as

\[
[s_1, \ldots, s_n] \triangleright_{\text{lex,b}} [t_1, \ldots, t_m] \iff [s_1, \ldots, s_n] \triangleright_{\text{lex}} [t_1, \ldots, t_m] \land (n = m \lor m \leq b).
\]

There are similar definitions for \( \triangleright_{\text{lex}} \) and \( \triangleright_{\text{lex,b}} \). We sometimes write \( \triangleright_{\text{lex}} \) and \( \triangleright_{\text{lex,b}} \) also for the bounded lexicographic extension if the bound is clear from the context.
a. \( s = f(s_1, \ldots, s_n) \) and \( \exists i \in \{1, \ldots, n\}. \ s_i \succeq_{\text{RoT}} t \), or

b. \( s = f(s_1, \ldots, s_n), \ t = g(t_1, \ldots, t_m) \) and
   i. \( \forall j \in \{1, \ldots, m\}. \ s \succ_{\text{RoT}} t_j \) and
   ii. A. \( f \succ g \) or
       B. \( f \preceq g \) and \( \tau(f) = \text{lex} \) for all \( f \in \mathcal{F} \).

The relation \( s \succeq_{\text{RoT}} t \) is defined in the same way, where \( \succeq_{\tau(f)} \) in case 2.b.ii.B is replaced by \( \succeq_{\tau(f)} \), 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;
  - \( \tau(f) = \text{lex} \) for all \( f \in \mathcal{F} \).

- Classical RPO is like LPO without the requirement on \( \tau \).

- Classical KBO is similar to the setup of LPO, but
  - instead of the trivial reduction pair one defines \((\succ, \succeq)\) with the help of weight functions and the multisets of variables of terms;
  - the structure of KBO and of \((\succ_{\text{RoT}}, \succeq_{\text{RoT}})\) 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 \( s \not\in \mathcal{V}, \ x \in \mathcal{V} \) and \( s \succeq x \), then both \( s \succ_{\text{KBO}} x \) and \( s \succeq_{\text{KBO}} x \).

- Remark. The relation \( \succ_{\text{RoT}} \) defined above looks like a simplified form of WPO, e.g., the status function \( \pi \) of WPO (for selecting arguments of each individual function symbol) has been omitted. However, the original WPO does not completely subsume \( \succ_{\text{RoT}} \), since the status function \( \tau \) of \( \succ_{\text{RoT}} \) 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 \( f_0(1) \succ_{\text{RoT}} f_1(0, 1) \succ_{\text{RoT}} f_2(0, 0, 1) \succ_{\text{RoT}} \ldots \) by using a quasi-precedence where \( f_i \geq_{\text{KBO}} f_j \) for all \( i, j \) and \( f_i > 1 > 0 \) for all \( i \). The second refinement are comparisons of the form \( x \succeq c \) in case 2.d. For LPO one requires that \( c \) is least in precedence among all symbols, in the same way as in \( \succ_{\text{RoT}} \). By contrast, for KBO \( f \geq_{\text{KBO}} c \) is only required for those \( f \) which are constants and have weight \( w_0 \).

Note that activating both requirements – quasi-precedences and \( x \succeq c \) comparisons – is sound for LPO, requires a special definition of lexicographic extensions for KBO, and is unsound for RPO.
4 Efficient Formalization of Simplification Orders

▶ Example 1. Consider RPO with both refinements, i.e., $(\succ, \succcurlyeq)$ is the trivial reduction pair. Let $\succeq = \mathcal{F} \times \mathcal{F}$ be the trivial precedence where all symbols are equivalent. Let $\tau(c) = \text{lex}$ and $\tau(d) = \text{mul}$ for two constants $c, d \in \mathcal{F}$. Then using case 2.d we have $x \succcurlyeq_{\text{RPO}} c$, but $d \succcurlyeq_{\text{RPO}} c$ 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 $f(x) \succ_{\text{KBO}} x$; however, for $f(g_i(t_1, \ldots, t_i)) \succ_{\text{KBO}} g_i(t_1, \ldots, t_i)$ (closure under substitutions), only case 2.b.ii.B is applicable, i.e., one needs lexicographic comparisons $[g_i(t_1, \ldots, t_i)] \succ_{\text{KBO}} [t_1, \ldots, t_i]$ 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. $f \succ g$ and $\tau(f) \neq \tau(g)$ and $m = 0$ (and $n > 0$ for $s \succ_{\text{RoT}} t$)

That $(\succ_{\text{RoT}}, \succcurlyeq_{\text{RoT}})$ really forms a reduction pair with this fix has been formally proven. Actually, we have formalized an extended version of $(\succ_{\text{RoT}}, \succcurlyeq_{\text{RoT}})$ 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 $\tau: \mathcal{F} \to \{\text{lex, mul}\}$, and add case 2.b.ii.C for symbols with different status.

▶ Theorem 3. $(\succ_{\text{RoT}}, \succcurlyeq_{\text{RoT}})$ is a reduction pair and $\succ_{\text{RoT}}$ 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 $\tau$). This covers quasi-precedences and the $x \succcurlyeq c$ 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 $\succeq c$ for all constants $f$ of weight $w_f$; whereas in WPO $f \succeq c$ is required for all $f \in \mathcal{F}$. We solve this problem by changing the quasi-precedence $\succeq$ of KBO into some quasi-precedence $\succeq'$ in a way that
   - KBO-minimal constants w.r.t. $\succeq$ are WPO-minimal w.r.t. $\succeq'$, and
   - $\succcurlyeq_{\text{KBO}}$ and $\succ_{\text{KBO}}$ are unmodified when switching from $\succeq$ to $\succeq'$.

2. For the syntactic differences, we prove that they do not affect the defined relations.
   - The additional case $f(C[x]) \succ_{\text{KBO}} x$ of KBO can be simulated since $\succ_{\text{RoT}}$ is a simplification order.
   - Assume that $f(s_1, \ldots, s_n) \succ_{\text{KBO}} f(t_1, \ldots, t_m)$ was shown by 2.b. Here we use some properties of KBO to conclude $f(s_1, \ldots, s_n) \succ_{\text{KBO}} t_j$ 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 \( s \uparrow \KBO t \rightarrow f(s_1, \ldots, s_n) \uparrow \KBO t \) 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 \( \succ \RoT \) 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.

\begin{itemize}
\item[] Theorem 4. Let a KBO with quasi-precedence \( \geq \) and some bound \( b \) be given. Then a reduction pair (encoding the weight-function) and quasi-precedence \( \geq' \) can be constructed as parameters to \( \succ \RoT \) and \( \preceq \RoT \) (or to WPO), such that \( (s \uparrow \KBO t) \leftrightarrow (s \uparrow \RoT t) \) and \( (s \preceq \KBO t) \leftrightarrow (s \preceq \RoT t) \) for all terms \( s, t \) whose function symbols have arity below \( b \).
\item[] Corollary 5. For every KBO over a finite signature there exists an equivalent WPO.
\item[] Corollary 6. KBO is transitive, closed under substitutions and well-founded.
\end{itemize}

**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 \( s \uparrow \RoT t \uparrow \RoT u \leftrightarrow s \uparrow \RoT u \) and \( s \uparrow \RoT t \rightarrow s\sigma \uparrow \RoT t\sigma \). 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 \( t_1 \uparrow \KBO t_2 \uparrow \KBO \ldots \). 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 \( t_1 \uparrow \RoT t_2 \uparrow \RoT \ldots \) 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

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.

Let \( B \) be a set of base types (or sorts). The set \( T_B \) of simple types is built using the right-associative \( \Rightarrow \) as follows. Every \( \iota \in B \) is a type of order 0. If \( \sigma, \tau \) 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 \( F \) of function symbols together with a function \( \text{typeof} : F \rightarrow T_B \). Additionally, we assume, for each \( \sigma \in T_B \), a countable infinite set of type-annotated variables \( X_\sigma \) disjoint from \( F \). We will denote \( f, g, \ldots \) 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 $\sigma$ using the following clauses:

(i) $x :: \sigma$, if $x \in X; (ii) f :: \sigma$, if $\text{typeOf}(f) = \sigma$; and (iii) $(s, t) :: \tau$, if $s :: \tau \Rightarrow t$ and $t :: \tau$.

Application is left-associative. We denote $\text{vars}(s)$ for the set of variables occurring in $s$ and say $s$ is ground if $\text{vars}(s) = \emptyset$. A rewriting rule $\ell \Rightarrow r$ is a pair of terms of the same type such that $\ell = f_1 \ldots f_n$ and $\text{vars}(\ell) \supseteq \text{vars}(r)$. 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 $s \Rightarrow t$. The innermost rewrite relation induced by $\mathcal{R}$ is defined as follows:

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

In what follows we only allow for innermost reductions. So, we drop the $i$ from the arrow, and $s \Rightarrow t$ is to be read as $s \Rightarrow^* t$. 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 $0 :: \text{nat}, s :: \text{nat} \Rightarrow \text{nat}, \text{nil} :: \text{list}, \text{cons} :: \text{nat} \Rightarrow \text{list} \Rightarrow \text{list},$ and $F,G \in \text{nat} \Rightarrow \text{nat}$; types of other function symbols and variables can be easily deduced.

<table>
<thead>
<tr>
<th>map $F \text{nil}$</th>
<th>$\rightarrow \text{nil}$</th>
<th>comp $F G x$</th>
<th>$\rightarrow F(G x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>map $F \text{cons}(x x x)$</td>
<td>$\rightarrow \text{cons}(F x)(\text{map} F x)$</td>
<td>app $F x$</td>
<td>$\rightarrow F x$</td>
</tr>
<tr>
<td>$d 0$</td>
<td>$\rightarrow 0$</td>
<td>$\text{add} x 0$</td>
<td>$\rightarrow x$</td>
</tr>
<tr>
<td>$d(s x)$</td>
<td>$\rightarrow s(s(d x))$</td>
<td>$\text{add} x (s y)$</td>
<td>$\rightarrow s(\text{add} x y)$</td>
</tr>
</tbody>
</table>

**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 $f : A \rightarrow B$ 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 $A \rightarrow B$ as usual. If $A, B$ are quasi-ordered, the notation $A \Rightarrow B$ 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 unit = $(\{u\}, \sqsupseteq)$, where $u \sqsupseteq u$.

3 **Higher-Order Tuple Interpretations for Innermost Rewriting**

To define interpretations, we will start by providing an interpretation of types (Def. 2). Types $\sigma$ are interpreted by tuples $\langle t, \rho \rangle$ 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 $\sigma$, we define the cost-size tuple interpretation of $\sigma$ as $\langle t, \rho \rangle = C_\sigma \times S_\sigma$ where $C_\sigma$ (respectively $S_\sigma$) is defined as follows:

- $C_\sigma = \mathbb{N} \times F^C_\sigma$
- $S_\sigma = \{N^K[i], \sqsupseteq\}, K[i] \geq 1$
- $F^C_\sigma = \text{unit}$
- $S_{\sigma \Rightarrow \tau} = S_{\sigma} \Rightarrow S_{\tau}$
- $F^C_{\sigma \Rightarrow \tau} = (F^C_\sigma \times S_\sigma) \Rightarrow C_{\tau}$,

where $F^C_{\sigma \Rightarrow \tau} (S_{\sigma \Rightarrow \tau})$ is the set of weakly monotonic functions from $F^C_\sigma \times S_\sigma$ to $C_\tau$ ($S_\sigma$ to $S_{\tau}$). The quasi-ordering on those sets is the induced point-wise comparison. The set $\langle t, \rho \rangle$ is ordered as follows: $(n, f), s) \succ ((m, g), t)$ if $n > m$, $f \geq g$ and $s \sqsupseteq t$; and $((n, f), s) \succ ((m, g), t)$ if $n \geq m$, $f \geq g$ and $s \sqsupseteq t$. 

WST 2022 7
The cost tuple $C_\sigma = N \times F^\sigma_\sigma$ of $\langle \sigma \rangle$ holds the cost information of reducing a term of type $\sigma$ 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 $F^\sigma_\sigma$; the cost tuple of a base type is then $C_\sigma = N \times \text{unit}$. Functional types do possess an intrinsically functional component (the cost of applying the function), which in our setting is expressed by $F^\sigma_\sigma \Rightarrow \tau = F^\sigma_\sigma \times S_\sigma \Rightarrow C_\tau$. For functional types the numeric component represents the cost of partial application.

To determine the number $K[\epsilon]$, associated to each sort $\epsilon$, 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 $(s^n 0) :: \text{nat}$ as their size. This gives us $K[\text{nat}] = 1$. Another example is the sort list (of natural numbers): it is natural to regard measures like length and maximum element size. This results in $K[\text{list}] = 2$. 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 $\sigma$, the set $C_\sigma$ is well-founded and $S_\sigma$ quasi-ordered. Their product, that is, $\langle (\sigma), \succ, \succeq \rangle$, is well-founded.

**Semantic Application** To interpret each term $s :: \sigma$ to an element of $\langle \sigma \rangle$ (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 $f \in \langle \sigma \Rightarrow \tau \rangle$, and $x \in \langle \sigma \rangle$, our goal is to define the application $f \cdot x$ of $f$ to $x$. Let us illustrate the idea with a concrete example: consider the type $\sigma = (\text{nat} \Rightarrow \text{nat}) \Rightarrow \text{list} \Rightarrow \text{list}$, which is the type of map defined in Example 1. The function map takes as argument a function $F$ of type $\text{nat} \Rightarrow \text{nat}$ and a list $q$, and applies $F$ to each element of $q$. The cost interpretation of map is a functional in $C_\sigma$ parametrized by functional arguments carrying the cost and size information of $F$ and a cost-size tuple for $q$.

\[
\text{the functional cost of map } = \begin{aligned}
\text{cost of } F & \quad \text{cost of } F \\
\text{size of } F & \quad \text{size of } q
\end{aligned}
\]

Hence, we write an element of such space as the tuple $(n, f^c)$. Size sets are somewhat simpler with $(N \Rightarrow N) \Rightarrow N \Rightarrow \text{nat}$, therefore, a functional cost-size tuple $f$ is represented by $f = ((n, f^c), f^s)$. 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 $f$ to a cost-size tuple $x$ of the form $\langle (m, g^c), g^s \rangle$ 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 $f^c(g^c, g^s) = (k, h)$ belongs to $C_{\text{nat}}$, and add the numeric components together obtaining: $f \cdot x = \langle (n + m + k), f^c(g^c, g^s)), f^s(g^s) \rangle$. Notice that this gives us a new cost-size tuple with cost component in $N \times (C_{\text{nat}} \Rightarrow C_{\text{nat}})$ and size component in $S_{\text{list}} \Rightarrow S_{\text{list}}$.

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

\[
\text{let } f^c(g^c, g^s) = (k, h); \text{ then } ((n, f^c), f^s) \cdot ((m, g^c), g^s) = ((n + m + k, h), f^s(g^s))
\]

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

**Lemma 5.** If $f$ is in $\langle \sigma \Rightarrow \tau \rangle$ and $x$ is in $\langle \sigma \rangle$, then $f \cdot x$ belongs to $\langle \tau \rangle$.  

WST 2022
Remark 6. In order to ease notation, we project sets $\pi_1 : A \times \text{unit} \to A$ and $\pi_2 : \text{unit} \times A \to A$ and compose functions with projections, so a function in $\text{unit} \times A \to B \times \text{unit}$ is lifted to a function in $A \to B$. The functional cost of map is then read as follows:

$$\text{cost of } F \quad \text{size of } F \quad \text{size of } q$$

The $n$ 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 $\text{add} 0 \to \text{id}$, then the cost tuple of $\text{add} 0$ is $(1, \lambda x.0)$. However, since in most cases the first component is 0, we will typically omit these zeroes and simply write for instance $\lambda F.q.f^c(F,q)$ instead of $(0, \lambda F.(0, \lambda q.f^c(F,q)))$.

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 $(B,F,\text{typeOf})$ consists of: (i) a family of cost/size tuples $\{(c\|\sigma)\}_{\sigma \in T_B}$ and (ii) an interpretation function $J$ which maps each $f \in F$ of type $\sigma$ to a cost-size tuple in $\{c\}$. 

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

$$J_0 = \langle 0,0 \rangle \quad J_1 = \langle \lambda x.0, \lambda x.x + 1 \rangle 
J_{\text{nil}} = \langle 0, \langle 0,0 \rangle \rangle \quad J_{\text{cons}} = \langle \lambda xq.0, \lambda xq.(q + 1, \text{max}(x,q_m)) \rangle$$

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:

$$\begin{align*}
J_{\text{app}} &= \langle \lambda F.x.F^c(x) + 1, \lambda F.x.F^c(x) \rangle \\
J_{\text{id}} &= \langle \lambda x.x + 1, \lambda x.2x \rangle \\
J_{\text{add}} &= \langle \lambda xy.y + 1, \lambda xy.x + y \rangle \\
J_{\text{comp}} &= \langle \lambda FGz.F^c(G^c(z)) + 1, \lambda FGz.F^c(G^c(x)) \rangle \\
J_{\text{map}} &= \langle \lambda Fq.q.F^c(q_m) + 1, \lambda Fq.\langle q, F^c(q_m) \rangle \rangle 
\end{align*}$$

A valuation $\alpha$ is a function that maps each $x : \sigma$ to a cost-size tuple in $\{c\}$. Due to innermost strategy, we can assume the interpretation of every variable $x : \iota$ has zero cost. This is formalized by assigning $\alpha(x) = \langle (0,0), x^\iota \rangle$, for all $x \in X$ 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 $\alpha(F) = \langle (0, f^\iota), f^\iota \rangle$ when $F : \sigma \Rightarrow \tau$.

Definition 9. We extend $J$ to an interpretation $\llbracket \cdot \rrbracket_{\alpha,\mathcal{J}}$ of terms as follows:

$$\llbracket x \rrbracket_{\alpha,\mathcal{J}} = \alpha(x) \quad \llbracket f \rrbracket_{\alpha,\mathcal{J}} = \langle n, J_f^c \rangle, n \in \mathbb{N} \quad \llbracket st \rrbracket_{\alpha,\mathcal{J}} = \llbracket s \rrbracket_{\alpha,\mathcal{J}} \cdot \llbracket t \rrbracket_{\alpha,\mathcal{J}}$$

We are interested in interpretations satisfying a compatibility requirement:

Theorem 10 (Innermost Compatibility Theorem). Let $\alpha$ be a valuation. If $\llbracket f \rrbracket_{\alpha,\mathcal{J}} \succeq \llbracket r \rrbracket_{\alpha,\mathcal{J}}$ for all rules $\ell \Rightarrow r \in \mathcal{R}$, then $\llbracket s \rrbracket_{\alpha,\mathcal{J}} \succeq \llbracket t \rrbracket_{\alpha,\mathcal{J}}$, whenever $s \rightarrow^i R t$.

One can check that the TRS from Example 1 interpreted as in Example 8 satisfies 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 $f \in F$ is a defined symbol if it occurs at the head of a rule, i.e., there is a rule $f \ell_1 \ldots \ell_k \rightarrow r \in R$. 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 $c d_1 \ldots d_k$ 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 $f d_1 \ldots d_m$ with $f$ a defined symbol and all $d_1, \ldots, d_m$ data terms. The set $T_B(F)$ 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 $\iota_i$ are the input data types and $\kappa$ the output type of the program.

Definition 13. The innermost derivation height of $s$ is $dh_R(s) = \{n \mid \exists t : s \rightarrow^n t\}$. The innermost runtime complexity function with respect to a TRS $R$ is $irc_R(n) = \max\{dh_R(s) \mid s \in T_B(F) \land |s| \leq n\}$.

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

We can build programs by adding a new 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: $\text{main } x q \rightarrow \text{map } (\text{add } x) q$. Hence, computing this program on inputs $n$ and list $q$ is equivalent to reducing the term $\text{main } n q$ to normal form. Its runtime complexity is therefore bounded by the cost-tuple of $[\text{main } n q]$.

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

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 $f(\lambda x.g(x), Y) \succ_{\text{cpo}} g(Y)$, but not $f(\lambda x.g(x), Y) \succ_{\text{cpo}}^+ g(Y)$.

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 appear 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 $S$ of base types is given, and the set $T$ of simple types is generated by the grammar $T ::= S | (T \to T)$. Right-associativity is assigned to $\to$ so that some parentheses in types can be omitted. We postulate two disjoint sets $F$ and $V$, 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 $a : A$ for $a$ of type $A$. In this paper, we let $f$ and $g$ range over the set $F$, $x$ over the set $V$ and $a$ over $F \cup V$.

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

$$a : A_1 \to \cdots \to A_n \to B \quad t_1 : A_1 \quad \ldots \quad t_n : A_n \quad \overline{a(t_1, \ldots, t_n)} : B \quad (a \in F \cup V)$$

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

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

$$f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma), \quad x(t_1, \ldots, t_n)\sigma = \sigma(x) \cdot t_1\sigma \cdots \cdot t_n\sigma.$$

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

- $t = \ell \sigma, t' = r \sigma$ and $\ell \to r \in R$ for some substitution $\sigma$.
- $t = t_1 \cdot t_2, t' = t_1' \cdot t_2$ and $t_1 \to_R t_1'$.
- $t = t_1 \cdot t_2, t' = t_1 \cdot t_2'$ and $t_2 \to_R t_2'$.

$\to_R$ is called the rewrite relation. This paper concerns the well-foundedness of $\to_R$.

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 $f : t_1 \cdot t_2$, or just $f(t_1, t_2)$. 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 $\succ$ on $F$, 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. $s = f(s_1, \ldots, s_m) \land \exists s_i \succ_{\text{horpo}} t_i$.
2. $s = f(s_1, \ldots, s_m), t = t_1 \cdot t_2 \cdots \cdot t_n$ and $f(s_1, \ldots, s_m) \succ \{t_1, \ldots, t_n\}$.
3. $s = f(s_1, \ldots, s_m), t = g(t_1, \ldots, t_n), f \succ g$ and $f(s_1, \ldots, s_m) \succ \{t_1, \ldots, t_n\}$.
4. $s = f(s_1, \ldots, s_m), t = f(t_1, \ldots, t_m), (s_1, \ldots, s_m) \succ_{\text{lex}} (t_1, \ldots, t_m)$ and $f(s_1, \ldots, s_m) \succ \{t_1, \ldots, t_m\}$.
5. $s = s_1 \cdot s_2, t = t_1 \cdot t_2, s_1 \succ_{\text{horpo}} t_1, s_2 \succ_{\text{horpo}} t_2$ and $s \neq t$.

In the above definition, $\succ_{\text{horpo}}$ is the reflexive closure of $\succ_{\text{horpo}}$. $\succ_{\text{lex}}$ lexicographically compares lists of the same length by $\succ_{\text{horpo}}$, and $f(s_1, \ldots, s_m) \succ \{t_1, \ldots, t_n\}$ stands for $\forall i (f(s_1, \ldots, s_m) \succ_{\text{horpo}} t_i \vee \exists j s_j \succ_{\text{horpo}} t_i)$. 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 $\sigma$). If in addition $\to_R$ is compatible with $\succ_{\text{horpo}}$ (i.e., $\ell \succ_{\text{horpo}} r$ for all $\ell \to_R r \in R$), then $\to_R$ is well-founded.

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

$$\text{rec}(0, Y, F) \to Y \quad \text{rec}(s(X), Y, F) \to F(X, \text{rec}(X, Y, F))$$

WST 2022
where $0 : \text{nat}$, $s : \text{nat} \rightarrow \text{nat}$ and $\text{rec} : \text{nat} \rightarrow \text{nat} \rightarrow (\text{nat} \rightarrow \text{nat} \rightarrow \text{nat}) \rightarrow \text{nat}$ 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 $\rightarrow \rho$, we need only to find a precedence $\triangleright$ making $\rightarrow \rho$ compatible with the generated relation $\succ_{\text{horpo}}$. While $\text{rec}(0, Y, F) \succ_{\text{horpo}} Y$ follows from the first condition, $\text{rec}(s(X), Y, F) \succ_{\text{horpo}} F(X, \text{rec}(X, Y, F))$ can be obtained as follows:

\[
\frac{F \succ_{\text{horpo}} F} {\text{rec}(s(X), Y, F) \succ_{\text{horpo}} F(X, \text{rec}(X, Y, F))}
\]

The precedence can be any well-founded ordering on $F$. 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:

\[
\text{add}(0, Y) \rightarrow Y \quad \text{add}(s(X), Y) \rightarrow s(\text{add}(X, Y)) \quad \text{sum}(X) \rightarrow \text{rec}(X, 0, \text{add})
\]

where $\text{add} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$ and $\text{sum} : \text{nat} \rightarrow \text{nat}$. If we ignore the rewrite rule on the right, we need only $\text{add} \triangleright s$ to complete the proof. Since the rule on the right only removes occurrences of $\text{sum}$, it seems harmless. However, $\text{sum}(X) \succ_{\text{horpo}} \text{rec}(X, 0, \text{add})$ is not obtainable due to the type restriction: neither $\text{sum}(X)$ nor $X$ has the same type as $\text{add}$ so the necessary premise $\text{sum}(X) \succ \{\text{add}\}$ 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 $F^*$ be $F \cup (F \times T)$, in which a function symbol is either a function symbol $f \in F$, or an ordered pair $(f, A)$, written as $f_A^*$, where $f \in F$ and $A \in T$. We assume $f_A^* : A$ and $F^* \cap V = \emptyset$. With $F^*$, 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 $f^*(t_1, \ldots, t_n) : B$ for any $B$, where $f^*$ stands for $f_A^* \rightarrow \cdots \rightarrow A_n \rightarrow B$. We will omit the type and write just $f^*$ whenever the type can be deduced from the context. In the above translation from $f(t_1, \ldots, t_n)$ to $f^*(t_1, \ldots, t_n)$, marking the head symbol serves two purposes:

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

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

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

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

---

**WST 2022**

13
In the above definition, $\succeq_*$ is the union of $\succ_*$ and the identity relation on unmarked terms, and $(s_1, \ldots, s_{\text{minar}(f)}) \succ_{\text{lex}}^* (t_1, \ldots, t_n)$ if and only if $\exists i \leq \min(\text{minar}(f)n) (s_i \succeq_*, t_j \land \forall j < i \ s_j = t_j)$. Occurrences of $f^*(s_1, \ldots, s_m)$ in $\text{Select}$ 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 $f^*$ takes at least $\text{minar}(f)$ arguments. Without this restriction, the ordering, denoted by $\succ$, will end up allowing the derivation in Figure 1. By Mono, we get an infinite sequence $f(g) > e(f(g,h)) > e(e(f(g,h),h)) > \cdots$, which shows that $\succ$ is not well-founded. The “minimal arity” restriction is necessary because $\text{Select}$ may cause function symbols to take extra arguments, which is not the case for HORPO, and taking into account the extra arguments in a $\text{Lex}$ step can break well-foundedness, as shown in Figure 1.

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

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

\[
\begin{align*}
\text{sum} & \triangleright \text{rec} & \text{sum}(X) & \succ_* X & \text{Select} & \text{sum} \triangleright 0 & \text{Copy} & \text{sum} \triangleright \text{add} & \text{Copy} \\
\text{sum}(X) & \succ_* \text{rec}(X,0,\text{add}) & \text{Copy} & \text{sum}(X) & \succ_* \text{add} & \text{Copy} & \text{Put} \\
\text{sum}(X) & \succ_* \text{rec}(X,0,\text{add}) & \text{Put}
\end{align*}
\]

Unlike $\succ_{\text{horpo}}$, $\succ_*$ is necessarily transitive, which we exemplify with the rewrite sequence $\text{rec}(s(s(X)),Y,\text{add}) \rightarrow_\mathcal{R} \text{add}(s(X),\text{rec}(s(X),Y,\text{add})) \rightarrow_\mathcal{R} s(\text{add}(X,\text{rec}(s(X),Y,\text{add})))$.

With either $\succ_{\text{horpo}}$ or $\succ_*$, 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 $\text{add} \triangleright s$ and $\text{minar} (\text{rec}) = \text{minar} (\text{add}) = 1$. If we skip the term in the middle and try to directly compare the first and the last in the sequence, $\succ_{\text{horpo}}$ fails unless we further impose $\text{rec} \triangleright s$. This shows that $\succ_{\text{horpo}}$ is not transitive as imposing extra assumptions can be problematic when there are other rewrite rules in the system. On the other hand, with $\succ_*$, we do have the following derivation (with irrelevant part omitted):

\[
\begin{align*}
\text{add} & \triangleright s & \text{add}^*(\text{rec}^*(\ldots),\text{rec}^*(\ldots)) & \succ_* \text{add}(X,\text{rec}(s(X),Y,\text{add})) & \text{Lex} & \text{Copy} \\
\text{add}^*(\text{rec}^*(\ldots),\text{rec}^*(\ldots)) & \succ_* \text{add}(X,\text{rec}(s(X),Y,\text{add})) & \text{Put} \\
\text{add}(\text{rec}^*(\ldots),\text{rec}^*(\ldots)) & \succ_* \text{add}(X,\text{rec}(s(X),Y,\text{add})) & \text{Select} \\
\text{rec}^*(s(s(X)),Y,\text{add}) & \succ_* \text{add}(X,\text{rec}(s(X),Y,\text{add})) & \text{Put}
\end{align*}
\]
In the above derivation, we “select” \texttt{add} in \texttt{rec}*(\texttt{s(s(X))}, Y, \texttt{add}) by \texttt{Select}. This reflects the derivation of \texttt{rec}(\texttt{s(X), Y, F}) \succ \star \texttt{F(X, rec(X, Y, F))}, in which \texttt{F} is selected. This step is not available with \succ_{\text{horpo}}. Also by \texttt{Select}, \texttt{add} gets two arguments, each with a marked head symbol. The arguments are later compared with some arguments on the right-hand side by \texttt{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 \texttt{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 \texttt{Copy}. Consider the system with only one rewrite rule \texttt{f(X) \rightarrow g(X, X)}, where \texttt{f : a \rightarrow a} and \texttt{g : a \rightarrow a \rightarrow a}. This rule is easily oriented by \succ_{\star} (or \succ_{\text{horpo}}) with \texttt{f} \succ \texttt{g}, but its applicative first-order counterpart, \texttt{@(f, X) \rightarrow @(g(X), X)}, cannot be tackled. Even if we do not ignore types and introduce a separate symbol \texttt{@A,B : (A \rightarrow B) \rightarrow A \rightarrow B} for any types \texttt{A} and \texttt{B}, the same problem still arises: @_{a,a}(f, X) \succ_{\star} @_{a,a}(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 \texttt{Select}, which can give function symbols extra arguments, HORPO is not necessarily transitive. We thus incorporate both \texttt{Select} and \texttt{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


WST 2022 15
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 \(\text{RFC}(R)\). By a Theorem of Dershowitz [3], termination of a rewrite system \(R\) is equivalent to termination of \(R\) on \(\text{RFC}(R)\).

For our purposes, we will employ the characterization \(\text{RFC}(R) = (\rightarrow_R \cup \neg \rightarrow_{\text{right}}(R))^*(\text{rhs}(R))\), where \(\neg \rightarrow_{\text{right}}(R)\) is the suffix rewrite relation induced by the system \(\text{right}(R) = \{\ell_1 \rightarrow r \mid (\ell_1 \ell_2 \rightarrow r) \in R, \ell_1 \neq \epsilon \neq \ell_2\}\), cf. [6, Lemma 4] for an equivalent variant.

It might be the case that the relation \(\rightarrow_R\) is never used when computing \(\text{RFC}(R)\). 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 \(\ell \in \text{lhs}(R)\) such that \(\ell\) is a factor of some string from \(\text{RFC}(R)\). Then \(R\) terminates on \(\text{RFC}(R)\), so \(R\) terminates everywhere.

The right barren property is decidable: we first compute the regular language \(L = (\neg \rightarrow_{\text{right}}(R))^*(\text{rhs}(R))\). 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., \(L \cap \Sigma^* \cdot \text{lhs}(R) \cdot \Sigma^*\) is empty.

Our implementation computes the closure of \(\text{rhs}(R)\) under suffix rewriting w. r. t. \(\text{right}(R)\) as follows: We start with a finite state automaton that consists of isolated paths, one for each string in \(\text{rhs}(R)\). 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 \((\ell_1 \rightarrow r) \in \text{right}(R)\), if the automaton contains a path \(p \ell_1 \rightarrow q \in F\) 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 \(\ell_1 \rightarrow r\) 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 \( R = \{bababa \rightarrow abaabbabba\} \), 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 \( 0 \leq s \leq |r| \).

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

We observe that the left-hand side \( \ell = bababa \) 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 \( R = \{aa \rightarrow cb, bb \rightarrow ac, cc \rightarrow ba\} \) (Zantema_04/z087). For \( |R| > 1 \), we choose states as pairs \( (n, s) \) with \( 1 \leq n \leq |R| \) and, as before, \( s \) with \( 0 \leq s \leq |r| \) for \( r \in \text{rhs}(R) \).

Here, completion starts with an automaton consisting of three paths, one for each right-hand side. For rule \( (a \rightarrow cb) \in \text{right}(R) \) 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 \((b \rightarrow ac) \in \text{right}(R)\) at \((1,1)\), and from \((2,1)\) to \((3,0)\) due to the suffix match of \((c \rightarrow ba) \in \text{right}(R)\) at \((2,1)\). The resulting automaton is closed w. r. t. \text{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 RFC\((R)\).

We now describe how to continue in this case.

We call a rule \( \ell \rightarrow r \) from \( R \) relatively right barren w. r. t. the other rules, if \( \ell \) does not occur as a factor of 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 \( \text{RFC}(R) \), 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 \( \ell \rightarrow r \) in \( R \) and a redex path \( p \rightarrow q \) 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 \( \text{rhs}(R) \), and \( A \) is closed w. r. t. \( \rightarrow_{\text{right}} \) and \( \rightarrow_{\text{right}(R)} \), so \( L(A) \) over-approximates \( \text{RFC}(R) \). 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.

\[ \text{Example 3.} \text{ Let } R = \{ab \rightarrow ba, ba \rightarrow acb\} (\text{Zantema}_04/z006). \text{ First we add two epsilon transitions, } (1, 1) \rightarrow (1, 0) \text{ due to the suffix match of } (a \rightarrow ba) \in \text{right}(R) \text{ at } (1, 1), \text{ and } (2, 2) \rightarrow (2, 0) \text{ due to the suffix match of } (b \rightarrow acb) \in \text{right}(R) \text{ at } (2, 2). \]

As there is a redex path \( (1, 0) \xrightarrow{b_a} (1, 2) \) for the second rule, we add the epsilon transitions \( (1, 0) \rightarrow (2, 0) \) and \( (2, 3) \rightarrow (1, 2) \). This creates a new redex path \( (1, 0) \xrightarrow{b_a} (2, 1) \) for the second rule (note that this path uses two epsilon transitions), resulting in \( (1, 0) \rightarrow (2, 0) \) (already present) and \( (2, 3) \rightarrow (2, 1) \) (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 \( \text{right}(R) \), so the completion procedure stops. As there is no path labelled by \( ab \), the rule \( ab \rightarrow ba \) can be removed from \( R \). The remaining system \( \{ba \rightarrow acb\} \) is terminating (it is right barren, in fact), proving termination of \( R \).

\section{Approximating Match-Bounds}

We refine the approximation of \( \text{RFC}(R) \) 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 \( \text{rhs}(R) \) 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 \( (\ell_1 \rightarrow r) \in \text{right}(R) \), 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 redex path at height \( h + 1 \). If we succeed, we obtain an automaton certifying that \( R \) is match-bounded by \( B \) on \( \text{RFC}(R) \), 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.

\[ \text{Example 4.} \text{ Let } R = \{abac \rightarrow baabba\}, \text{ the reversal of Zantema}_04/z034. \text{ Our algorithm produces a certificate for match-bound 1 on RFC}(R), \text{ as follows.} \]
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 \(a \rightarrow baabbaa \in \text{right}(R)\) at \((0,1,6)\) we add the transition \((0,1,6) \xrightarrow{\epsilon} (0,1,0)\). This creates the \(R\)-redex path \((0,1,5) \xrightarrow{\epsilon} (0,1,6) \xrightarrow{\epsilon} (0,1,0) \cdots (0,1,3) \xrightarrow{b} (0,1,4)\) 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 \(a \rightarrow baabbaa,\) since \((1,1,7)\) is not final, but for \(abaa \rightarrow baabbaa,\) via the path \((1,1,6) \xrightarrow{a} (1,1,7) \xrightarrow{\epsilon} (0,1,4) \xrightarrow{b} (0,1,7)\). 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 \(< B\). 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 \(RFC(R)\), and thus, they terminate relatively to \(R\). The method of Section 2 is the special case of \(B = 0\).

Example 5. For \(R = \{aba \rightarrow baa, aac \rightarrow acab\}\), 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.

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

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
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.

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:

![Hydra Tree Diagram](image)

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 $|$ for siblings, and a unary function symbol $i$ representing the internal nodes. We use infix notation for $|$ and declare it to be an AC symbol.
Example 2. The Hydras in the above example fight are represented by the terms

\[ H_1 = i(i(h) | i(i(h) | i(h))) | h) \]
\[ H_2 = i(i(h) | i(i(h) | h | h)) | h) \]
\[ H_3 = i(i(h) | i(i(h) | h) | i(i(h) | h)) | h) \]
\[ H_4 = i(h | h | i(i(h) | h) | i(i(h) | h) | i(i(h) | h)) | h | h) \]
\[ H_5 = i(h | h | i(i(i(h) | h) | i(i(h) | h) | i(i(h) | h)) | h | h) \]

Definition 3. The TRS \( H \) consists of the following 14 rewrite rules:

\[
\begin{align*}
A(n, i(h)) & \rightarrow_1 h & D(n, i(i(x))) & \rightarrow_8 i(D(n, i(x))) \\
A(n, i(h | x)) & \rightarrow_2 A(s(n), i(x)) & D(n, i(i(h | x) | y)) & \rightarrow_9 i(D(n, i(x)) | y) \\
A(n, i(x)) & \rightarrow_3 B(n, D(s(n), i(x))) & D(n, i(i(h | x) | y)) & \rightarrow_{10} i(C(n, i(x)) | y) \\
C(0, x) & \rightarrow_4 E(x) & D(n, i(i(h | x) | y)) & \rightarrow_{11} i(C(n, i(x))) \\
C(s(n), x) & \rightarrow_5 x | C(n, x) & D(n, i(i(h) | y)) & \rightarrow_{12} i(C(n, h) | y) \\
i(i(E(x) | y) | y) & \rightarrow_6 E(i(x) | y)) & D(n, i(i(i(h)))) & \rightarrow_{13} i(C(n, h)) \\
i(E(x)) & \rightarrow_7 E(i(x)) & B(n, E(x)) & \rightarrow_{14} A(s(n), x)
\end{align*}
\]

The Battle is started with the term \( A(0, t) \) where \( t \) is the term representation of the initial Hydra. Rule 1 takes care of the dying Hydra. 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 \( T(\{h,i,\}) \setminus \{h\} \) of successive Hydras in an arbitrary battle then \( A(n, H) (=_{AC} \cdot \rightarrow \cdot =_{AC})^n A(s(n), H') \) for some \( n \in T(\{0,s\}) \).

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

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

\[
\begin{align*}
A(0, H_1) & \rightarrow_3 B(0, D(s(0), H_1)) \\
 (=_{AC} \cdot \rightarrow_9 B(0, i(D(s(0), i(i(i(h) | i(h)))) | i(h) | h)) & \rightarrow_8 B(0, i(i(D(s(0), i(i(h) | i(h)))) | i(h) | h)) \\
& \rightarrow_{12} B(0, i(i(D(s(0), i(h)) | i(h)) | i(h) | h)) \\
& \rightarrow_5 B(0, i(i(i(h) | C(0, h) | i(h))) | i(h) | h)) \\
& \rightarrow_4 B(0, i(i(i(h) | E(h) | i(h))) | i(h) | h)) \\
(=_{AC} \cdot \rightarrow_6 B(0, i(i(E(i(h | h) | i(h)))) | i(h) | h)) & \rightarrow_7 B(0, i(E(i(i(h | h) | i(h)))) | i(h) | h)) \\
& \rightarrow_{14} A(s(0), i(i(i(h | h) | i(h))) | i(h) | h)) =_{AC} A(s(0), H_2)
\end{align*}
\]
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 $\phi : A_1 \times \cdots \times A_n \rightarrow A$ is

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

Definition 7. An $S$-sorted $\mathcal{F}$-algebra $A = ([S_A]_{S \in S}, \{f_A\}_f \in \mathcal{F})$ equipped with a strict order $>$ on the union of all carriers $S_A$ is simple monotone if

- every carrier $S_A$ is non-empty,
- $(S_i)_A \subseteq S_A$ for all $f : S_1 \times \cdots \times S_n \rightarrow S$ in $\mathcal{F}$ and $1 \leq i \leq n$, and
- every interpretation function $f_A$ is simple and weakly monotone.

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

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 $(A, >)$ such that $\mathcal{R} \subseteq >_A$, $AC \subseteq =_A$, and $f_A$ 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 $\{f^{(1)}, g^{(1)}, |^{(2)}\}$:

$$f(g(x | y)) \rightarrow g(f(f(x | y)))$$

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

$$f_A(x) = x + 1 \quad g_A(x) = x + \omega \quad x |_A y = x \oplus y$$

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, $AC \subseteq =_A$ holds. Moreover, $\mathcal{R} \subseteq >_A$ is verified by the inequality

$$f_A(g_A(x |_A y)) = (x \oplus y) + \omega + 1 > (x \oplus y) + \omega = (x \oplus y) + 2 + \omega = g_A(f_A(f_A(x |_A y)))$$

where $\omega + 1 > \omega$ and $2 + \omega = \omega$ are used. Hence $\mathcal{R}$ is AC terminating.
1:4 Hydra Battles and AC Termination

It is essential for Theorem 8 to demand \( f_A(x_1, \ldots, x_i, \ldots, x_n) \geq x_i \) even if \( f \) has the sort declaration \( S_1 \times \cdots \times S_i \times \cdots \times S_n \rightarrow S \) with \( S_i \neq S \).

Example 10. Consider the non-terminating TRS \( \mathcal{R} = \{ a \rightarrow g(f(a)) \} \) over the signature \( \{ a : A, f : A \rightarrow B, g : B \rightarrow A \} \) and the algebra \( \mathcal{A} \) with carriers \( A_A = B_A = N \) and interpretation functions \( a_A = 1, f_A(x) = 0 \) and \( g_A(x) = x \). Observe that the argument sort (A) and output sort (B) of \( f \) are different. If the requirement of \( f_A(x) \geq x \) 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 \( \mathcal{F}' \):

\[
\begin{align*}
\text{type declarations:} & \quad h : \mathcal{O} & i, \iota : \mathcal{O} \rightarrow \mathcal{O} & \quad | : \mathcal{O} \times \mathcal{O} \rightarrow \mathcal{O} & \quad 0 : \mathcal{N} & \quad s : \mathcal{N} \rightarrow \mathcal{N} & \quad A, B, C, D : \mathcal{N} \times \mathcal{O} \rightarrow \mathcal{O}
\end{align*}
\]

where \( \mathcal{O} \) and \( \mathcal{N} \) 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 \( \mathcal{O}_A = (\{0, 1\} \times \mathcal{N} \times \mathcal{N}) \) and \( \mathcal{N}_A = (\mathcal{N} \setminus \{0, 1\}) \times \mathcal{N} \times \mathcal{N} \) and the following interpretation functions, where we write \( \vec{n} \) for \((n_1, n_2, n_3) \in \mathcal{N}_A \) and \( \vec{x} \) and \( \vec{y} \) for \((x_1, x_2, x_3), (y_1, y_2, y_3) \in \mathcal{N}_A^3 \):

\[
\begin{align*}
0_A &= h_A = (2, 0, 0) & A_A(\vec{n}, \vec{x}) &= (n_1 + x_1, n_2 + 2x_2, 2, 0) \\
i_A(\vec{x}) &= (\omega^{x_1}, x_2 + 1, x_3 + 1) & B_A(\vec{n}, \vec{x}) &= (2 + n_1 + x_1, n_2 + 2x_2 + 1, 0) \\
\iota_A(\vec{y}) &= x_1 & C_A(\vec{n}, \vec{x}) &= (x_1 \cdot n_1, 0, 0) \\
D_A(\vec{n}, \vec{x}) &= (n_1 + x_1, n_2 + x_2, n_3 + x_2 + 2x_3) & E_A(\vec{x}) &= (x_1, x_2 + 1, 0)
\end{align*}
\]

The carriers \( \mathcal{O}_A \) and \( \mathcal{N}_A \) are equipped with the lexicographic order \( >_\mathcal{O} \) on \( \mathcal{O}_A \). We write \( >_\mathcal{N} \) for the restriction of \( >_\mathcal{O} \) to \( \mathcal{N}_A \). 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 \( \iota \) symbols that do not occur below \( \iota \). First we argue that \( \mathcal{A} \) is simple monotone and \( |_A \) is strictly monotone. Due to lack of space, we only treat \( C_A \).

\( C_A(\vec{n}, \vec{x}) \geq_\mathcal{O} C_A(\vec{m}, \vec{x}) \) holds if \( \vec{n} >_\mathcal{N} \vec{m} \), because \( (x_1 \cdot n_1, 0, 0) \geq_\mathcal{O} (x_1 \cdot m_1, 0, 0) \) follows from \( x_1 \cdot n_1 \geq_\mathcal{O} x_1 \cdot m_1 \).
Next we argue that $\mathcal{A}$ is compatible with $H$. For brevity we treat only rules 3, 5, 7, 13 and omit unimportant elements in vectors.

- $A_\mathcal{A}(\vec{n}, i(\vec{x})) = (\omega x_1, n_2 + 2x_2 + 3, -) >_O (\omega x_1, n_2 + 2x_2 + 3, -) = A_\mathcal{A}(\vec{n}, D_\mathcal{A}(s_\mathcal{A}(\vec{n}), i_\mathcal{A}(\vec{x})))$

Here $\omega x_1$ is obtained from $n + \omega x_1 = \omega x_1$ and $n + 2 + \omega x_1 = \omega x_1$ by exploiting the condition $n_1 \in N$.

- $C_\mathcal{A}(s_\mathcal{A}(\vec{n}), \vec{x}) = (x_1 \cdot (n_1 + 2), -) >_O (x_1 \cdot (n_1 + 1), -) = \vec{x} | \mathcal{A} C_\mathcal{A}(\vec{n}, \vec{x})$

Remark that $s_\mathcal{A}(\vec{n})$ is defined as $(n_1 + 2, 0, 0)$ rather than $(n_1 + 1, 0, 0)$.

- $i_\mathcal{A}(E_\mathcal{A}(\vec{x})) = (\omega x_1, x_2 + 2, 1) >_O (\omega x_1, x_2 + 2, 0) = E_\mathcal{A}(i_\mathcal{A}(\vec{x}))$

- $D_\mathcal{A}(\vec{n}, i_\mathcal{A}(h_\mathcal{A}(\vec{x}))) = (\omega ^2, -) >_O (\omega ^2, -) = i_\mathcal{A}(C_\mathcal{A}(\vec{n}, h_\mathcal{A}(\vec{x})))$

It follows from $\omega ^2 > 2n_1$ due to $n_1 \in N$. 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**


A Calculus for Modular Non-Termination Proofs
by Loop Acceleration

Florian Frohn
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 $\psi$ from a single-path loop $T$, i.e., a loop without branching in its body, such that $\psi$ under-approximates (or is equivalent to) $T$. 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 $\vec{x}, \vec{y}, \vec{z}, \ldots$ for vectors. We consider loops of the form

$$\textbf{while } \varphi \textbf{ do } \vec{x} \leftarrow \vec{a}$$

($\mathcal{T}_{\text{loop}}$)

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

We identify $\mathcal{T}_{\text{loop}}$ 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 $\varphi$) with the predicate $\vec{x} \mapsto \varphi$.

Our goal is to prove non-termination of $\mathcal{T}_{\text{loop}}$.

**Definition 1 ((Non-)Termination).** We call a vector $\vec{x} \in \mathbb{Z}^d$ a witness of non-termination for $\mathcal{T}_{\text{loop}}$ (denoted $\vec{x} \rightarrow^{\varphi(\vec{x})} \perp$) 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 $\mathcal{T}_{\text{loop}}$ is non-terminating. Otherwise, $\mathcal{T}_{\text{loop}}$ 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) - 259590644 (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 $\eta \in \text{Conj}(\vec{x})$ a certificate of non-termination for $T_{\text{loop}}$ if it is satisfiable and the implication $\eta(\vec{x}) \implies \vec{x} \rightarrow^{\infty}_{(\vec{\eta}, \vec{a})} \perp$ 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 $\epsilon(\vec{x}) \leq \epsilon(\vec{a}(\vec{x})) \implies \epsilon(\vec{a}(\vec{x})) \leq \epsilon(\vec{a}^2(\vec{x}))$ holds for each inequation $\epsilon(\vec{x}) > 0$ in $\varphi$. 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 $T_{\text{loop}}$.

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 $\|\psi | \varphi | \vec{\varphi}\|_d$ where $\psi, \varphi, \vec{\varphi} \in \text{Conj}(\vec{x})$ and $\vec{a} : \mathbb{Z}^d \rightarrow \mathbb{Z}^d$ is a non-termination problem. It is consistent if every model of $\psi$ is a witness of non-termination for $\langle \varphi, \vec{a} \rangle$ and solved if it is consistent and $\vec{\varphi} \equiv \top$. The canonical non-termination problem of a loop $T_{\text{loop}}$ is $\|\top | \top | \varphi\|_d$.

Example 4. Consider the loop

\[
\text{while } x_1 > 0 \text{ do } (x_1) \leftarrow (x_1 + x_2). \quad (T_{\text{ev-inc}})
\]

Its canonical non-termination problem is $\|\top | \top | x_1 > 0\|_d (x_1, x_2)$, which is consistent as $\langle \varphi, \vec{a} \rangle = \langle \top, (x_1, x_2) \rangle$ diverges for all valuations of $x_1$ and $x_2$, but not solved as $\vec{\varphi} \equiv x_1 > 0 \neq \top$.

The first component $\psi$ of a non-termination problem $\|\psi | \varphi | \vec{\varphi}\|_d$ is the partial result that has been computed so far. The second component $\varphi$ corresponds to the part of the loop condition that has already been processed successfully. As our calculus preserves consistency, $\psi$ is always a certificate of non-termination for $\langle \varphi, \vec{a} \rangle$. 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 $\langle \varphi, \vec{a} \rangle$.

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 $\|\top | \top | \varphi\|_d$ to $\|\psi_1 | \varphi | \vec{\varphi}\|_d$, then $\varphi \equiv \varphi \land \vec{\varphi}$ and $\psi_1 \implies \vec{x} \rightarrow^{\infty}_{(\varphi, \vec{a})} \perp$. Then it suffices to find some $\psi_2 \in \text{Conj}(\vec{x})$ such that $(\vec{x} \rightarrow^{\infty}_{(\varphi, \vec{a})} \perp \land \psi_2) \implies \vec{x} \rightarrow^{\infty}_{(\varphi, \vec{a})} \perp$. The reason is that we have $\rightarrow(\varphi, \vec{a}) \cap \rightarrow(\varphi, \vec{a}) = \rightarrow(\varphi, \vec{a})$ and thus $\psi_1 \land \psi_2 \implies \vec{x} \rightarrow^{\infty}_{(\varphi, \vec{a})} \perp$, i.e., $\psi_1 \land \psi_2$ is a certificate of non-termination for $T_{\text{loop}}$.

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 $nt : \text{Loop} \times \text{Conj}(\vec{x}) \rightarrow \text{Conj}(\vec{x})$ be a partial function. We call $nt$ a conditional non-termination technique if

\[
\vec{x} \rightarrow^{\infty}_{(\varphi, \vec{a})} \perp \land nt((\chi, \vec{a}), \vec{\varphi}) \quad \text{implies} \quad \vec{x} \rightarrow^{\infty}_{(\chi, \vec{a})} \perp \quad \text{for all } ((\chi, \vec{a}), \vec{\varphi}) \in \text{dom}(nt), \vec{x} \in \mathbb{Z}^d.
\]

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 $\{e_1 > 0, \ldots, e_n > 0\}$:

\[
\|\psi_1 | \varphi | \vec{\varphi}\|_{nt} \rightsquigarrow_{nt} \|\psi_1 \land \psi_2 | \varphi \land \chi | \vec{\varphi} \setminus \chi\|_d
\]

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

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

Moreover, termination of \(\sim_{nt}\) is trivial, as each step removes an inequation from \(\bar{\varphi}\). It remains to present conditional non-termination techniques that can be used with our novel calculus. We first derive a non-conditional non-termination technique from \textit{Acceleration via Monotonic Increase} \cite{5}.

▶ Theorem 8 (Non-Termination via Monotonic Increase). If \(\bar{\varphi}(\vec{x}) \land \chi(\vec{x}) \implies \chi(\bar{a}(\vec{x}))\), then \((\chi, \bar{a}, \bar{\varphi}) \mapsto \chi\) is a conditional non-termination technique.

▶ Example 9. Consider the following loop:

\[
\text{while } x > 0 \text{ do } x \leftarrow x + 1 \quad (T_{\text{inc}})
\]

Its canonical non-termination problem is \(\|\top|\top|_{(x+1)}\). Thus, in order to apply \(\sim_{nt}\) with Thm. 8, the only possible choice for the formula \(\chi\) is \(x > 0\). Furthermore, we have \(\bar{\varphi} := \top\) and \(\bar{a} := (x + 1)\). Hence, Thm. 8 is applicable if the implication \(x > 0 \implies x + 1 > 0\) is valid, which is clearly the case. Thus, we get \(\|\top|\top|_{(x+1)} \sim_{nt} \|x > 0 \land x > 0 \land \top\|_{(x+1)}\). Since the latter non-termination problem is solved and \(x > 0\) is satisfiable, \(x > 0\) is a certificate of non-termination for \(T_{\text{inc}}\) 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 \textit{Acceleration via Eventual Increase} \cite{5}.

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

\[
\bar{\varphi}(\vec{x}) \land e(\vec{x}) \leq e(\bar{a}(\vec{x})) \implies e(\bar{a}(\vec{x})) \leq e(\bar{a}^2(\vec{x})),
\]

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

\[
((\chi, \bar{a}, \bar{\varphi}) \mapsto \bigwedge_{e(\vec{x}) > 0 \in \chi} 0 < e(\vec{x}) \leq e(\bar{a}(\vec{x})))
\]

▶ Example 11. We continue Ex. 4. To apply \(\sim_{nt}\) with Thm. 10 to the canonical non-termination problem of \(T_{\text{inc}}\), the only possible choice for the formula \(\chi\) is \(x_1 > 0\). Moreover, we again have \(\bar{\varphi} := \top\), and \(\bar{a} := (x_1 + x_2)\). Thus, Thm. 10 is applicable if \(x_1 \leq x_1 + x_2 \iff x_1 + x_2 \leq x_1 + 2 \cdot x_2 + 1\) 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 \(\|\top|\top|_{(x_1 + x_2)} \sim_{nt} \|0 < x_1 \leq x_1 + x_2 \land x_1 > 0 \land \top\|_{(x_1 + x_2)}\). Since \(0 < x_1 \leq x_1 + x_2 \equiv x_1 > 0 \land x_2 \geq 0\) is satisfiable, \(x_1 > 0 \land x_2 \geq 0\) is a certificate of non-termination for \(T_{\text{inc}}\) due to Thm. 7.

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

▶ Example 12. Consider the loop

\[
\text{while } x_1 > 0 \text{ do } \langle \begin{cases} x_1 \leftarrow x_1 + x_2 \\ x_2 \leftarrow x_1 \end{cases} \rangle. \quad (T_{\text{fixpoint}})
\]

Thm. 8 is inapplicable, since \(x_1 > 0 \notimplies x_2 > 0\). Furthermore, Thm. 10 is inapplicable, since \(x_1 \leq x_2 \notimplies x_2 \leq x_1\).

However, \(T_{\text{fixpoint}}\) has fixpoints, i.e., there are valuations such that \(\vec{x} = \bar{a}(\vec{x})\). Therefore, it can be handled by existing approaches like \cite{7, 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 \(V(e)\) denote the set of variables occurring in \(e\). Moreover, we define \(\text{closure}_x(e) := \bigcup_{n \in \mathbb{N}} V(\bar{a}^n(x))\). Then

\[
((\chi, \bar{a}, \bar{\varphi}) \mapsto \bigwedge_{e(\vec{x}) > 0 \in \chi} e(\vec{x}) > 0 \land \bigwedge_{x_j \in \text{closure}_x(\chi)} x_j = \bar{a}(\vec{x})_j)
\]

is a conditional non-termination technique.
Example 14. We continue Thm. 12 by showing how to apply Thm. 13 to $T_{\text{fixpoint}}$, i.e., we have $\chi := x_1 > 0$, $\bar{\phi} := \top$, and $\vec{a} := (\vec{x}_2)$. Thus, we get $\text{closure}_2(x_1 > 0) = \{x_1, x_2\}$. So starting with the canonical non-termination problem of $T_{\text{fixpoint}}$, we get $\| \top \mid \top \mid x_1 > 0 \|_{\vec{x}_1} \sim_{\text{nt}} \| x_1 > 0 \land x_1 = x_2 \mid x_1 > 0 \mid \top \|_{\vec{x}_2}$. Since the formula $x_1 > 0 \land x_1 = x_2$ is satisfiable, it is a certificate of non-termination for $T_{\text{fixpoint}}$ 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:

\[
\text{while } x_1 > 0 \land x_3 > 0 \land x_4 + 1 > 0 \text{ do } \left( \begin{array}{c}
\bar{x}_1 \\
\bar{x}_2 \\
\bar{x}_3 \\
\bar{x}_4
\end{array} \right) \leftarrow \left( \begin{array}{c}
\bar{x}_1 + x_1 \\
\bar{x}_2 + x_2 \\
\bar{x}_3 + x_3 \\
\bar{x}_4 - x_4
\end{array} \right)
\]

So we have $\phi := x_1 > 0 \land x_3 > 0 \land x_4 + 1 > 0$ and $\vec{a} := (\vec{x}_1)$.

Then the canonical non-termination problem is $\| \top \mid \top \mid x_1 > 0 \land x_3 > 0 \land x_4 + 1 > 0 \|_{\vec{x}_1}$. First, our implementation applies Thm. 8 to $x_1 > 0$ (as $x_1 > 0 \implies 1 > 0$), resulting in $\| x_1 > 0 \mid x_1 > 0 \land x_3 > 0 \land x_4 + 1 > 0 \|_{\vec{x}_1}$. Next, it applies Thm. 10 to $x_3 > 0$, which is possible since $x_1 > 0 \land x_3 \leq x_3 + x_2 \implies x_3 + x_2 < x_3 + 2 \cdot x_2 + x_1$ 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 $\| x_1 > 0 \land x_2 > 0 \land x_1 > 0 \mid x_1 > 0 \land x_3 > 0 \mid x_4 + 1 > 0 \|_{\vec{x}_1}$.

Here, we simplified $0 < x_3 \leq x_3 + x_2$ to $x_2 > 0 \land x_3 > 0$. 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 $(-1)^n \cdot x_4^{\text{init}}$, where $x_4^{\text{init}}$ denotes the initial value of $x_4$. Thus, we apply Thm. 13 and we get $\| x_1 > 0 \land x_2 > 0 \land x_3 > 0 \land x_4 = 0 \mid \phi \mid \top \|_{\vec{x}_1}$, 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 $x_1 > 0 \land x_2 > 0 \land x_3 > 0 \land x_4 = 0$ 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.

2 https://sprove-developers.github.io/LoAT/
F. Frohn and C. Fuhs

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:

\[
\text{while } x > 9 \land x \geq 0 \text{ do } (x, x_1) \leftarrow \left(\frac{x^2 + 2x_1 + 1}{x_1 + 1}\right)
\]

We conjecture that the other tools fail for this example due to the presence of non-linear arithmetic.

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

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 \( S \) 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 \( S \). 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 \( S \) has the form \texttt{while} (\( \varphi \)) \texttt{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 \( S \) and \( A \) is a \( d \times d \) matrix over \( S \). For loops \texttt{while} (\( \varphi \)) \texttt{do} \( \vec{x} \leftarrow A \cdot \vec{x} + \vec{c} \), note that \( \vec{c} \subseteq S^d \) can be eliminated by introducing an additional variable. The guard \( \varphi \) is a quantifier-free formula over the atoms \{ \( f \supset 0 \mid f \in S[\vec{x}]_{\text{lin}}, \supset \in \{\geq, >\} \} \), where \( S[\vec{x}]_{\text{lin}} \) contains all polynomials of degree at most 1.

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

While \((\phi)\) do
\[
\vec{x} ← A \cdot \vec{x}
\]
\[
A = \begin{bmatrix}
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 2 & 1 & 0 \\
0 & 0 & 0 & 2 & 1 \\
0 & 0 & 0 & 0 & 2
\end{bmatrix}
\]
\[
\tilde{q} = \begin{bmatrix}
x_1 + n \cdot x_2 \\
x_4 + 2n + 2n^2 \\
x_5 + 2n + n^2 \cdot 2n
\end{bmatrix}
\]

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 \(L ⊆ \Sigma^* \times \mathbb{N}\), where \(\Sigma\) is a finite alphabet. The second component is the parameter of the problem. \(L\) is an element of the complexity class \(XP\) if the time needed for deciding the question “\(x, k\) ∈ \(L\)?” is in \(O(|x|^f(k))\) 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: \(((\phi, A \cdot \vec{x}), k) \in L_{k\text{-termination}}\) if the loop \((\phi, A \cdot \vec{x})\) terminates over \(S\) 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 \(\mathbb{R}_k\) is \(\text{Co-NP}\)-complete). Intuitively, our class prohibits multiple updates like \(x_i ← x_i\) where variables “stabilize”, as termination is essentially equivalent to unsatisfiability of the guard for such loops.

Deciding Termination

To decide termination, we first transform the uniform loop \((\phi, A \cdot \vec{x})\) such that the update matrix is in Jordan normal form. Let \(λ_1 < \ldots < λ_k\) 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 \(η(\vec{x}) = T^{-1} \cdot \vec{x}\). As shown in [3, 4], instead of termination of the original loop on \(S^d\), we can prove termination of the transformed loop on \((η^{-1}(\phi), Q \cdot \vec{x}) = (\phi', Q \cdot \vec{x})\) on \(T \cdot S^d\). Here, \(η^{-1}(\phi)\) results from \(\phi\) by applying \(η^{-1}\) to all polynomials that occur in \(\phi\). For \(S \in \{\mathbb{Q}, \mathbb{R}_k\}\), the transformation matrix \(T\) is an invertible matrix over \(S\). Therefore, we obtain \(T \cdot S^d = S^d\), i.e., we now have to analyze termination of \((\phi', Q \cdot \vec{x})\) over \(S\). In contrast, if \(S = \mathbb{Z}\), then the transformation matrix \(T\) or its inverse may contain non-integer rational numbers. Thus, we focus on uniform loops over \(S = \{\mathbb{Q}, \mathbb{R}_k\}\) and refer to [4] for \(S = \mathbb{Z}\).

So we now assume that the update matrix of our loop \((\phi, A \cdot \vec{x})\) is in Jordan normal form. As shown in [4], one can easily compute a normalized closed form \(\tilde{q}\) of \(A\), i.e., \(\tilde{q}\) is a vector of \(d\) arithmetic expressions over \(\vec{a}\) and a designated variable \(n\) such that \(\tilde{q} = A^n \cdot \vec{x}\) for all large enough \(n \in \mathbb{N}\) (see Fig. 1 for such a \(\tilde{q}\) in our example). Then \((\phi, A \cdot \vec{x})\) is non-terminating iff
\[
\exists \vec{x} \in S^d, \quad n_0 \in \mathbb{N}, \quad \forall n \in \mathbb{N}_{>n_0}, \quad \phi(\tilde{q}) \quad \text{is valid,}
\]
where \(\phi(\tilde{q}) = \phi(\vec{x}/\tilde{q})\). To check this, we examine the dominant terms in \(\phi(\tilde{q})\)'s inequations.

Let \(p > 0\) occur in \(\phi(\tilde{q})\), where \(\triangleright \in \{≥, >\}\). Then we order \(p\)'s addends according to their asymptotic growth w.r.t. \(n\). Here, let \(Q_S = \{\frac{r}{s} \mid r, s ∈ S_{>0}\}\) be the quotient field of \(S\).

Definition 4 (Ordering Coefficients). Marked coefficients are of the form \(α^{(b,a)}\) where \(α ∈ Q_S[\vec{x}]_{\mathbb{N}}, b ∈ S_{>0}\), and \(a ∈ \mathbb{N}\). We define \(\text{unmark}(α^{(b,a)}) = α\) and \(α^{(b_1,a_1) ⊆_{\text{coeff}} α^{(b_2,a_2)}\) if
$b_1 < b_2$ or $b_1 = b_2 \land a_1 < a_2$. Let $p = \sum_{j=1}^{\ell} \alpha_j \cdot n^{a_j} \cdot b_j$, where $\alpha_j \neq 0$ for all $1 \leq j \leq \ell$. Then the coefficients of $p$ are $\text{coeffs}(p) = \{0^{(1.0)}\}$, if $\ell = 0$, and $\text{coeffs}(p) = \{\alpha_j^{(b_j,a_j)} | 1 \leq j \leq \ell\}$ otherwise. W.l.o.g., let $\alpha_j^{(b_j,a_j)} < \text{coeff} \alpha_j^{(b_j,a_j)}$ 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 $\varphi(q) = f(q) > 0 \land f'(q) > 0 = p > 0 \land p' > 0$. We have $p = f(q_1, \ldots, q_5) = -q_1 + 3 \cdot q_3 + 4 = (-x_1 + 4) - x_2 \cdot n + 3 \cdot x_3 \cdot 2^n + \left(\frac{3n^3}{2} - \frac{3nx}{5}\right) \cdot n \cdot 2^n + \frac{3nx}{5} \cdot n^2 \cdot 2^n$. (2)

So $\text{coeffs}(p) = \{\alpha_1^{(1.0)}, \alpha_2^{(1.1)}, \alpha_3^{(2.0)}, \alpha_4^{(2.1)}, \alpha_5^{(2.2)}\}$, where $\alpha_1 = -x_1 + 4, \alpha_2 = -x_2, \alpha_3 = 3 \cdot x_3, \alpha_4 = \frac{3n^3}{2} - \frac{3nx}{5}$, and $\alpha_5 = \frac{3nx}{5}$.

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, $\exists \vec{x} \in \mathcal{S}^d, n_0 \in \mathbb{N} \forall n \in \mathbb{N}_{>n_0}, p > 0$ holds iff there is a $\vec{v} \in \mathcal{S}^d$ such that unmark $\{\max \{\text{coeff}(p(\vec{v}))\} \} > 0$. This is equivalent to satisfiability of $\text{red}(p > 0)$, where

\[
\text{red}(p > 0) = \bigvee_{i=1}^{\ell} (\alpha_i > 0 \land \bigwedge_{j=i+1}^{\ell} \alpha_j = 0) \quad \text{and} \quad \text{red}(p \geq 0) = \text{red}(p > 0) \lor \bigwedge_{i=1}^{\ell} \alpha_i = 0.
\]

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

\[
(\alpha_1 > 0 \land \alpha_2 = 0 \land \ldots \land \alpha_5 = 0) \lor (\alpha_2 > 0 \land \alpha_3 = 0 \land \ldots \land \alpha_5 = 0) \\
\lor (\alpha_3 > 0 \land \alpha_4 = 0 \land \alpha_5 = 0) \lor (\alpha_4 > 0 \land \alpha_5 = 0)\quad \lor \alpha_5 > 0.
\]

To lift our reduction to quantifier-free formulas $\xi$, let the formula $\text{red}(\xi)$ result from $\xi$ by replacing each atom $p > 0$ in $\xi$ by $\text{red}(p > 0)$. 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 $q$ is non-terminating iff $\exists \vec{x} \in \mathcal{S}^d$. $\text{red}(\varphi(q))$ is valid.

## 4 Interval Conditions

For uniform loops, $\text{red}(p > 0)$ can be expressed as a disjunction of *interval conditions*, whose satisfiability is particularly easy to check. More precisely, for any $f = f(q)$ 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 $\bigwedge_{i=1}^{\ell} (\alpha_j = 0)$.

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

\[
(-x_1 + 4 > 0 \land \bigwedge_{j=2}^{5} x_j = 0) \lor (-x_2 > 0 \land \bigwedge_{j=3}^{5} x_j = 0) \\
\lor (x_3 > 0 \land x_4 = 0 \land x_5 = 0) \lor (x_4 > 0 \land x_5 = 0)\quad \lor (x_5 > 0).
\]

The formulas $\rho_{f,i}$ 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 $0 \neq c \in \mathbb{Q}_s$, an interval condition has one of the following forms:

\[
(a) \quad sg \cdot x_i > 0 \land \bigwedge_{j \in I \setminus \{i\}} (x_j = 0) \\
(b) \quad sg \cdot x_i > 0 \land \bigwedge_{j \in I} (x_j = 0)
\]
Deciding Termination of Uniform Loops

\[(c)\] \[x_i' = c \land \bigwedge_{j \in I \setminus \{i'\}} (x_j = 0)\]
\[(d)\] \[sg \cdot x_i > 0 \land x_i' = c \land \bigwedge_{j \in I \setminus \{i, i'\}} (x_j = 0)\]
\[(e)\] \[sg \cdot x_i + c > 0 \land \bigwedge_{j \in I \setminus \{i\}} (x_j = 0)\]

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

**Definition 10 (Evaluation).** Let \(p\) be a formula built from \(\land, \lor, \land\), and atoms of the form \(sg \cdot x + c > 0\) and \(x = c\) for \(sg \in \{1, -1\}\), \(c \in \mathbb{Q}_S\), and \(x \in \{x_1, \ldots, x_d\}\). Moreover, let \(\vec{v} \in \{0, 1, -1, \star\}^d\). The evaluation of \(p\) w.r.t. \(\vec{v}\) (written \(\rho(\vec{v})\)) results from \(\rho(\vec{v}) = \rho[\vec{x}/\vec{v}]\) by simplifying (in)equalities without \(\star\) to true or false, and by simplifying conjunctions and disjunctions with true resp. false. We write \(\vec{v} \models^\star p\) if \(\rho(\vec{v})\downarrow\) false.

So if \(\rho\) is \((x_1 - \frac{2}{3} > 0) \land (x_2 = 0)\) and \(\vec{v} = (\star, 0)\), then \(\rho(\vec{v})\downarrow\) is \(\star - \frac{2}{3} > 0\). Hence, \(\vec{v} \models^\star p\).

Let \(v_1, \ldots, v_k\) be the algebraic multiplicities of the eigenvalues \(\lambda_1, \ldots, \lambda_k\). So in Fig. 1 we have \(\lambda_1 = 1, \lambda_2 = 2, \nu_1 = 2, \nu_2 = 3\). Then we define the blocks \(B_{\lambda_1} = \{1, \nu_1\}, B_{\lambda_2} = \{\nu_1 + 1, \ldots, \nu_1 + \nu_2\}, \ldots, B_{\lambda_k} = \{\nu_1 + 1, \ldots, \nu_1 + \nu_2 + 1, \ldots, d\}\). Note that if \(\lambda_1 = 0\), then all entries \(q_1, \ldots, q_{\nu_1}\) of \(\vec{v}\) are 0. Thus, we assume that 0 is not an eigenvalue of \(A\).

Now we define candidate assignments \(cndAssg(\rho_{f,i})\) for the formulas \(\rho_{f,i}\), which contain all \(\vec{v} \in \{0, 1, -1, \star\}^d\) that may satisfy \(\rho_{f,i}\) (if a suitable value for \(\star\) is found). For however, to 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 \(\lambda_1\) = 1.

**Definition 11 (Sets of Candidate Assignments).** For all \(0 \leq s \leq \ell\), we define \(cndAssg(\rho_{f,s}) = \{\vec{v} \in \{0, 1, -1, \star\}^d \mid \vec{v} \models^\star \rho_{f,s}, v_j = \star \implies j \in B_1, \forall B \in \{B_{\lambda_1}, \ldots, B_{\lambda_k}\}, \text{there is at most one } j \in B \text{ with } v_j \neq 0\}\)

**Example 12.** For \(\rho_{f,4} = (x_4 > 0 \land x_5 = 0)\) in Ex. 8, \(\vec{v} \models^\star \rho_{f,4}\) 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 may be non-zero, we have \(v_5 = 0\). 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 \(\rho_{f,0} = (x_1 = 4) \land \bigwedge_{j=2}^5 (x_j = 0)\):

\[cndAssg(\rho_{f,4}) = \left\{ \begin{array}{c} \{0, 1, 0, 0, 0, 0, 0\} \\
0, 0, 0, 0, 0, 0, 0 \end{array} \right\}, \ cndAssg(\rho_{f,0}) = \left\{ \begin{array}{c} \{0, 0, 0, 0, 0, 0, 0\} \\
0, 0, 0, 0, 0, 0, 0 \end{array} \right\} \]

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

\[|cndAssg(ic(p > 0))| \leq (d + 2) \cdot (3 \cdot \max \{v_\ell \mid 1 \leq i \leq k \} + 1)^k.\]

**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 \(ic(\vec{v})\) by replacing each atom \(f(\vec{q}) \geq 0\) in \(\varphi(\vec{q})\) (where \(f \in \mathcal{S}[\mathcal{F}_{\text{lin}}]\) by \(ic(f(\vec{q}) > 0)\), and then checks each candidate assignment from \(cndAssg(ic(\varphi(\vec{q}))) = \bigcup_{f(\vec{q}) \geq 0} \text{atom in } \varphi(\vec{q}) \text{ cndAssg}(ic(f(\vec{q}) > 0))\).

To this end, it calls a method SMT(\(\varphi, \mathcal{V}, \mathcal{S}\)) which checks whether the linear formula \(\varphi\) 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 \(\varphi(\vec{v})\downarrow\) is satisfiable. Here, we have to assign all occurrences of \(\star\) in the formula \(\varphi(\vec{v})\downarrow\) the same value.

2022 34
\textbf{Input:} a formula $\varphi$ over the atoms $\{f \triangleright 0 \mid f \in S[\vec{x}]_{\text{lin}}, \triangleright \in \{>, \geq\}\}$, a normalized closed form $\vec{q}$, and a ring $S \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$

\textbf{Result:} $\top$ if $\exists \vec{a} \in S^d$. $\text{ic}(\varphi(\vec{q}))$ is valid, $\bot$ otherwise

\textbf{foreach} $\vec{v} \in \text{cnfAssg}(\psi)$ \textbf{do}
\begin{itemize}
  \item $\psi' \leftarrow \psi(\vec{v})$
  \item if $\text{SMT}((\psi' \land \neg \phi) \land (\neg \phi \land (\phi \lor \psi')), S)$ then return $\top$
\end{itemize}
\textbf{return $\bot$}

\textbf{Algorithm 1} Checking Interval Conditions

The formula $\text{ic}(\varphi(\vec{q}))$ and each element of $\text{cnfAssg}(\text{ic}(\varphi(\vec{q})))$ can be computed in polynomial time. By (3), $\text{cnfAssg}(\text{ic}(\varphi(\vec{q})))$ 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 $\varphi$ and $\nu_i \leq d$ for all $1 \leq i \leq k$. Thus, $\text{cnfAssg}(\text{ic}(\varphi(\vec{q})))$ 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.

\textbf{Example 13.} Reconsider the loop in Fig. 1 and $\varphi, p, p'$ in Ex. 5. Here, $\psi = \text{ic}(\varphi(\vec{q})) = \text{ic}(p > 0) \land \text{ic}(p' > 0)$, where $\text{ic}(p > 0) = \bigvee_{i=1}^{\nu} \rho_{f,i}$ as in Ex. 8. Note that $\text{coefs}(p') = \{\alpha^{(1,0)}_1, \alpha^{(1,1)}_2\}$ with $\alpha^{(1,0)}_1 = 2 \cdot x_1 - 5$ and $\alpha^{(1,1)}_2 = 2 \cdot x_2$. 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 $\vec{v} = (\ast, 0, 0, 0, 0)$. Then $(\rho_{f,1} \lor \rho_{f,2})(\vec{v}) = (\ast \land 4 > 0) \land (\ast \land \frac{5}{2} > 0)$ is satisfiable with the model $\ast = 3$. Hence, this model also satisfies $\psi(\vec{v}) \land \neg \phi$. Thus, Alg. 1 proves validity of $\exists \vec{a} \in S^d$. $\text{ic}(\varphi(\vec{q}))$ and therefore, non-termination of the uniform loop over $S$ for both $S = \mathbb{Q}$ and $S = \mathbb{R}$. 

5 Conclusion

We presented an approach to decide termination of uniform loops over $S \in \{\mathbb{Q}, \mathbb{R}\}$ 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 $\langle \varphi', Q \cdot \vec{x} \rangle$ 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 $S \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$, i.e., our decision procedure can indeed be used as an efficient technique for termination analysis.

References

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:

\[
\text{while } (x > 0) \text{ do } (x, y) \leftarrow (x + y, y - 1)
\]  

(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 (\(M\PhiRFs\), 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 \(M\PhiRF\) for a loop implies linear runtime complexity. In this paper, we show how to integrate \(M\PhiRFs\) into our modular approach for complexity analysis of integer programs from [4]. In contrast to [2], we infer \(M\PhiRFs\) 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.
2 Improving Automatic Complexity Analysis of Integer Programs

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

Instead of while loops as in (1), we use a formalism based on transitions. Fig. 1 depicts an integer program with locations \( L = \{ \ell_0, \ell_1, \ell_2 \} \) and variables \( V = \{ x, y, z \} \). \( T \) is the set of transitions \( t = (\ell, \tau, \eta, \ell') \), i.e., directed edges from a location \( \ell \) to \( \ell' \) which are labeled with a guard \( \tau \) and an update function \( \eta : V \rightarrow \mathbb{Z}[V] \). In Fig. 1 we omitted trivial guards, i.e., \( \tau = \text{true} \), 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 \( \ell \) to \( \ell' \) if the guard \( \tau \) is fulfilled, and the current state \( \sigma : V \rightarrow \mathbb{Z} \) is updated according to \( \eta \). We denote such an evaluation step by \( (\ell, \sigma) \rightarrow_t (\ell', \sigma') \), where \( \sigma'(v) = \sigma(\eta(v)) \) for all \( v \in V \). Moreover, \( \rightarrow \) is \( \bigcup_{t \in T} \rightarrow_t \).

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

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 \( SB(t, v) \). The arithmetic expression \( SB(t, v) \) 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 M\( \Phi RF \) 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 M\( \Phi RFs \) 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 M\( \Phi RFs \) for sub-programs \( T'_2 \subseteq T' \subseteq T \) which is crucial for our modular approach (see Thm. 3). Let \( \mathbb{Z}[V]_{\text{lin}} \) denote the set of linear polynomials (i.e., of degree at most 1) over \( \mathbb{Z} \) in the variables \( V \).

▶ Definition 1 (M\( \Phi RFs \) for Sub-Programs). Let \( \emptyset \neq T'_2 \subseteq T' \subseteq T \) and \( d \geq 1 \). A tuple \( f = (f_1, \ldots, f_d) \) of functions \( f_1, \ldots, f_d : L \rightarrow \mathbb{Z}[V]_{\text{lin}} \) is an M\( \Phi RF \) of depth \( d \) for \( T'_2 \) and \( T' \) if for all evaluation steps \( (\ell, \sigma) \rightarrow_t (\ell', \sigma') \):

(a) If \( t \in T'_2 \), 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 \( t \in \mathcal{T}' \setminus \mathcal{T}_2 \), then we have \( \sigma(f_i(\ell)) \geq \sigma'(f_i(\ell')) \) for all \( 1 \leq i \leq d \).

For instance, \( (f_1, f_2) \) is an \( \mathcal{M}\PhiRF \) for \( \mathcal{T}_2 = \{t_2\} \) 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 \( \mathcal{T}_\ell = \{ t \mid t = (\ell', \tau, \eta, \ell) \land t \in \mathcal{T} \setminus \mathcal{T}' \} \) and the set of entry locations is \( \mathcal{E}_{\mathcal{T}'} = \{ \ell_{in} \mid \mathcal{T}_{in} \neq \varnothing \land \exists \ell' : (\ell_{in}, \tau, \eta, \ell') \in \mathcal{T}' \} \). Moreover, the entry transitions of \( \mathcal{T}' \) are \( \mathcal{E}_{\mathcal{T}'} = \bigcup_{t \in \mathcal{E}_{\mathcal{T}'}} \mathcal{T}_t \). So for the program in Fig. 1 and \( \mathcal{T}' = \{t_2, t_3\} \), we have \( \mathcal{T}_\ell = \{t_0\}, \mathcal{T}_\ell = \{t_1\}, \mathcal{E}_{\mathcal{T}'} = \{t_2\}, \) and \( \mathcal{E}_{\mathcal{T}'} = \{t_1\} \).

Now Lemma 2 gives rise to a runtime bound \( \beta_t \). In any run through \( \mathcal{T}' \) starting in \( (\ell, \sigma) \), all functions \( f_i \) of the \( \mathcal{M}\PhiRF \) become negative after at most \( |\sigma|(|\beta_t|) \) many \( \mathcal{T}_2 \)-evaluations. To use the bound \( \beta_t \) 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 \(|\cdot|\)). So for example, \(|-x + 2| = | -1 | \cdot x + 2 | = x + 2 \).

**Lemma 2 (Local Runtime Bound for Sub-Program).** Let \( \varnothing \neq \mathcal{T}_2 \subseteq \mathcal{T}' \subseteq \mathcal{T} \) and let \( f = (f_1, \ldots, f_d) \) be an \( \mathcal{M}\PhiRF \) for \( \mathcal{T}_2 \) and \( \mathcal{T}' \). For all \( 1 \leq i \leq d \) we define the constants \( \gamma_i \in Q \), and for all \( \ell \in \mathcal{E}_{\mathcal{T}'} \), we define the arithmetic expression \( \beta_i : \)

- \( \gamma_1 = 1 \) and \( \gamma_i = 2 + \frac{1}{i-1} + \frac{1}{i-1} \) for \( i > 1 \)
- \( \beta_i = 1 + d! \cdot \gamma_i \cdot (|f_1(\ell)| + \ldots + |f_d(\ell)|) \)

Then for any run \( (\ell, \sigma) \rightarrow_{\mathcal{T}_2}^* (\ell', \sigma') \) with \( n \geq |\sigma|(|\beta_t|) \) and any \( 1 \leq i \leq d \), we have \( \sigma'(f_i(\ell')) < 0 \).

We have \( \gamma_1 = 1 \) and \( \gamma_2 = 2 + \frac{1}{4} + \frac{1}{4} = 4 \). 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 \( \gamma_2 \cdot |\sigma(\beta_t)| = |\sigma(1 + 2!) \cdot \gamma_2 \cdot (|f_1(\ell_2)| + |f_2(\ell_2)|)| = 8 \cdot |\sigma(1 + 2!) \cdot (|f_1(\ell_2)| + |f_2(\ell_2)|)| = 8 \cdot |\sigma(x) + 8 \cdot |\sigma(y)| + 9 \) many times when starting in the configuration \( (\ell_2, \sigma) \).

Note that \( \beta_t \) 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 \( t \in \mathcal{T}_t \) to an entry program \( \ell \in \mathcal{E}_{\mathcal{T}'} \). Thus, to lift \( \beta_t \) to a (possibly non-linear) global bound, we have to instantiate the variables in \( \beta_t \) 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* \( SB(t, v) \) which over-approximate the largest absolute value of \( v \) after the transition \( t \). We also use the notation \( SB(t, b) \) for arithmetic expressions \( b \), where \( SB(t, b) \) results from \( b \) by replacing each variable \( v \) in \( b \) by \( SB(t, v) \). Hence, \( SB(t, \beta_t) \) is a (global) bound on the number of applications of transitions from \( \mathcal{T}_2 \) if \( \mathcal{T}' \) is entered once via the entry transition \( t \). Here, weak monotonic increase of \( \beta_t \) ensures that the over-approximation of the variables \( v \) in \( \beta_t \) by \( SB(t, v) \) indeed leads to an over-approximation of \( \mathcal{T}^*_2 \)'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 \( RB(t) \). This leads to Thm. 3. The analysis starts with a runtime bound \( RB \) and a size bound \( SB \) which map all transitions to \( \omega \), except for the transitions \( t \) outside of strongly connected components (SCCs), where \( RB(t) = 1 \). Afterwards, \( RB \) and \( SB \) are refined repeatedly (see [4] for the computation of size bounds). Instead of using a single ranking function for the refinement of \( RB \) as in [4], Thm. 3 now allows us to replace \( RB \) by a refined bound \( RB' \) based on an \( \mathcal{M}\PhiRF \).

**Theorem 3 (Refining Runtime Bounds Based on \( \mathcal{M}\PhiRFs \)).** Let \( RB \) be a runtime bound, \( SB \) a size bound, \( \varnothing \neq \mathcal{T}'_2 \subseteq \mathcal{T}' \subseteq \mathcal{T} \) such that \( \mathcal{T}' \) does not contain any initial transitions, and \( \beta_t \) be as in Lemma 2. Then \( RB' \) is also a runtime bound, where \( RB'(t) = RB(t) \) for \( t \notin \mathcal{T}'_2 \) and

\[
RB'(t_{>}) = \sum_{t \in \mathcal{E}_{\mathcal{T}'}} \sum_{t' \in \mathcal{T}_{t'}} RB(t') \cdot SB(t, \beta_t) \quad \text{for all } t_{>} \in \mathcal{T}'_2.
\]

**Example 4.** In Fig. 1, \( t_1 \) and \( t_3 \) are evaluated at most \( z \) times (this can be shown by
Improving Automatic Complexity Analysis of Integer Programs

4 Improving Automatic Complexity Analysis of Integer Programs

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

\[\text{Figure 2 Original Loop}\]

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

\[\text{Figure 3 After Control-Flow Refinement}\]

rank the ordering of \(T^\prime = \{t_1, t_2, t_3\}\) and \(T^\prime_2 = \{t_1\}\) resp. \(T^\prime_3 = \{t_3\}\). Hence, \(RB(t_0) = 1\) and \(RB(t_1) = RB(t_3) = z\) is a runtime bound. So by Thm. 3 and \(SB(t_1, v) = z\) for all \(v \in V\), we get \(RB(t_2) = RB(t_1) \cdot SB(t_1, t_2) = z \cdot (8 \cdot SB(t_1, x) + 8 \cdot SB(t_1, y) + 9) = 16 \cdot z^2 + 9 \cdot z\). Thus, the runtime of the full program is bounded by \(\sum_{i=0}^{3} RB(t_i) = 16 \cdot z^2 + 11 \cdot z + 1\).

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 \(P\) into a new program \(P^\prime\) which is “easier” to analyze. Of course, we ensure that the runtime of \(P^\prime\) is at least the runtime of \(P\). Then it is sound to infer upper runtime bounds for \(P^\prime\) instead of \(P\). 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 \(T_{SCC}\) 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 \(\varphi\) stands for all states \(\sigma\) with \(\sigma(\varphi) = \text{true}\). To formalize this, we label every location \(\ell\) in the SCC by a constraint \(\varphi\) which describes (a superset of) those states \(\sigma\) which can occur in this location. So for any location \(\ell\), all reachable configurations have the form \((\ell, \sigma)\) such that \(\sigma(\varphi) = \text{true}\).

We begin with labeling the entry locations of \(T_{SCC}\) by the constraint \text{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
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
 & \(O(1)\) & \(O(n)\) & \(O(n^2)\) & \(O(n^{>2})\) & \(O(\text{EXP})\) & \(< \infty\) & \(\text{AVG}^+ (s)\) & \(\text{AVG} (s)\) \\
\hline
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 & 0 & 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 \\
\hline
\end{tabular}

**Figure 4** Evaluation on the Category Complexity\_C\_Integer from the TPDB

initial values, and “< ∞” 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/](https://aprove-developers.github.io/ComplexityMprfCfr/).

**References**


WST 2022
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
Improving Automatic Complexity Analysis of Integer Programs


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


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.


3 Non-Sparse Tiling

Since sparse tiling is not certified currently, Matchbox applies “full tiling”: an $R$ over $\Sigma$ is transformed to an equivalent $R'$ over the set of $k$-tiles $\Sigma^k$. 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.

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%20%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