19th International Workshop on Termination

WST 2023, August 24–25, 2023
held as part of Obergurgl Summer on Rewriting 2023

https://termination-portal.org/wiki/WST2023

Edited by

Akihisa Yamada

Preface

This report contains the proceedings of the 19th International Workshop on Termination (WST 2023), which was held in Obergurgl during August 24–25 as part of Obergurgl Summer on Rewriting (OSR 2023).

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 19th 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), Oxford (2018), the virtual space (2021), and Haifa (2022).

The WST 2023 program included an invited talk by Benjamin Lucien Kaminski on *Termination of Probabilistic Programs*. WST 2023 received 13 regular submissions and six abstracts for tool presentations, two of which were accompanied by a system description. After light reviewing the program committee decided to accept all submissions. The proceedings is published on arXiv. Each of the 13 regular submissions is made available as an arXiv article, and the two system descriptions are combined into this preface, due to the arXiv policy not to accept very short papers.

I would like to thank the program committee members for their dedication and effort, and the organizers of OSR 2023 for the invaluable help in the organization.

Innsbruck, August 2023 Akihisa Yamada

Organization

Program Committee

Martin Avanzini	INRIA Sophia Antipolis
Florian Frohn	RWTH Aachen
Carsten Fuhs	Birkbeck, U. London
Raúl Gutiérrez	U. Politécnica de Madrid
Étienne Payet	U. La Réunion
Albert Rubio	Complutense U. Madrid
René Thiemann	U. Innsbruck
Deivid Vale	Radboud U. Nijmegen
Johannes Waldmann	HTWK Leipzig
Akihisa Yamada (chair)	AIST Tokyo Waterfront

Contents

Tool Descriptions

Invited Talk

Termination of Probabilistic Programs

Benjamin Lucien Kaminski
Saarland University
University College London

Abstract

Unlike for ordinary programs, termination of probabilistic programs is more nuanced: A probabilistic program can terminate with probability 1 while still requiring an expected infinite number of computation steps until termination. We will explore the complexity landscape of probabilistic program termination and present proof rules for proving both almost-sure termination (i.e. termination with probability 1) as well as positive almost-sure termination (i.e. termination within finite expected time). Time permitting, we will furthermore dive into open problems on termination of weighted programs – a generalization of probabilistic programs where branches can be associated with more general weights from a semiring.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Probabilistic Programming, Termination

On Singleton Self-Loop Removal for Termination of LCTRSs with Bit-Vector Arithmetic

Ayuka Matsumi
Graduate School of Informatics, Nagoya University, Japan

Naoki Nishida
Graduate School of Informatics, Nagoya University, Japan

Misaki Kojima
Graduate School of Informatics, Nagoya University, Japan

Donghoon Shin
Graduate School of Informatics, Nagoya University, Japan

Abstract

As for term rewrite systems, the dependency pair (DP, for short) framework with several kinds of DP processors is useful for proving termination of logically constrained term rewrite systems (LCTRSs, for short). However, the polynomial interpretation processor is not so effective against LCTRSs with bit-vector arithmetic (BV-LCTRSs, for short). In this paper, we propose a novel DP processor for BV-LCTRSs to solve a singleton DP problem consisting of a dependency pair forming a self-loop. The processor is based on an acyclic directed graph such that the nodes are bit-vectors and any dependency chain of the problem is projected to a path of the graph. We show a sufficient condition for the existence of such an acyclic graph, and simplify it for a specific case.

2012 ACM Subject Classification Theory of computation → Rewrite systems

Keywords and phrases constrained rewriting, dependency pair framework, imperative program

Funding This work was partially supported by JSPS KAKENHI Grant Number 18K11160.

1 Introduction

Logically constrained term rewrite systems (LCTRSs, for short) [12] are expected to be useful computational models for verifying not only functional but also imperative programs [7]. Especially, *LCTRSs with bit-vector arithmetic* (BV-LCTRSs, for short) are useful for programs written in C or other languages with *fixed-width integers* such as `int` of C because primitive data types, structures, and unions are represented by bit-vectors in a natural and precise manner [10]. In proving validity of an equation w.r.t. a given rewrite system by means of *rewriting induction* [15, 7], we need to *frequently* try to prove termination of rewrite systems obtained by adding rewriting rules for induction hypotheses into the given system. Therefore, for verification tools based on rewriting induction, the performance of proving termination of rewriting systems has a great influence on the proof power and execution time.

The *dependency pair framework* (DP framework, for short) [8] equipped with *DP processors* which decompose DP problems is a well investigated technique for proving termination of rewrite systems, and has been extended to many kinds of constrained rewrite systems including LCTRSs [3, 6, 11, 16]. Some fundamental DP processors are applicable to almost all kinds of rewrite systems without any change. For example, the *dependency graph processor* based on SCC decomposition is applicable to LCTRSs. On the other hand, the *polynomial interpretation processor*, one of the most powerful processors in proving termination of LCTRSs with integer arithmetic, is applicable to a DP problem of BV-LCTRSs but not so effective against it: It is ineffective if it contains a dependency pair with a usable rule for an operator of BVs such as addition, which may cause overflow and/or underflow. To enhance

Listing 1 A C program defining a function to count x times

```
int cnt(int x){
    int z=0;
    for(int i=0; i<x; i++) z++;
    return z;
}
```

the power of proving termination of BV-LCTRSs, we need to develop DP processors specific to BV-LCTRSs.

In this paper, we propose a novel DP processor, called a *singleton self-loop removal processor* (SSR processor, for short), aiming at developing a method to prove termination of BV-LCTRSs. Here, a dependency pair is said to *form a self-loop* if it forms a dependency chain of length two or more, and a DP problem is called a *singleton self-loop* problem if it is a singleton set, the pair in which forms a self-loop. The processor takes a singleton self-loop DP problem as an input and is based on an acyclic directed graph such that the nodes are bit-vectors and any dependency chain of the problem is projected to a path of the graph. We show a sufficient condition for the existence of such an acyclic graph, and simplify it for a specific case. Note that the processor returns the empty set—the solved DP problem—if the sufficient condition is satisfied by a given singleton self-loop DP problem.

In the rest of the paper, familiarity with basic notions and notations on term rewriting [1, 14] is assumed. We follow the definition of LCTRSs in [12, 7]. For brevity, we use the 4-bits bit-vectors for type `int`. We denote the set of bit-vectors of length n by $\mathbb{BV}_n$. To distinguish bit-vectors from decimal numbers, we follow the SMT-LIB notation for bit-vectors: A bit-vector $c \in \mathbb{BV}_n$, which is written as a binary numeral in sans-serif font, is denoted by $\mathsf{\#b}c$. We often use regular expressions for binary numerals, e.g., $\mathsf{0}^3$ stands for $\mathsf{000}$.

2 From C Programs to BV-LCTRSs

A set $\mathcal{S}$ of sorts for bit-vectors includes sort bv_n for the n-bits bit-vectors ($n \geq 1$): $\mathcal{S} \supseteq \{bool\} \cup \{bv_n \mid n \geq 1\}$. The set $\mathcal{V}al$ of values is $\{\mathsf{true}, \mathsf{false} : bool\} \cup \bigcup_{n\geq 1}\{b : bv_n \mid b \in \mathbb{BV}_n\}$. The set Σ_{theory} of theory symbols for bit-vectors is an extension of the *core theory* Σ^{core}_{theory} [7] for logical connectives $(\vee, \wedge, \neg)$: $\Sigma_{theory} = \Sigma^{core}_{theory} \cup \mathcal{V}al \cup \{+_{bv_n} : bv_n \times bv_n \Rightarrow bv_n,\ =_{bv_n}, <_{bv_n,S}, <_{bv_n,U}, \geq_{bv_n,S}, \geq_{bv_n,U} : bv_n \times bv_n \Rightarrow bool,\ \ldots \mid n \geq 1\}$. We drop the subscript bv_n from $+_{bv_n}$, $<_{bv_n,S}$, and so on if it is clear from the context. The interpretation of theory symbols for bit-vectors follow the usual semantics of bit-vector arithmetic [13].

▶ **Example 1.** The C program in Listing 1 is transformed into the following BV-LCTRS [10]:

$$\mathcal{R}_1 = \left\{ \begin{array}{ll} \mathsf{cnt}(x) \to \mathsf{u}_1(x, \mathsf{\#b0000}, \mathsf{\#b0000}) & \\ \mathsf{u}_1(x,i,z) \to \mathsf{u}_1(x, i+\mathsf{\#b0001}, z+\mathsf{\#b0001}) & [i <_S x] \\ \mathsf{u}_1(x,i,z) \to z & [i \geq_S x] \end{array} \right\}$$

where $\mathsf{cnt} : bv_4 \Rightarrow bv_4$ and $\mathsf{u}_1 : bv_4 \times bv_4 \times bv_4 \Rightarrow bv_4$. Note that the above LCTRS is a simplified one by means of *chaining* (cf. [4, Section 7]). Note also that *calculation rules* [7] such as $x + y \to z$ $[z = x + y]$ are implicitly included in $\mathcal{R}_1$. For example, we have that $\mathsf{cnt}(\mathsf{\#b0010}) \to_{\mathcal{R}_1} \mathsf{u}_1(\mathsf{\#b0010}, \mathsf{\#b0000}, \mathsf{\#b0000}) \to_{\mathcal{R}_1} \mathsf{u}_1(\mathsf{\#b0010}, \mathsf{\#b0000} + \mathsf{\#b0001}, \mathsf{\#b0000} + \mathsf{\#b0001}) \to_{\mathcal{R}_1} \mathsf{u}_1(\mathsf{\#b0010}, \mathsf{\#b0001}, \mathsf{\#b0000} + \mathsf{\#b0001}) \to_{\mathcal{R}_1} \cdots \to_{\mathcal{R}_1} \mathsf{\#b0010}$.

3 The DP Framework for LCTRSs

The DP framework [8] for TRSs has been extended for LCTRSs [11].

Let $\mathcal{R}$ be an LCTRS. The marked symbol of a defined symbol $f : \iota_1 \times \cdots \times \iota_n \Rightarrow \iota \in \mathcal{D}_\mathcal{R}$ is denoted by $f^\#$ and the set of marked symbols for $\mathcal{D}_\mathcal{R}$ is denoted by $\mathcal{D}^\#_\mathcal{R}$. We introduce a fresh basic sort *dpsort*, and $f^\#$ has sort $\iota_1 \times \cdots \times \iota_n \Rightarrow \mathit{dpsort}$. If $t = f(t_1, \ldots, t_n)$ with $f \in \mathcal{D}_\mathcal{R}$, then $f^\#(t_1, \ldots, t_n)$ is denoted by $t^\#$. For each rule $\ell \to r\ [\phi] \in \mathcal{R}$, a constrained rewrite rule $\ell^\# \to t^\#\ [\phi]$ is called a *dependency pair* (DP, for short) of $\mathcal{R}$ if t is a subterm of r and $\mathit{root}(t) \in \mathcal{D}_\mathcal{R}$. The set of DPs of $\mathcal{R}$ is denoted by $DP(\mathcal{R})$. In the following, we use $\mathcal{P}$ as a set of DPs of $\mathcal{R}$, i.e., $\mathcal{P} \subseteq DP(\mathcal{R})$. A sequence $\rho_1, \rho_2, \ldots$ of DPs in $\mathcal{P}$ is called a *dependency chain* of $\mathcal{P}$ ($\mathcal{P}$-chain, for short) if there are substitutions $\gamma_1, \gamma_2, \ldots$ such that for each $i > 0$, γ_i *respects* $\rho_i = (s_i^\# \to t_i^\#\ [\phi_i])$—$\mathcal{R}an(\gamma_i|_{\mathcal{V}ar(\phi_i) \cup (\mathcal{V}ar(t_i) \setminus \mathcal{V}ar(s_i))}) \subseteq \mathcal{V}al$ and $[\![\phi_i\gamma_i]\!] = \top$—and $t_i^\#\gamma_i \to^*_\mathcal{R} s_{i+1}^\#\gamma_{i+1}$. A *DP problem* $(\mathcal{P}, \mathcal{R})$, abbreviated to $\mathcal{P}$, is called *chain-free* if there is no infinite $\mathcal{P}$-chain.

▶ **Theorem 2** ([11]). *An LCTRS $\mathcal{R}$ is terminating iff the DP problem $DP(\mathcal{R})$ is chain-free.*

A DP processor *Proc* is a function that maps a DP problem to a finite set of DP problems: $\mathit{Proc}(\mathcal{P}) \subseteq 2^\mathcal{P}$. We say that *Proc* is *sound* if for any DP problem $\mathcal{P}$, $\mathcal{P}$ is chain-free, whenever all DP problems in $\mathit{Proc}(\mathcal{P})$ are chain-free. If the initial problem $DP(\mathcal{R})$ is decomposed into the solved DP problem $\emptyset$ by applying sound DP processors, then the framework succeeds in proving termination of $\mathcal{R}$.

A *dependency graph* (DG, for short) of $\mathcal{P}$ is a directed graph $\mathcal{G} = (\mathcal{P}, \mathcal{E})$, denoted by $DG(\mathcal{P})$, such that $\mathcal{E} = \{(\rho_1, \rho_2) \mid \rho_1, \rho_2 \in \mathcal{P},$ the seqeunce ρ_1, ρ_2 is a $\mathcal{P}$-chain$\}$. Moreover, a directed graph $\mathcal{G}' = (\mathcal{P}, \mathcal{E}')$ with $\mathcal{E}' \supseteq \mathcal{E}$ is called a *DG approximation* of $\mathcal{P}$. A computation of DG approximations can be seen in [11].

▶ **Theorem 3** (cf. [11]). *The* dependency graph processor Proc_{DG} *such that* $\mathit{Proc}_{DG}(\mathcal{P}) = \{\mathcal{P}' \mid \mathcal{P}'$ *are the nodes of an SCC in a DG approximation of* $\mathcal{P}\}$ *is a sound DP processor.*

▶ **Example 4.** Consider $\mathcal{R}_1$ in Example 1 again. The following pairs are the DPs of $\mathcal{R}_1$:

$$DP(\mathcal{R}_1) = \left\{ \begin{array}{ll} (1) & \mathsf{cnt}^\#(x) \to \mathsf{u}_1^\#(x, \mathtt{\#b0000}, \mathtt{\#b0000}) \\ (2) & \mathsf{u}_1^\#(x, i, z) \to \mathsf{u}_1^\#(x, i + \mathtt{\#b0001}, z + \mathtt{\#b0001})\ [i <_S x] \end{array} \right\}$$

Since $DG(DP(\mathcal{R}_1)) = (DP(\mathcal{R}_1), \{((1),(2)),((2),(2))\})$, we have that $\mathit{Proc}_{DG}(DP(\mathcal{R}_1)) = \{\ \{\ (2)\ \}\ \}$. In the following, we denote $\{\ (2)\ \}$ by $\mathcal{P}_1$:

$$\mathcal{P}_1 = \{\ (2)\ \mathsf{u}_1^\#(x, i, z) \to \mathsf{u}_1^\#(x, i + \mathtt{\#b0001}, z + \mathtt{\#b0001})\ [i <_S x]\ \}$$

4 DP Processor for Singleton Self-Loop Removal

The polynomial interpretation (PI, for short) processor over the integers (cf. [11, Theorem 10]) is one of the most powerful DP processors in proving termination of LCTRSs with integer arithmetic. As indicated in Section 1, however, the PI processor is ineffective against DP problems of BV-LCTRSs in the case where a usable calculation rule may cause overflow and/or underflow, because such a rule (e.g., $x + y \to z\ [\, z = x + y\,]$) cannot be ordered by any meaningful PI order. In this section, we propose a DP processor, called an *singleton self-loop removal processor*, that solves singleton self-loop DP problems under a certain condition.

A *singleton self-loop DP problem* in this paper is assumed to be a singleton set of the form $\{\ \mathsf{f}^\#(x_1, \ldots, x_n) \to \mathsf{f}^\#(t_1, \ldots, t_n)\ [\phi]\ \}$ that forms a chain of length two or more. For

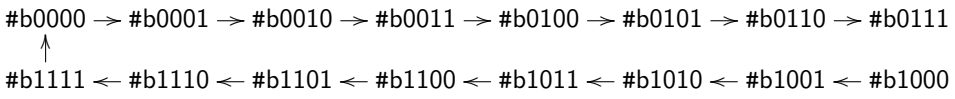


Figure 1 The acyclic graph obtained from $\mathsf{u}_1^{\#}(t_1, b, t_3)$ by projecting to the second argument.

example, $\mathcal{P}_1$ in Example 4 is a singleton self-loop DP problem. In the rest of this section, we let $\mathcal{P} = \{\ f^{\#}(x_1, \ldots, x_n) \to f^{\#}(t_1, \ldots, t_n)\ [\phi]\ \}$ be a singleton self-loop DP problem, where f has sort $\iota_1 \times \cdots \times \iota_{i-1} \times bv_l \times \iota_{i+1} \times \cdots \times \iota_n \Rightarrow \iota$, l is a natural number, $x_1, \ldots, x_n$ are pairwise distinct variables, and ϕ is satisfiable.

In rewriting a term by the DP (2) of $\mathcal{P}_1$, the first and third arguments of $\mathsf{u}_1^{\#}$ do not affect the constraint $i <_S x$, i.e., they preserve the evaluation of the constraint, while the second argument of $\mathsf{u}_1^{\#}$ does not preserve the value of $i <_S x$—for a substitution θ such that $(i <_S x)\theta$ holds, $(i <_S x)\{i \mapsto i+1\}\theta$ $(= (i+1 <_S x)\theta)$ may not hold. We formulate this notion as follows: A rewrite rule $f^{\#}(x_1, \ldots, x_n) \to f^{\#}(t_1, \ldots, t_n)\ [\phi]$ is said to *preserve its constraint w.r.t.* $\bar{I}$ $(\subseteq \{1, \ldots, n\})$ if all of the following hold:

(1) $\{x_i \mid 1 \le i \le n,\ i \notin \bar{I}\} \subseteq \mathcal{V}ar(\phi)$,

(2) for each $j \in \{1, \ldots, n\}$, if $x_j \in \mathcal{V}ar(\phi)$, then $t_j \in \mathcal{T}(\Sigma_{theory}, \mathcal{V})$, and

(3) $(\exists y_1, \ldots, y_m.\ \phi) \Leftrightarrow (\exists y_1, \ldots, y_m.\ \phi)\theta$ is valid, where $\{y_1, \ldots, y_m\} = \mathcal{V}ar(\phi) \setminus \{x_1, \ldots, x_n\}$ and $\theta = \{x_j \mapsto t_j \mid 1 \le j \le n,\ j \in \bar{I}\}$.

For example, the DP (2) in $\mathcal{P}_1$ preserves its constraint w.r.t. $\{1, 3\}$.

To prove chain-freeness of $\mathcal{P}_1$, we use the fact that, given a DP problem $\mathcal{P}$, if there exists an acyclic directed graph such that the nodes are bit-vectors and any $\mathcal{P}$-chain is projected to a path of the graph, then $\mathcal{P}$ is solved, i.e., $\mathcal{P}$ is chain-free.

Let us consider the DP (2) in $\mathcal{P}_1$. Let π_1 be a projection of terms rooted by $\mathsf{u}_1^{\#}$ to the second argument of the root symbol, i.e., $\pi_1(\mathsf{u}_1^{\#}(t_1, t_2, t_3)) = t_2$. Then, we construct a directed graph such that the nodes are the 4-bits bit-vectors and the edges illustrated in Figure 1 are obtained from $\mathcal{P}_1$ by applying π_1 to ground instances of the DP (2) in $\mathcal{P}_1$. Since the graph is acyclic and any $\mathcal{P}_1$-chain is projected to a path of the graph, the graph ensures the non-existence of infinite $\mathcal{P}_1$-chains.

In the following, we show a sufficient condition for the existence of such an acyclic graph.

▶ **Theorem 5.** *Let $i \in \{1, \ldots, n\}$. Suppose that*

(4) *$f^{\#}(x_1, \ldots, x_n) \to f^{\#}(t_1, \ldots, t_n)\ [\phi]$ preserves its constraint w.r.t. $\{1, \ldots, i-1, i+1, \ldots, n\}$,*

(5) *there exists $a \in \{0, \ldots, l\}$ such that $t_i - x_i =_{bv_l} \#\mathsf{b}c\mathsf{10}^a$ is valid for some $c \in \{0,1\}^{l-a-1}$, and*

(6) *there exists some terms $u, v : bv_l \in \mathcal{T}(\Sigma_{theory}, \mathcal{V}ar(\phi))$ such that*

- **a.** *$\phi \Rightarrow (u =_{bv_l} u\theta \wedge v =_{bv_l} v\theta \wedge v - u \ge_U \#\mathsf{b}\mathsf{0}^{l-a-1}\mathsf{10}^a)$ is valid, where $\theta = \{x_j \mapsto t_j \mid 1 \le j \le n\}$, and*
- **b.** *$(\forall x_i.\ (\phi \Rightarrow ((x_i <_U u \vee v \le_U x_i) \wedge u <_U v))) \vee (\forall x_i.\ (\phi \Rightarrow (v \le_U x_i \wedge x_i <_U u)))$ is valid.*

Then, $\mathcal{P}$ is chain-free.

The assumptions in Theorem 5 mean the following, respectively: **(4)** The evaluation of ϕ is only affected by the i-th argument of f in applying the DP to terms; **(5)** the i-th argument of f plays a role of a *loop variable*, and $t_i - x_i$ is a fixed amount $(\#\mathsf{b}c\mathsf{10}^a)$ of the increment or decrement of the i-th argument at the application of the DP; **(6)** the terms u, v imply a fixed interval $[u, v)$ that is not affected by the application of the DP and has the length more than

$\texttt{\#b0}^{l-a-1}\texttt{10}^a$. Note that if $v <_U u$ holds, then the interval is $\{b \in \mathbb{BV}_l \mid b <_U v \vee u \leq_U b\}$. In applying the DP to a term (i.e., ϕ is satisfied), the value of x_i is out of the interval. In other words, if the value of x_i is in the interval, then ϕ is not satisfied and thus, the DP is not applicable. By repeating the application of the DP, the value of x_i always enter the interval. This means that a $\mathcal{P}$-chain can no longer be extended and thus, there is no infinite $\mathcal{P}$-chain.

▶ **Example 6.** Let us consider the DP (2) in $\mathcal{P}_1$ again. Regarding the second argument, the DP (2) satisfies both **(4)** and **(5)**: $(i + \texttt{\#b0001}) - i = \texttt{\#b0001}$. Let $u = x + \texttt{\#b1000}$ and $v = x + \texttt{\#b1001}$. Then, u, v satisfy both **(6) a** and **(6) b**: $i <_S x \Rightarrow (x + \texttt{\#b1000} =_{bv_4} x + \texttt{\#b1000} \wedge x + \texttt{\#b1001} =_{bv_4} x + \texttt{\#b1001} \wedge (x + \texttt{\#b1001}) - x =_{bv_4} \texttt{\#b0001}$ and $(\forall i.\ (i <_S x \Rightarrow ((i <_U x + \texttt{\#b1000} \vee x + \texttt{\#b1001} \leq_U i) \wedge x + \texttt{\#b1000} <_U x + \texttt{\#b1001}))) \vee (\forall i.\ (i <_S x \Rightarrow (x + \texttt{\#b1001} \leq_U i \wedge i <_U x + \texttt{\#b1000})))$ are valid. Therefore, by Theorem 5, $\mathcal{P}_1$ is chain-free, i.e., $\mathcal{R}_1$ is terminating.

In the assumption **(6) b** of Theorem 5, theory terms with sort bv_l are interpreted as unsigned integers, but this does not mean that Theorem 5 only works for BV-LCTRSs obtained from C programs where all `int` variables are unsigned. For example, the variable `x`, `i`, and `z` in Listing 1 are signed ones.

To prove chain-freeness of $\mathcal{P}$ by Theorem 5, we need to find terms u, v that satisfy **(6) a** and **(6) b**. It is not easy to find such terms u, v mechanically, because each of them may be either a variable such as x or a bit-vector expression consisting of variables and constants such as $x + \texttt{\#b0111}$. To overcome this difficulty, we propose another implementable criterion that can be applied in a specific case.

Let us focus on the case where $a = 0$ (i.e., increment or decrement of the i-th argument is an odd number) in Theorem 5. Since $\texttt{\#b0}^{l-a-1}\texttt{10}^a = \texttt{\#b0}^{l-1}\texttt{1}$, the formula $v - u \geq_U \texttt{\#b0}^{l-a-1}\texttt{10}^a$ is equivalent to $u \neq_{bv_l} v$. If **(6) b** holds, then $\phi \Rightarrow (u \neq_{bv_l} v)$ is valid. In addition, neither $\forall x_i.\ ((x_i <_U u \vee v \leq_U x_i) \wedge u <_U v)$ nor $\forall x_i.\ (v \leq_U x_i \wedge x_i <_U u)$ is satisfiable. Thus, **(6) b** is equivalent to unsatisfiability of $\forall x_i.\ \phi$ which does not contain either u or v and enables us to drop **(6) a**. In summary, Theorem 5 is simplified in the case where $a = 0$.

▶ **Theorem 7.** *Let $i \in \{1, \ldots, n\}$. Suppose that*

(4) *$f^\#(x_1, \ldots, x_n) \to f^\#(t_1, \ldots, t_n)\ [\phi]$ preserves its constraint w.r.t. $\{1, \ldots, i-1, i+1, \ldots, n\}$,*

(5') *$t_i - x_i =_{bl_l} \texttt{\#b}c\texttt{1}$ is valid for some $c \in \{0, 1\}^{l-1}$, and*

(6') *$\forall x_i.\ \phi$ is unsatisfiable.*

Then, $\mathcal{P}$ is chain-free.

Theorem 7 can be applied only in the case when $a = 0$ in Theorem 5, but in most practical programs, loop variables are incremented by one. Thus, Theorem 7 must be sufficient to prove termination of BV-LCTRSs obtained from such programs. Besides, the criterion in Theorem 7 is more implementable than that in Theorem 5.

▶ **Example 8.** Since $\forall i.\ i <_S x$ is unsatisfiable, by Theorem 7, $\mathcal{P}_1$ is chain-free.

Finally, we propose a DP processor based on Theorems 5 and 7.

▶ **Definition 9.** *Suppose that $\mathcal{P}$ satisfies the assumptions in Theorems 5 or 7. Then, given $\mathcal{P}$, the* singleton self-looping removal processor *$Proc_{SSR}$ returns $\{\emptyset\}$: $Proc_{SSR}(\mathcal{P}) = \{\emptyset\}$.*

By Theorems 5 and 7, it is clear that $Proc_{SSR}$ is a sound DP processor.

5 Future Work

The applicability of the SSR processor $Proc_{SSR}$ is very limited because it works for singleton (self-loop) problems only. We may make a single loop formed by two or more DPs a self-loop formed by a DP by means of chaining. On the other hand, we need a device for multiple loops. An idea for such loops is to extract an innermost loop, decomposing a multiple loop into an innermost one and the others; for the latter, we overapproximate the innermost loop by replacing the innermost loop-variable and accumulators by fresh variables. Our future work is to formulate and implement this idea.

In [5, 9], bit-vectors and their operators are represented over integer arithmetic, e.g., by case analysis for the finite interval of integers ($[-2^{31}, 2^{31} - 1]$) or *modulo* relations. In [2], bit-precise termination is synthesised over lexicographic linear ranking function templates. We have to compare our method with such approaches from the theoretical and empirical points of view.

References

1 F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
2 H.-Y. Chen, C. David, D. Kroening, P. Schrammel, and B. Wachter. Bit-precise procedure-modular termination analysis. *ACM Transactions on Programming Languages and Systems*, 40(1):1:1–1:38, 2018.
3 S. Falke and D. Kapur. Dependency pairs for rewriting with built-in numbers and semantic data structures. In *Proc. RTA 2008*, volume 5117 of *LNCS*, pages 94–109. Springer, 2008.
4 S. Falke and D. Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In *Proc. CADE 2009*, volume 5663 of *LNCS*, pages 277–293. Springer, 2009.
5 S. Falke, D. Kapur, and C. Sinz. Termination analysis of imperative programs using bitvector arithmetic. In *Proc. VSTTE 2012*, volume 7152 of *LNCS*, pages 261–277. Springer, 2012.
6 C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving termination of integer term rewriting. In *Proc. RTA 2009*, volume 5595 of *LNCS*, pages 32–47. Springer, 2009.
7 C. Fuhs, C. Kop, and N. Nishida. Verifying procedural programs via constrained rewriting induction. *ACM Trans. Comput. Log.*, 18(2):14:1–14:50, 2017.
8 J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. LPAR 2004*, volume 3452 of *LNCS*, pages 301–331. Springer, 2005.
9 J. Hensel, J. Giesl, F. Frohn, and T. Ströder. Termination and complexity analysis for programs with bitvector arithmetic by symbolic execution. *J. Log. Algebraic Methods Program.*, 97:105–130, 2018.
10 Y. Kanazawa, N. Nishida, and M. Sakai. On representation of structures and unions in logically constrained rewriting. IEICE Technical Report SS2018-38, IEICE, 2019. Vol. 118, No. 385, pp. 67–72, in Japanese.
11 C. Kop. Termination of LCTRSs. In *Proc. WST 2013*, pages 1–5, 2013.
12 C. Kop and N. Nishida. Term rewriting with logical constraints. In *Proc. FroCoS 2013*, volume 8152 of *LNCS*, pages 343–358, 2013.
13 D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, second edition, 2016.
14 E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002.
15 U. S. Reddy. Term rewriting induction. In *Proc. CADE 1990*, volume 449 of *LNCS*, pages 162–177. Springer, 1990.
16 T. Sasano, N. Nishida, M. Sakai, and T. Ueyama. Transforming Dependency Chains of Constrained TRSs into Bounded Monotone Sequences of Integers. In *Proc. WPTE 2017*, volume 265 of *EPTCS*, pages 82–97. Open Publishing Association, 2018.

Generalizing Weighted Path Orders

Teppei Saito
JAIST, Japan

Nao Hirokawa
JAIST, Japan

Abstract

We show that weighted path orders are special instances of a variant of semantic path orders. Exploiting this fact, we introduce a generalization of weighted path orders that goes beyond the realm of simple termination. Experimental data show that generalized weighted path orders are viable.

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

Keywords and phrases Term Rewriting, Termination, Weighted Path Orders, Semantic Path Orders

Funding *Teppei Saito*: JST SPRING, Grant Number JPMJSP2102
Nao Hirokawa: JSPS KAKENHI Grant Number JP22K11900

Acknowledgements We are grateful to Vincent van Oostrom for his valuable questions and comments on our preliminary work. We also thank Alfons Geser for his support on literature. The comments of the anonymous reviewers helped to improve the presentation of the paper.

1 Introduction

Various classes of reduction orders have been proposed since reduction orders are a fundamental tool in termination analysis of term rewrite systems and in completion-based theorem proving. Concurrently, several attempts have been made to unify these classes of reduction orders. Among others, *weighted path orders* (WPOs) [7] are known as a powerful generalization of Knuth–Bendix orders (KBOs). Extending notion of weight function to *simple monotone algebra*, WPOs unify KBOs and lexicographic path orders (LPOs).

Another approach toward unification is to generalize parameters of so-called path orders so as to take orders on terms, such as semantic path orders (SPOs) [4], *monotonic semantic path orders* (MSPOs) [2, 1], and general path orders (GPOs). However, relationship between WPOs and these approaches has not been thoroughly investigated, except for the one between extended KBOs and SPOs [3]. In particular, the relationship between WPOs and MSPOs has remained open [6].

In this note, we demonstrate an effective construction of an MSPO from the algebra and the precedence of a given WPO. This simulation result leads to a generalization of WPOs that does not impose simplicity on their underlying algebras. The fact that the generalization can show termination of term rewrite systems that are not simply terminating forms a sharp contrast from the original ones. Besides, WPOs have been implemented in several tools for termination analysis and automated theorem proving. Upgrading WPOs to GWPOs can be done with little implementation effort, so we anticipate that these tools may benefit from power of GWPOs.

The rest is organized as follows: In Section 2 we recall weighted path orders. In Section 3 it is shown that weighted path orders can be simulated by a variant of semantic path orders. Section 4 is devoted to a generalization of weighted path orders and discussion on its experimental data. Section 5 concludes the note with future work.

This note is a short version of our recent conference paper [5].

2 Weighted Path Orders

First we recall notions related to weighted path orders, see also the original work [7]. Terms are constructed from the finite signature $\mathcal{F}$ and variables $\mathcal{V}$, and the set of terms is denoted by $\mathcal{T}(\mathcal{F},\mathcal{V})$. We may write $f^{(n)}$ for a function symbol f in order to indicate that f has the arity n. WPOs comprise two ingredients. One is a *precedence*, which is a quasi-order on the signature. The other is an *ordered algebra* $\mathcal{A} = (A, \{f_\mathcal{A}\}_{f\in\mathcal{F}}, >)$. Here $>$ is a strict order on a set A, and $f_\mathcal{A}$ is an n-ary function on A associated with each $f^{(n)} \in \mathcal{F}$. We write $s >_\mathcal{A} t$ if $[\alpha]_\mathcal{A}(s) > [\alpha]_\mathcal{A}(t)$ for all assignments α. Similarly, we write $s \geqslant_\mathcal{A} t$ if $[\alpha]_\mathcal{A}(s) \geqslant [\alpha]_\mathcal{A}(t)$ for all assignments α where $\geqslant$ is the reflexive closure of $>$. The ordered algebra $\mathcal{A}$ is

- *simple* if $f_\mathcal{A}(a_1,\dots,a_i,\dots,a_n) \geqslant a_i$ for all $f^{(n)} \in \mathcal{F}$, $1 \leqslant i \leqslant n$, and $a_1,\dots,a_n \in A$;
- *weakly monotone* if $f_\mathcal{A}(a_1,\dots,a_i,\dots,a_n) \geqslant f_\mathcal{A}(a_1,\dots,b,\dots,a_n)$ for all $f^{(n)} \in \mathcal{F}$, argument positions $1 \leqslant i \leqslant n$, and $a_1,\dots,a_n,b \in A$ with $a_i > b$;
- *simple monotone* if it is simple and weakly monotone; and
- *well-founded* if $>$ is well-founded.

▶ **Definition 1** ([7]). *Let $\mathcal{A}$ be an ordered $\mathcal{F}$-algebra and $\succsim$ a precedence. The* weighted path order $>_{\mathsf{wpo}}$ *is defined on terms as follows: $s >_{\mathsf{wpo}} t$ if*

1. *$s >_\mathcal{A} t$, or*
2. *$s \geqslant_\mathcal{A} t$, $s = f(s_1,\dots,s_m)$, and one of the following conditions holds.*
   a. *$s_i \geqslant_{\mathsf{wpo}} t$ for some $1 \leqslant i \leqslant m$.*
   b. *$t = g(t_1,\dots,t_n)$ and $s >_{\mathsf{wpo}} t_j$ for all $1 \leqslant j \leqslant n$, and moreover*
      (i) *$f \succ g$, or*
      (ii) *$f \succsim g$ and $(s_1,\dots,s_m) >^{\mathsf{lex}}_{\mathsf{wpo}} (t_1,\dots,t_n)$.*

Here $\geqslant_{\mathsf{wpo}}$ denotes the reflexive closure of $>_{\mathsf{wpo}}$.

▶ **Theorem 2** ([7]). *For every simple monotone well-founded algebra and precedence the induced relation $>_{\mathsf{wpo}}$ is a reduction order.*

A TRS $\mathcal{R}$ is terminating if and only if there exists a reduction order $>$ with $\mathcal{R} \subseteq >$. Therefore, WPOs can be used for showing termination of TRSs.

▶ **Example 3.** Consider the following TRS $\mathcal{R} = \{\mathsf{f}(\mathsf{g}(x)) \to \mathsf{g}(\mathsf{f}(\mathsf{f}(x))), \mathsf{f}(\mathsf{h}(x)) \to \mathsf{h}(\mathsf{h}(\mathsf{f}(x)))\}$ taken from [7, Example 9]. Let $\mathcal{A}$ be the simple monotone algebra on $\mathbb{N}$ with $\mathsf{f}_\mathcal{A}(x) = \mathsf{h}_\mathcal{A}(x) = x$ and $\mathsf{g}_\mathcal{A}(x) = x + 1$. Take a precedence $\succsim$ with $\mathsf{f} \succ \mathsf{g} \succ \mathsf{h}$. The relation $\mathsf{f}(\mathsf{g}(x)) >_{\mathsf{wpo}} \mathsf{g}(\mathsf{f}(\mathsf{f}(x)))$ is verified by the following derivation:

$$\dfrac{\mathsf{f}(\mathsf{g}(x)) \geqslant_\mathcal{A} \mathsf{g}(\mathsf{f}(\mathsf{f}(x))) \qquad \mathsf{f} \succ \mathsf{g} \qquad \dfrac{\mathsf{f}(\mathsf{g}(x)) >_\mathcal{A} \mathsf{f}(\mathsf{f}(x))}{\mathsf{f}(\mathsf{g}(x)) >_{\mathsf{wpo}} \mathsf{f}(\mathsf{f}(x))}\ \text{WPO 1}}{\mathsf{f}(\mathsf{g}(x)) >_{\mathsf{wpo}} \mathsf{g}(\mathsf{f}(\mathsf{f}(x)))}\ \text{WPO 2b(i)}$$

Here WPO 1 and WPO 2b(i) indicate the corresponding conditions in Definition 1. Besides, one can verify $\mathsf{f}(\mathsf{h}(x)) >_{\mathsf{wpo}} \mathsf{h}(\mathsf{h}(\mathsf{f}(x)))$. Hence, we conclude that $\mathcal{R}$ is terminating.

In the case that the signature is finite, a rewrite order $>$ is called a *simplification order* if it has the *subterm property*, that is, $s > t$ holds whenever t is a proper subterm of a term s. For instance, WPOs satisfying the conditions of Theorem 2 are simplification orders. A TRS $\mathcal{R}$ is *simply terminating* if it has a simplification order $>$ with $\mathcal{R} \subseteq >$. The termination proving power of WPOs as a reduction order is characterized by simple termination. Needless to say, not every terminating TRS is simply terminating. Such instances are considered in Section 4.

3 Simulating WPOs by SPOs

Borralleras [1, Definition 4.1.19] introduced a variant of SPO that employs a pair of a quasi-order and a strict order. This variant compares arguments of terms by a multiset order. In order to simulate WPOs which compare arguments in a lexicographic manner, we introduce another variant of SPO.

We say that the pair $(\gtrsim, >)$ of a quasi-order $\gtrsim$ and a strict order $>$ is an *order pair* if $\gtrsim \cdot > \cdot \gtrsim \subseteq >$. We say that an order pair $(\gtrsim\!\!\!\!\!\!\sqsupset, \sqsupset)$ on $\mathcal{T}(\mathcal{F},\mathcal{V}) \setminus \mathcal{V}$ is *stable* if both $\gtrsim\!\!\!\!\!\!\sqsupset$ and $\sqsupset$ are stable.

▶ **Definition 4.** *Let $(\gtrsim\!\!\!\!\!\!\sqsupset, \sqsupset)$ be a stable order pair on $\mathcal{T}(\mathcal{F},\mathcal{V}) \setminus \mathcal{V}$. The* semantic path order $>_{\mathsf{spo}}$ (SPO) *is defined on terms as follows: $s >_{\mathsf{spo}} t$ if $s = f(s_1,\dots,s_m)$ and one of the following conditions hold:*

1. *$s_i \geqslant_{\mathsf{spo}} t$ for some $1 \leqslant i \leqslant m$.*
2. *$t = g(t_1,\dots,t_n)$ and $s >_{\mathsf{spo}} t_j$ for all $1 \leqslant j \leqslant n$, and moreover*
   a. *$s \sqsupset t$, or*
   b. *$s \gtrsim\!\!\!\!\!\!\sqsupset t$ and $(s_1,\dots,s_m) >^{\mathsf{lex}}_{\mathsf{spo}} (t_1,\dots,t_n)$.*

Here $\geqslant_{\mathsf{spo}}$ denotes the reflexive closure of $>_{\mathsf{spo}}$.

In general, semantic path orders are not closed under contexts. For a remedy, Borralleras et al. [2] propose the use of another preorder with the *harmony* property. This results in monotonic semantic path orders.

▶ **Definition 5** ([1, Definition 4.1.19])**.** *A triple $(\gtrsim, \gtrsim\!\!\!\!\!\!\sqsupset, \sqsupset)$ is a* reduction triple *if $\gtrsim$ is a rewrite preorder on terms, $(\gtrsim\!\!\!\!\!\!\sqsupset, \sqsupset)$ is a stable order pair on $\mathcal{T}(\mathcal{F},\mathcal{V}) \setminus \mathcal{V}$ with $\sqsupset$ well-founded, and $\gtrsim$ and $\gtrsim\!\!\!\!\!\!\sqsupset$ have the* harmony *property, meaning that for every $f^{(n)} \in \mathcal{F}$ the implication $s_i \gtrsim t \implies f(s_1,\dots,s_i,\dots,s_n) \gtrsim\!\!\!\!\!\!\sqsupset f(s_1,\dots,t,\dots,s_n)$ holds for all terms $s_1,\dots,s_n,t$ and argument positions $1 \leqslant i \leqslant n$.*

▶ **Definition 6.** *Let $(\gtrsim, \gtrsim\!\!\!\!\!\!\sqsupset, \sqsupset)$ be a reduction triple, and let $>_{\mathsf{spo}}$ be the semantic path order induced from $(\gtrsim\!\!\!\!\!\!\sqsupset, \sqsupset)$. The* monotonic semantic path order $s >_{\mathsf{mspo}} t$ (MSPO) *is defined as $s \gtrsim t$ and $s >_{\mathsf{spo}} t$.*

▶ **Theorem 7.** *Every monotonic semantic path order is a reduction order.* ◀

We construct a suitable order pair $(\gtrsim\!\!\!\!\!\!\sqsupset, \sqsupset)$ from the weakly monotone well-founded algebra $\mathcal{A}$ and the precedence $\succsim$ of a WPO $>_{\mathsf{wpo}}$. For terms $s = f(s_1,\dots,s_m), t = g(t_1,\dots,t_n)$ we write $s \gtrsim\!\!\!\!\!\!\sqsupset t$ if $s >_{\mathcal{A}} t$, or both $s \geqslant_{\mathcal{A}} t$ and $f \succsim g$. Similarly, we write $s \sqsupset t$ if $s >_{\mathcal{A}} t$, or both $s \geqslant_{\mathcal{A}} t$ and $f \succ g$.

▶ **Lemma 8.** *The pair $(\gtrsim\!\!\!\!\!\!\sqsupset, \sqsupset)$ is a stable order pair with $\sqsupset$ well-founded.* ◀

▶ **Lemma 9.** *The triple $(\geqslant_{\mathcal{A}}, \gtrsim\!\!\!\!\!\!\sqsupset, \sqsupset)$ forms a quasi-reduction triple.* ◀

It is worth noting that $\sqsupset$ is not always the strict part of $\gtrsim\!\!\!\!\!\!\sqsupset$. This is why the variant of SPOs is introduced.

▶ **Example 10.** Let the signature be $\{\mathsf{f}^{(1)}\}$. Consider the trivial precedence $\mathsf{f} \succsim \mathsf{f}$ and the algebra $\mathcal{A}$ over the carrier $\mathbb{N}$ with the interpretation $\mathsf{f}_{\mathcal{A}}(x) = 2x$. On the one hand we have $\mathsf{f}(\mathsf{f}(x)) \gtrsim\!\!\!\!\!\!\sqsupset \mathsf{f}(x)$ from $\mathsf{f}(\mathsf{f}(x)) \geqslant_{\mathcal{A}} \mathsf{f}(x)$ but not $\mathsf{f}(x) \gtrsim\!\!\!\!\!\!\sqsupset \mathsf{f}(\mathsf{f}(x))$ as $\mathsf{f}(x) \not\geqslant_{\mathcal{A}} \mathsf{f}(\mathsf{f}(x))$. On the other hand $\mathsf{f}(\mathsf{f}(x)) \sqsupset \mathsf{f}(x)$ does not hold. So $\sqsupset$ is not the strict part of $\gtrsim\!\!\!\!\!\!\sqsupset$.

Let $>_{\mathsf{spo}}$ be the SPO induced from $(\gtrsim\!\!\!\!\!\!\sqsupset, \sqsupset)$ and $>_{\mathsf{mspo}}$ the MSPO induced from $(\geqslant_{\mathcal{A}}, \gtrsim\!\!\!\!\!\!\sqsupset, \sqsupset)$. The following theorem is shown by a straightforward induction proof.

▶ **Theorem 11.** *The three orders* $>_{\mathsf{wpo}}, >_{\mathsf{spo}}, >_{\mathsf{mspo}}$ *coincide, provided that* $\mathcal{A}$ *is simple.* ◀

▶ **Example 12** (continued from Example 3)**.** From $\mathsf{f}(\mathsf{g}(x)) \geqslant_{\mathcal{A}} \mathsf{g}(\mathsf{f}(\mathsf{f}(x)))$ and $\mathsf{f} \succ \mathsf{g}$ the orientation $\mathsf{f}(\mathsf{g}(x)) \sqsupset \mathsf{g}(\mathsf{f}(\mathsf{f}(x)))$ is obtained. Moreover, we have $\mathsf{f}(\mathsf{g}(x)) >_{\mathcal{A}} \mathsf{f}(\mathsf{f}(x))$. Because $\geqslant_{\mathcal{A}}$ has the subterm property, the subterm $\mathsf{f}(x)$ of $\mathsf{f}(\mathsf{f}(x))$ also satisfies $\mathsf{f}(\mathsf{g}(x)) >_{\mathcal{A}} \mathsf{f}(x)$. So we obtain $\mathsf{f}(\mathsf{g}(x)) \sqsupset \mathsf{f}(\mathsf{f}(x)), \mathsf{f}(x)$. Therefore, $\mathsf{f}(\mathsf{g}(x)) >_{\mathsf{spo}} \mathsf{g}(\mathsf{f}(\mathsf{f}(x)))$ is verified as follows:

$$\dfrac{\mathsf{f}(\mathsf{g}(x)) \sqsupset \mathsf{g}(\mathsf{f}(\mathsf{f}(x))) \qquad \dfrac{\mathsf{f}(\mathsf{g}(x)) \sqsupset \mathsf{f}(\mathsf{f}(x)) \qquad \dfrac{\mathsf{f}(\mathsf{g}(x)) \sqsupset \mathsf{f}(x) \qquad \dfrac{\mathsf{g}(x) \geqslant_{\mathsf{spo}} x}{\mathsf{f}(\mathsf{g}(x)) >_{\mathsf{spo}} x}\,\text{SPO 1}}{\mathsf{f}(\mathsf{g}(x)) >_{\mathsf{spo}} \mathsf{f}(x)}\,\text{SPO 2a}}{\mathsf{f}(\mathsf{g}(x)) >_{\mathsf{spo}} \mathsf{f}(\mathsf{f}(x))}\,\text{SPO 2a}}{\mathsf{f}(\mathsf{g}(x)) >_{\mathsf{spo}} \mathsf{g}(\mathsf{f}(\mathsf{f}(x)))}\,\text{SPO 2a}$$

In addition, $\mathsf{f}(\mathsf{h}(x)) >_{\mathsf{spo}} \mathsf{h}(\mathsf{h}(\mathsf{f}(x)))$ can be verified. Hence, the inclusion $\mathcal{R} \subseteq >_{\mathsf{spo}}$ holds. Observe that the use of WPO 1 in Example 3 is replaced by SPO 1 and SPO 2a.

4 Generalized Weighted Path Orders

Theorem 11 states that weighted path orders can be defined as monotonic semantic path orders. Moreover, Lemma 9 reveals that even for non-simple algebras the construction of reduction triples is valid. This suggests a generalization of weighted path orders, which does not impose simplicity on algebras. Besides, we exploit the fact that stable order pairs need not be closed under contexts, marking root symbols of function applications [2, Definition 5].

Let $\mathcal{F}$ be a signature. For each $f \in \mathcal{F}$ we associate a marked function symbol $f^\sharp \notin \mathcal{F}$ of the same arity. The set $\{f^\sharp \mid f \in \mathcal{F}\}$ is denoted by $\mathcal{F}^\sharp$. For each term $t = f(t_1, \dots, t_n)$ in $\mathcal{T}(\mathcal{F}, \mathcal{V}) \setminus \mathcal{V}$ we denote $f^\sharp(t_1, \dots, t_n)$ by $t^\sharp$. Let $\mathcal{A}$ be a weakly monotone well-founded $(\mathcal{F} \cup \mathcal{F}^\sharp)$-algebra and $\succsim$ a precedence on $\mathcal{F}$. The pair $(\mathrel{\gtrsim\!\!\!\sqsupset}, \sqsupset)$ of relations on $\mathcal{T}(\mathcal{F}, \mathcal{V}) \setminus \mathcal{V}$ is defined as follows: Let $s = f(s_1, \dots, s_n), t = g(t_1, \dots, t_m)$. We write $s \mathrel{\gtrsim\!\!\!\sqsupset} t$ if $s^\sharp >_{\mathcal{A}} t^\sharp$, or $s^\sharp \geqslant_{\mathcal{A}} t^\sharp$ and $f \succsim g$; Similarly we write $s \sqsupset t$ if $s^\sharp >_{\mathcal{A}} t^\sharp$, or $s^\sharp \geqslant_{\mathcal{A}} t^\sharp$ and $f \succ g$. The relation $\gtrsim$ is defined as the restriction of $\geqslant_{\mathcal{A}}$ to $\mathcal{T}(\mathcal{F}, \mathcal{V})$.

▶ **Proposition 13.** *The triple* $(\gtrsim, \mathrel{\gtrsim\!\!\!\sqsupset}, \sqsupset)$ *is a reduction triple on* $\mathcal{T}(\mathcal{F}, \mathcal{V})$*.* ◀

▶ **Definition 14.** *The* generalized weighted path order (GWPO) $>_{\mathsf{gwpo}}$ *induced from* $\mathcal{A}$ *and* $\succsim$ *is the monotonic semantic path order induced from* $(\gtrsim, \mathrel{\gtrsim\!\!\!\sqsupset}, \sqsupset)$*.*

▶ **Corollary 15.** *Every generalized weighted path order is a reduction order.* ◀

▶ **Corollary 16.** *The relations* $>_{\mathsf{gwpo}}$ *and* $>_{\mathsf{wpo}}$ *coincide, provided that* $\mathcal{A}$ *is simple and* $f_{\mathcal{A}}(x_1, \dots, x_n) = f^\sharp_{\mathcal{A}}(x_1, \dots, x_n)$ *for all* $f^{(n)} \in \mathcal{F}$*.* ◀

The following two examples illustrate termination proofs by Corollary 15. Neither of the TRSs is simply terminating, and thus their termination cannot be shown by WPOs.

▶ **Example 17.** Consider the TRS $\mathcal{R}$ for round-up division:

$$\begin{array}{lll} \mathsf{p}(0) \to 0 & x - 0 \to x & 0 \div \mathsf{s}(y) \to 0 \\ \mathsf{p}(\mathsf{s}(x)) \to x & x - \mathsf{s}(y) \to \mathsf{p}(x) - y & \mathsf{s}(x) \div \mathsf{s}(y) \to \mathsf{s}((x - y) \div \mathsf{s}(y)) \end{array}$$

Let $\mathcal{A}$ be the weakly monotone algebra on $\mathbb{N}$ with the interpretations

$$\begin{array}{lllll} 0_{\mathcal{A}} = 0 & \mathsf{s}_{\mathcal{A}}(x) = x+1 & \mathsf{p}_{\mathcal{A}}(x) = x & x -_{\mathcal{A}} y = x & x \div_{\mathcal{A}} y = x \\ 0^{\sharp}_{\mathcal{A}} = 0 & \mathsf{s}^{\sharp}_{\mathcal{A}}(x) = 0 & \mathsf{p}^{\sharp}_{\mathcal{A}}(x) = 0 & x -^{\sharp}_{\mathcal{A}} y = y & x \div^{\sharp}_{\mathcal{A}} y = x + y \end{array}$$

and let $\succsim$ be an arbitrary precedence. The GWPO induced from $\mathcal{A}$ and $\succsim$ orients all rules in $\mathcal{R}$. Hence, $\mathcal{R}$ is terminating.

▶ **Example 18.** Consider the TRS $\mathcal{R}$ (`Strategy_removed_AG01_#4.28` from TPDB 11.3):

$$\begin{array}{lll} \mathsf{half}(0) \to 0 & \mathsf{half}(\mathsf{s}(0)) \to 0 & \mathsf{half}(\mathsf{s}(\mathsf{s}(x))) \to \mathsf{s}(\mathsf{half}(x)) \\ \mathsf{bits}(0) \to 0 & \mathsf{bits}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{bits}(\mathsf{half}(\mathsf{s}(x)))) & \end{array}$$

Let $\mathcal{A}$ be the weakly monotone algebra on $\mathbb{N}$ with:

$$\begin{array}{llll} 0_{\mathcal{A}} = 0 & \mathsf{s}_{\mathcal{A}}(x) = x+1 & \mathsf{half}_{\mathcal{A}}(x) = \max\{0, x-1\} & \mathsf{bits}_{\mathcal{A}}(x) = x \\ 0^{\sharp}_{\mathcal{A}} = 0 & \mathsf{s}^{\sharp}_{\mathcal{A}}(x) = x+1 & \mathsf{half}^{\sharp}_{\mathcal{A}}(x) = \max\{0, x-1\} & \mathsf{bits}^{\sharp}_{\mathcal{A}}(x) = x \end{array}$$

The GWPO $>_{\mathsf{gwpo}}$ induced by $\mathcal{A}$ and a precedence $\succsim$ with $\mathsf{half}, \mathsf{bits} \succ \mathsf{s}$ satisfies $\mathcal{R} \subseteq >_{\mathsf{gwpo}}$. So $\mathcal{R}$ is terminating.

In order to evaluate GWPOs in termination analysis we implemented a prototype termination tool based on Corollary 15. Following the automation techniques of WPO [7], we search a suitable weakly monotone well-founded algebra from two classes of algebras over $\mathbb{N}$. One is *linear interpretation* and the other is *max/plus interpretation*. Since simplicity of algebras is not required for GWPOs, we may use more general forms of interpretations.

Algebras $\mathcal{A}$ of linear interpretations use linear polynomials over $\mathbb{N}$ like Example 17. For each $f^{(n)} \in \mathcal{F} \cup \mathcal{F}^{\sharp}$ its interpretation is of the form $f_{\mathcal{A}}(x_1, \ldots, x_n) = c_0 + c_1 x_1 + \cdots + c_n x_n$ where $c_0 \in \mathbb{N}$ and $c_1, \ldots, c_n \in \{0, 1\}$.[1] Simple monotone algebras for WPOs are obtained by setting $c_1 = \cdots = c_n = 1, f_{\mathcal{A}} = f^{\sharp}_{\mathcal{A}}$ for all $f^{(n)} \in \mathcal{F}$, and those for Knuth–Bendix orders (KBOs) are obtained by further restriction for admissibility, see [7].

Algebras $\mathcal{A}$ of max/plus interpretations use a combination of $+$ and max like Example 18. For each $f^{(n)} \in \mathcal{F} \cup \mathcal{F}^{\sharp}$ its interpretation is of the form $f_{\mathcal{A}}(x_1, \ldots, x_n) = \max\{c_0, c_1 + d_1 x_1, \cdots, c_n + d_n x_n\}$ where $c_0 \in \mathbb{N}, c_1, \ldots, c_n \in \mathbb{Z}$ and $d_1, \ldots, d_n \in \{0, 1\}$. Simple monotone algebras for WPOs are obtained by imposing $c_1, \ldots, c_n \in \mathbb{N}, d_1 = \cdots = d_n = 1, f_{\mathcal{A}} = f^{\sharp}_{\mathcal{A}}$ for all $f^{(n)} \in \mathcal{F}$, and algebras for lexicographic path orders (LPOs) are obtained by further restriction $c_0 = c_1 = \cdots = c_n = 0$ for all $f^{(n)} \in \mathcal{F}$ as in [7]. The restriction $c_1, \ldots, c_n \in \mathbb{N}$ is necessary for WPOs because allowing $c_1, \ldots, c_n \in \mathbb{Z}$ results in non-simple interpretations such as $\max\{0, x-1\}$.

The problem set consists of 1511 term rewrite systems from version 11.3 of the Termination Problem Database (TPDB). The reference implementation uses the SMT solver Z3 as an external tool for solving linear constraints. The experiments were run on a PC with Intel Core i7-1065G7 CPU (1.30 GHz) and 16 GB memory.

Now let us discuss the experimental results.[2] Table 1 shows that, as a whole, use of non-simple algebras substantially improves termination analysis, at the small cost of extra

[1] If we allow $c_0 < 0$ by extending the carrier to $\mathbb{Z}$, the well-foundedness of GWPOs is lost. In fact, when $\mathsf{f}_{\mathcal{A}}(x) = \mathsf{f}^{\sharp}_{\mathcal{A}}(x) = x - 1$ and $\mathsf{g}_{\mathcal{A}}(x) = \mathsf{g}^{\sharp}_{\mathcal{A}}(x) = x - 2$, we have $\mathsf{f}(x) >_{\mathsf{gwpo}} \mathsf{f}(\mathsf{g}(x)) >_{\mathsf{gwpo}} \mathsf{f}(\mathsf{g}(\mathsf{g}(x))) >_{\mathsf{gwpo}} \cdots$.

[2] The implementation and the detailed experimental data are available at: `https://www.jaist.ac.jp/project/maxcomp/23frocos/`

Table 1 Experiments on 1511 TRSs from TPDB 11.3.

interpretations	*linear*			*max/plus*		
order	KBO	WPO	GWPO	LPO	WPO	GWPO
proved TRSs	103	122	357	149	221	385
timeouts (60 sec)	*8*	*9*	*9*	*12*	*12*	*28*

running time. In particular, in the case of linear interpretation, GWPOs significantly outperform WPOs. As a matter of fact, linear WPOs are unable to orient variable duplicating rules $\ell \to r$ such as $\mathsf{f}(x) \to \mathsf{g}(x, x)$ since $\ell \geqslant_{\mathcal{A}} r$ cannot be satisfied, but this does not apply to GWPOs based on linear interpretations which allow 0 as coefficients. In the case of max/plus interpretations there are two TRSs (with over 100 rules) that are proved to be terminating by WPOs, but not by GWPOs due to the time limit. This indicates that using non-simple algebras for max/plus interpretation can result in increase of search space. This is not the case for linear interpretations.

5 Future Work

We have introduced a generalization of WPOs whose termination proving power goes beyond the realm of simple termination. We conclude the paper by discussing future work.

General path orders. In this paper only the lexicographic versions of path orders were investigated. However, it is very likely that the same result can be obtained even if we adopt multiset comparison or status functions. General path orders (GPOs) are a unifying framework for such extensions, parameterizing the way to compare arguments. It is worth investigating simulation results between GPOs and WPOs by extending the parameters of GPOs so as to take order pairs.

Reduction pairs based on WPOs. In order to build reduction pairs for Arts and Giesl's dependency pair method, Yamada et al. [7, Section 4] extended the definition of WPOs by the notion of *partial status function*. The resulting reduction-pair version of WPOs does not impose the simplicity condition on algebras. It is not difficult to see that whenever termination of a TRS is shown by a GWPO then it can also be shown by the dependency pair method with a reduction pair based on WPOs. We anticipate that the converse also holds if we integrate the notion of partial status functions into GWPOs.

References

1 C. Borralleras. *Ordering-Based Methods for Proving Termination Automatically*. PhD thesis, Universitat Politècnica de Catalunya, 2003.

2 C. Borralleras, M. Ferreira, and A. Rubio. Complete monotonic semantic path orderings. In *Proc. 17th CADE*, volume 1831 of *LNCS (LNAI)*, pages 346–364, 2000.

3 A. Geser. On a monotonic semantic path ordering. Technical Report 92–13, Ulmer Informatik-Berichte, Universität Ulm, Germany, 1992.

4 S. Kamin and J.J. Lévy. Two generalizations of the recursive path ordering. Technical report, University of Illinois, 1980. Unpublished manuscript.

5 T. Saito and N. Hirokawa. Weighted path orders are semantic path orders. In *Proc. 14th FroCoS*, LNCS (LNAI), 2023. To appear.

6 A. Yamada. Towards a unified method for termination. In *Proc. 16th WST*, pages 248–267, 2018. The presentation slides are available at `http://wst2018.webs.upv.es/`.

7 A. Yamada, K. Kusakari, and T. Sakabe. A unified ordering for termination proving. *SCP*, 111:110–134, 2015.

Automated Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops (Short WST Version)

Nils Lommen
Eleanore Meyer
Jürgen Giesl
LuFG Informatik 2, RWTH Aachen University, Aachen, Germany

Abstract

There exist several results on deciding termination and computing runtime bounds for *triangular weakly non-linear loops* (twn-loops). We show how to use results on such subclasses of programs where complexity bounds are computable within incomplete approaches for complexity analysis of full integer programs. To this end, we present a novel modular approach which computes local runtime bounds for subprograms which can be transformed into twn-loops. These local runtime bounds are then lifted to global runtime bounds for the whole program. The power of our approach is shown by our implementation in the tool KoAT which analyzes complexity of programs where all other state-of-the-art tools fail.

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

Keywords and phrases Complexity Analysis, Upper Runtime Bounds, Decidability, Integer Programs

Related Version *See [9]. Full version, including all proofs*: `https://arxiv.org/abs/2205.08869`

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

Most approaches for automated complexity analysis of programs are based on incomplete techniques like ranking functions. However, there also exist subclasses of programs where termination is *decidable* and in [8] we presented the first subclass where runtime bounds are *computable*: For *triangular weakly non-linear loops* (twn-loops), there exist *complete* techniques for analyzing termination and runtime complexity. An example for a twn-loop is:

$$\textbf{while } (x_1^2 + x_3^5 < x_2 \wedge x_1 \neq 0) \textbf{ do } (x_1, x_2, x_3) \leftarrow (4 \cdot x_1,\ 9 \cdot x_2 - 8 \cdot x_3^3,\ x_3) \tag{1}$$

Its guard is a propositional formula over (possibly *non-linear*) polynomial inequations. The update is *triangular*, i.e., we can order the variables such that the update of any x_i does not depend on the variables $x_1, \ldots, x_{i-1}$ with smaller indices. So the restriction to triangular updates prohibits "cyclic dependencies" of variables (e.g., where the new values of x_1 and x_2 both depend on the old values of x_1 and x_2). For example, a loop whose body consists of the assignment $(x_1, x_2) \leftarrow (x_1 + x_2^2, x_2 + 1)$ is triangular, whereas a loop with the body $(x_1, x_2) \leftarrow (x_1 + x_2^2, x_1 + 1)$ is not triangular. From a practical point of view, the restriction to triangular loops seems quite natural. For example, in [5], 1511 polynomial loops were extracted from the *Termination Problems Data Base* [11], the benchmark collection which is used at the annual *Termination and Complexity Competition* [6], and only 26 of them were non-triangular.

Furthermore, the update is *weakly non-linear*, i.e., no variable x_i occurs non-linear in its own update. So for example, a loop with the body $(x_1, x_2) \leftarrow (x_1 + x_2^2, x_2 + 1)$ is weakly

non-linear, whereas a loop with the body $(x_1, x_2) \leftarrow (x_1 \cdot x_2, x_2 + 1)$ is not. With triangularity and weak non-linearity, by handling one variable after the other, one can compute a *closed form* which corresponds to applying the loop's update n times. Using these closed forms, termination can be reduced to an existential formula over $\mathbb{Z}$ [4] (whose validity is decidable for linear arithmetic and where SMT solvers often also prove (in)validity in the non-linear case). In this way, one can show that non-termination of twn-loops over $\mathbb{Z}$ is semi-decidable (and it is decidable over the real numbers). While termination of twn-loops over $\mathbb{Z}$ is not decidable, by using the closed forms, [8] presented a "*complete*" complexity analysis technique. More precisely, for every twn-loop over $\mathbb{Z}$, it infers a polynomial which is an upper bound on the runtime for all those inputs where the loop terminates. So for all (possibly non-linear) terminating twn-loops over $\mathbb{Z}$, this technique *always* computes polynomial runtime bounds. In contrast, existing tools based on incomplete techniques for complexity analysis often fail for programs with non-linear arithmetic.

In [2,7] we presented such an incomplete modular technique for complexity analysis which uses individual ranking functions for different subprograms. In this paper, we introduce a novel approach to automatically infer runtime bounds for programs possibly consisting of multiple loops by handling some subprograms as twn-loops and by using ranking functions for others. Thus, complete complexity analysis techniques for subclasses of programs with non-linear arithmetic are combined with incomplete techniques based on ranking functions.

2 Integer Programs

Let $\mathcal{V}$ be a set of variables. The set of *atoms* $\mathcal{A}(\mathcal{V})$ consists of all inequations $p_1 < p_2$ for polynomials $p_1, p_2 \in \mathbb{Z}[\mathcal{V}]$. $\mathcal{F}(\mathcal{V})$ is the set of all propositional *formulas* built from atoms $\mathcal{A}(\mathcal{V})$, $\wedge$, and $\vee$. In addition to "$<$", we also use "$\geq$", "$=$", "$\neq$", etc., and negations "$\neg$", which can be simulated by formulas (e.g., $p_1 \geq p_2$ is equivalent to $p_2 < p_1 + 1$ for integers).

For integer programs, we use a formalism based on transitions, which also allows us to represent **while**-programs like (1) easily. Formally, an integer program is a tuple $(\mathcal{V}, \mathcal{L}, \ell_0, \mathcal{T})$ with a finite set of variables $\mathcal{V}$, a finite set of locations $\mathcal{L}$, a fixed initial location $\ell_0 \in \mathcal{L}$, and a finite set of transitions $\mathcal{T}$. A *transition* is a 4-tuple $(\ell, \tau, \eta, \ell')$ with a *start location* $\ell \in \mathcal{L}$, *target location* $\ell' \in \mathcal{L} \setminus \{\ell_0\}$, *guard* $\tau \in \mathcal{F}(\mathcal{V})$, and *update* $\eta : \mathcal{V} \to \mathbb{Z}[\mathcal{V}]$. Our programs may have *non-deterministic branching*, i.e., the guards of several applicable transitions can be satisfied. To simplify the presentation, we do not consider "temporary" variables (whose update is non-deterministic), but the approach can easily be extended accordingly (see [9]).

▶ **Example 1.** Consider the program in Fig. 1 with $\mathcal{V} = \{x_i \mid 1 \leq i \leq 5\}$, $\mathcal{L} = \{\ell_0, \ell_1, \ell_2\}$, and $\mathcal{T} = \{t_i \mid 0 \leq i \leq 3\}$. We omitted trivial guards, i.e., $\tau = \texttt{true}$, and identity updates of the form $\eta(v) = v$. Here, t_3 corresponds to the **while**-program (1).

A *state* is a mapping $\sigma : \mathcal{V} \to \mathbb{Z}$, Σ denotes the set of all states, and $\mathcal{L} \times \Sigma$ is the set of *configurations*. We also apply states to arithmetic expressions p or formulas φ, where the number $\sigma(p)$ resp. the Boolean value $\sigma(\varphi)$ results from replacing each variable v by $\sigma(v)$. From now on, we fix a program $(\mathcal{V}, \mathcal{L}, \ell_0, \mathcal{T})$.

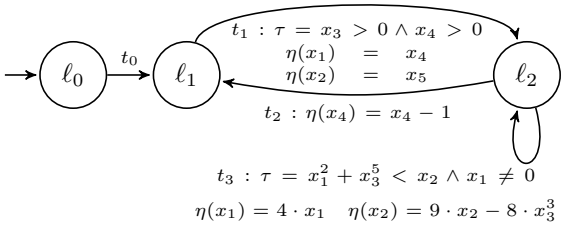


Figure 1 An Integer Program with a Nested Self-Loop

▶ **Definition 2** (Evaluation of Programs). *For configurations* (ℓ,σ), (ℓ',σ') *and* $t=(\ell_t,\tau,\eta,\ell'_t)\in\mathcal{T}$, $(\ell,\sigma)\to_t(\ell',\sigma')$ *is an* evaluation *step if* $\ell=\ell_t$, $\ell'=\ell'_t$, $\sigma(\tau)=\texttt{true}$, *and* $\sigma(\eta(v))=\sigma'(v)$ *for all* $v\in\mathcal{V}$. *Let* $\to_{\mathcal{T}}=\bigcup_{t\in\mathcal{T}}\to_t$, *where we also write* $\to$ *instead of* $\to_t$ *or* $\to_{\mathcal{T}}$. *Let* $(\ell_0,\sigma_0)\to^k(\ell_k,\sigma_k)$ *abbreviate* $(\ell_0,\sigma_0)\to\ldots\to(\ell_k,\sigma_k)$ *and let* $(\ell,\sigma)\to^*(\ell',\sigma')$ *if* $(\ell,\sigma)\to^k(\ell',\sigma')$ *for some* $k\geq 0$.

So when denoting states σ as tuples $(\sigma(x_1),\ldots,\sigma(x_5))\in\mathbb{Z}^5$, for the program in Fig. 1 we have $(\ell_0,(7,5,1,1,3))\to_{t_0}(\ell_1,(7,5,1,1,3))\to_{t_1}(\ell_2,(1,3,1,1,3))\to^2_{t_3}(\ell_2,(16,163,1,1,3))\ldots$ The *runtime complexity* $\mathrm{rc}(\sigma_0)$ of a program corresponds to the length of the longest evaluation starting in the initial state σ_0. Formally, the runtime complexity is $\mathrm{rc}\colon\Sigma\to\overline{\mathbb{N}}$ with $\overline{\mathbb{N}}=\mathbb{N}\cup\{\omega\}$ and $\mathrm{rc}(\sigma_0)=\sup\{k\in\mathbb{N}\mid\exists(\ell',\sigma').\,(\ell_0,\sigma_0)\to^k(\ell',\sigma')\}$.

3 Computing Global Runtime Bounds for Integer Programs

We now introduce our general approach for computing (upper) runtime bounds. We use weakly monotonically increasing functions as bounds, since they can easily be "composed" (i.e., if f and g increase monotonically, then so does $f\circ g$). The set of *bounds* $\mathcal{B}$ is the smallest set with $\overline{\mathbb{N}}\subseteq\mathcal{B}$, $\mathcal{V}\subseteq\mathcal{B}$, and $\{b_1{+}b_2,\ b_1{\cdot}b_2,\ k^{b_1}\}\subseteq\mathcal{B}$ for all $k\in\mathbb{N}$ and $b_1,b_2\in\mathcal{B}$. A bound constructed from $\mathbb{N}$, $\mathcal{V}$, $+$, and $\cdot$ is *polynomial*. We measure the size of variables by their absolute values. For any $\sigma\in\Sigma$, $|\sigma|$ is the state with $|\sigma|(v)=|\sigma(v)|$ for all $v\in\mathcal{V}$. So if σ_0 denotes the initial state, then $|\sigma_0|$ maps every variable to its initial absolute value. $\mathcal{RB}:\mathcal{T}\to\mathcal{B}$ is a *global runtime bound* if for each transition t and initial state $\sigma_0\in\Sigma$, $\mathcal{RB}(t)$ evaluated in the state $|\sigma_0|$ over-approximates the number of evaluations of t in any run starting in the configuration (ℓ_0,σ_0). So we have $|\sigma_0|(\mathcal{RB}(t))\geq\sup\{n\in\mathbb{N}\mid\exists(\ell',\sigma').\,(\ell_0,\sigma_0)\,(\to^*_{\mathcal{T}}\circ\to_t)^n\,(\ell',\sigma')\}$ for all $t\in\mathcal{T}$ and all states $\sigma_0\in\Sigma$ where $\to^*_{\mathcal{T}}\circ\to_t$ denotes the relation where arbitrary many evaluation steps are followed by a step with t.

For the program in Fig. 1, we have $\mathcal{RB}(t_0)=1$ (as t_0 is not on a cycle) and we will infer $\mathcal{RB}(t_i)=x_4$ for $i\in\{1,2\}$ and $\mathcal{RB}(t_3)=x_4\cdot(2\cdot x_5+1)$ in Ex. 4. By adding the bounds for all transitions, a global runtime bound $\mathcal{RB}$ yields an upper bound on the program's runtime complexity. So for all $\sigma_0\in\Sigma$ we have $|\sigma_0|(\sum_{t\in\mathcal{T}}\mathcal{RB}(t))\geq\mathrm{rc}(\sigma_0)$.

To infer global runtime bounds automatically, we first consider smaller subprograms $\mathcal{T}'\subseteq\mathcal{T}$ and compute *local runtime bounds*. A local runtime bound measures how often a transition $t\in\mathcal{T}'_>\subseteq\mathcal{T}'$ can occur in a run through $\mathcal{T}'$ that starts after an entry transition $r\in\mathcal{E}_{\mathcal{T}'}$. The *entry transitions* of $\mathcal{T}'$ are $\mathcal{E}_{\mathcal{T}'}=\{t\mid t{=}(\ell,\tau,\eta,\ell')\in\mathcal{T}\setminus\mathcal{T}'\wedge\ \text{there is a transition }(\ell',\ldots)\in\mathcal{T}'\}$. So in Fig. 1, we have $\mathcal{E}_{\mathcal{T}\setminus\{t_0\}}=\{t_0\}$ and $\mathcal{E}_{\{t_3\}}=\{t_1\}$. Thus, local runtime bounds do not consider how many $\mathcal{T}'$-runs take place in a global run and they do not consider the sizes of the variables before starting a $\mathcal{T}'$-run. We lift these local bounds to global runtime bounds for the complete program afterwards. Formally, $\mathcal{RB}_{\mathcal{T}'_>}\in\mathcal{B}$ is a *local runtime bound* for $\mathcal{T}'_>$ w.r.t. $\mathcal{T}'$ if for all $t\in\mathcal{T}'_>$, all $r\in\mathcal{E}_{\mathcal{T}'}$ with $r=(\ell,\ldots)$, and all $\sigma\in\Sigma$, we have $|\sigma|(\mathcal{RB}_{\mathcal{T}'_>})\geq\sup\{n\in\mathbb{N}\mid\exists\sigma_0,(\ell',\sigma').\,(\ell_0,\sigma_0)\to^*_{\mathcal{T}}\circ\to_r\,(\ell,\sigma)\,(\to^*_{\mathcal{T}'}\circ\to_t)^n\,(\ell',\sigma')\}$ for $\varnothing\neq\mathcal{T}'_>\subseteq\mathcal{T}'$.

▶ **Example 3.** In Fig. 1, by using the ranking function x_4, one can infer that $\mathcal{RB}_{\{t_1,t_2\}}=x_4$ is a local runtime bound for $\mathcal{T}'_>=\{t_1,t_2\}$ w.r.t. $\mathcal{T}'=\mathcal{T}'_>\cup\{t_3\}$. For $\mathcal{T}'_>=\mathcal{T}'=\{t_4\}$, our approach for twn-loops yields the local runtime bound $\mathcal{RB}_{\{t_3\}}=2\cdot x_2+1$, see Sect. 4.

If we have a local runtime bound $\mathcal{RB}_{\mathcal{T}'_>}$ w.r.t. $\mathcal{T}'$, then setting $\mathcal{RB}(t)$ to $\sum_{r\in\mathcal{E}_{\mathcal{T}'}}\mathcal{RB}(r)\cdot(\mathcal{RB}_{\mathcal{T}'_>}[v/\mathcal{SB}(r,v)\mid v\in\mathcal{V}])$ for all $t\in\mathcal{T}'_>$ yields a global runtime bound. Here, we over-approximate the number of local $\mathcal{T}'$-runs which are started by an entry transition $r\in\mathcal{E}_{\mathcal{T}'}$ by an already computed global runtime bound $\mathcal{RB}(r)$. Moreover, we instantiate each $v\in\mathcal{V}$ by a *size bound* $\mathcal{SB}(r,v)$ which is a bound on the size of v before a local $\mathcal{T}'$-run is started. To be precise,

a size bound satisfies $|\sigma_0|(\mathcal{SB}(r,v)) \geq \sup\{|\sigma'(v)| \mid \exists\, \ell' \in \mathcal{L}.\ (\ell_0,\sigma_0)\ (\to_{\mathcal{T}}^* \circ \to_r)\ (\ell',\sigma')\}$ for all $(r,v) \in \mathcal{T} \times \mathcal{V}$ and all states $\sigma_0 \in \Sigma$. Thus, our implementation alternates between runtime bound and size bound computations (see [2] for the computation of size bounds).

▶ **Example 4.** We now show how to obtain global runtime bounds from local runtime bounds in our example from Fig. 1. Here, we obtain the global runtime bound $\mathcal{RB}(t_1) = \mathcal{RB}(t_2) = \mathcal{RB}(t_0) \cdot (\mathcal{RB}_{\{t_1,t_2\}}[v/\mathcal{SB}(t_0,v) \mid v \in \mathcal{V}]) = x_4$ as $\mathcal{SB}(t_0,x_4) = x_4$, and $\mathcal{RB}(t_3) = \mathcal{RB}(t_1) \cdot (\mathcal{RB}_{\{t_3\}}[v/\mathcal{SB}(t_1,v) \mid v \in \mathcal{V}]) = x_4 \cdot (2 \cdot x_5 + 1)$ as $\mathcal{SB}(t_1,x_2) = x_5$. Thus, $\mathrm{rc}(\sigma_0) \in \mathcal{O}(n^2)$ where n is the largest initial absolute value of all variables. Our new technique allows us to use both local bounds resulting from twn-loops (for transition t_3 with non-linear arithmetic where tools based on ranking functions cannot infer a bound) and local bounds resulting from ranking functions (for t_1 and t_2).

To improve size and runtime bounds repeatedly, we treat the strongly connected components (SCCs) of the program in topological order such that improved bounds for previous transitions are already available when handling the next SCC. We first try to infer local runtime bounds by multiphase-linear ranking functions (see [7] which also contains a heuristic for choosing $\mathcal{T}'_>$ and $\mathcal{T}'$ when using ranking functions). If ranking functions do not yield finite local bounds for all transitions of the SCC, then we apply the twn-technique. Afterwards, the global runtime bound is updated accordingly. Note that the twn-approach is not only limited to self-loops but is also applicable for so-called simple cycles (see [9]).

4 Local Runtime Bounds for TWN-Loops

In this section we briefly recapitulate how we infer runtime bounds for twn-loops, based on [4, 8]. The tuple (τ,η) is a twn-loop (over the variables $\vec{x} = (x_1,\ldots,x_d)$) if $\tau \in \mathcal{F}(\mathcal{V})$ and $\eta : \mathcal{V} \to \mathbb{Z}[\mathcal{V}]$ for $\mathcal{V} = \{x_1,\ldots,x_d\}$ such that for all $1 \leq i \leq d$ we have $\eta(x_i) = c_i \cdot x_i + p_i$ for some $c_i \in \mathbb{Z}$ (where w.l.o.g. we can assume $c_i \geq 0$) and $p_i \in \mathbb{Z}[x_{i+1},\ldots,x_d]$. Our algorithm starts with computing a closed form for the loop update, which describes the values of the variables after n iterations of the loop. Formally, a tuple of arithmetic expressions $\mathtt{cl}^n_{\vec{x}} = (\mathtt{cl}^n_{x_1},\ldots,\mathtt{cl}^n_{x_d})$ over $\vec{x}$ and the distinguished variable n is a *closed form* for the update η with start value $n_0 \geq 0$ if for all $1 \leq i \leq d$ and all $\sigma : \{x_1,\ldots,x_d,n\} \to \mathbb{Z}$ with $\sigma(n) \geq n_0$, we have $\sigma(\mathtt{cl}^n_{x_i}) = \sigma(\eta^n(x_i))$. These closed forms can be represented as so-called *poly-exponential expressions*. The set of all poly-exponential expressions is defined as $\mathbb{PE} = \{\sum_{j=1}^{\ell} p_j \cdot n^{a_j} \cdot b_j^n \;\big|\; \ell, a_j \in \mathbb{N},\ p_j \in \mathbb{Q}[\mathcal{V}],\ b_j \in \mathbb{N}_{\geq 1}\}$.

▶ **Example 5.** A closed form (with start value $n_0 = 0$) for the twn-loop in (1) is $\mathtt{cl}^n_{x_1} = x_1 \cdot 4^n$, $\mathtt{cl}^n_{x_2} = (x_2 - x_3^3) \cdot 9^n + x_3^3$, and $\mathtt{cl}^n_{x_3} = x_3$.

Using the closed form, as in [4] one can represent non-termination of a twn-loop (τ,η) by the formula $\exists\, \vec{x} \in \mathbb{Z}^d,\ m \in \mathbb{N}.\ \forall n \in \mathbb{N}_{\geq m}.\ \tau[\vec{x}/\mathtt{cl}^n_{\vec{x}}]$. Here, $\tau[\vec{x}/\mathtt{cl}^n_{\vec{x}}]$ means that each variable x_i in τ is replaced by $\mathtt{cl}^n_{x_i}$. Hence, whenever $\forall n \in \mathbb{N}_{\geq m}.\ \tau[\vec{x}/\mathtt{cl}^n_{\vec{x}}]$ holds, then $\mathtt{cl}^{\max\{n_0,m\}}_{\vec{x}}$ witnesses non-termination. Thus, invalidity of the previous formula is equivalent to termination of the loop. Poly-exponential expressions have the advantage that it is always clear which addend determines their asymptotic growth when increasing n. So as in [4], the formula can be transformed into an existential formula and we use an SMT solver to prove its invalidity in order to prove termination of the loop.

As observed in [8], since the closed forms for twn-loops are poly-exponential expressions that are weakly monotonic in n, every twn-loop (τ,η) *stabilizes* for each input $\vec{e} \in \mathbb{Z}^d$. So there is a number of loop iterations (a *stabilization threshold* $\mathrm{sth}_{(\tau,\eta)}(\vec{e})$), such that the truth

	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{>2})$	$\mathcal{O}(EXP)$	$< \infty$	AVG$^+$(s)	AVG(s)
KoAT2 + TWN + MΦRF5	26	231	73	13	1	344	8.72	23.93
KoAT2 + MΦRF5	24	226	68	10	0	328	8.23	21.63
MaxCore	23	216	66	7	0	312	2.02	5.31
CoFloCo	22	196	66	5	0	289	0.62	2.66
KoAT1	25	169	74	12	6	286	1.77	2.77
Loopus	17	170	49	5	0	241	0.42	0.43
KoAT2 + TWN	20	111	3	2	0	136	2.54	26.59

Figure 2 Evaluation on the Collection CINT

value of the loop guard τ does not change anymore when performing further loop iterations. Hence, the runtime of every terminating twn-loop is bounded by its stabilization threshold. See [8] for the computation of bounds on the stabilization thresholds.

▶ **Example 6.** The stabilization threshold $\text{sth}_{(\tau,\eta)}$ of the twn-loop (1) is bounded by $2 \cdot x_2 + 1$. Thus, as the twn-loop (1) terminates, $2 \cdot x_2 + 1$ is a bound on the runtime for the loop (1). Due to the correspondence between t_3 and the loop (1), $2 \cdot x_2 + 1$ is also a local runtime bound for t_3.

5 Conclusion and Evaluation

We showed that results on subclasses of programs with computable complexity bounds like [4, 8] are not only theoretically interesting, but they have an important practical value. By integrating such complete techniques into incomplete approaches for general programs, the power of automated complexity analysis is increased substantially, in particular because now one can also infer runtime bounds for programs containing non-linear arithmetic. We evaluated this integration in our re-implementation of the tool KoAT and compared the results to other state-of-the-art tools. Let KoAT1 refer to the original tool from [2] and let KoAT2 refer to our new re-implementation [7]. We tested the following configurations of KoAT2, which differ in the techniques used for the computation of local runtime bounds:

- KoAT2+MΦRF5 uses multiphase-linear ranking functions (MΦRFs) of depth ≤ 5
- KoAT2+TWN only uses the twn-technique
- KoAT2+TWN+MΦRF5 uses the twn-technique and MΦRFs of depth ≤ 5

Fig. 2 presents our evaluation on the 504 *Complexity C Integer Programs* (CINT) used in the annual *Termination and Complexity Competition* [6]. Here, all variables are interpreted as integers over $\mathbb{Z}$ (i.e., without overflows). We compare KoAT with the tools CoFloCo [3], MaxCore [1] with CoFloCo in the backend, and Loopus [10]. In Fig. 2, the runtime bounds are compared asymptotically. So for instance, there are $26 + 231 = 257$ programs in CINT where KoAT2+TWN+MΦRF5 can show that $\text{rc}(\sigma_0) \in \mathcal{O}(n)$ holds for all initial states σ_0 where $|\sigma_0(v)| \leq n$ for all $v \in \mathcal{V}$. For 26 of these programs, KoAT2+TWN+MΦRF5 can even show that $\text{rc}(\sigma_0) \in \mathcal{O}(1)$, i.e., their runtime complexity is constant. Overall, this configuration succeeds on 344 examples, i.e., "$< \infty$" is the number of examples where a finite bound on the runtime complexity could be computed by the respective tool within the time limit of 5 minutes per example. "AVG$^+$(s)" is the average runtime of the tool on successful runs in seconds, i.e., where the tool inferred a finite time bound before reaching the timeout, whereas "AVG(s)" is the average runtime of the tool on all runs including timeouts.

KoAT's source code, a binary, and a Docker image are available at `https://koat.verify.rwth-aachen.de/twn`. The website also has details on our experiments and *web interfaces*

to run KoAT's configurations directly online.

References

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

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

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

4 F. Frohn, M. Hark, and J. Giesl. Termination of Polynomial Loops. In *Proc. SAS*, LNCS 12389, pages 89–112, 2020.

5 F. Frohn and C. Fuhs. A Calculus for Modular Loop Acceleration and Non-Termination Proofs. *STTT*, 24(5):691–715, 2022.

6 J. Giesl, A. Rubio, C. Sternagel, J. Waldmann, and A. Yamada. The Termination and Complexity Competition. In *Proc. TACAS*, LNCS 11429, pages 156–166, 2019.

7 J. Giesl, N. Lommen, M. Hark, and F. Meyer. Improving Automatic Complexity Analysis of Integer Programs. In *The Logic of Software: A Tasting Menu of Formal Methods*, LNCS 13360, pages 193–228, 2022.

8 M. Hark, F. Frohn, and J. Giesl. Polynomial Loops: Beyond Termination. In *Proc. LPAR*, EPiC 73, pages 279–297, 2020.

9 N. Lommen, F. Meyer, and J. Giesl. Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops. In *Proc. IJCAR*, LNCS 13385, pages 734–754, 2022. Full version appeared in *CoRR*, abs/2205.08869.

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

11 TPDB (Termination Problems Data Base). URL: `https://github.com/TermCOMP/TPDB`.

Proving Non-Termination by Acceleration Driven Clause Learning

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

Abstract

We recently proposed *Acceleration Driven Clause Learning* (ADCL), a novel calculus to analyze satisfiability of *Constrained Horn Clauses* (CHCs). Here, we adapt ADCL to disprove termination of transition systems, and we evaluate its implementation in our tool LoAT against the state of the art.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Non-Termination, Program Verification, Acceleration, Transition Systems

Related Version *See [8]. Full version, including all proofs*: https://arxiv.org/abs/2304.10166

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

1 Introduction

We are concerned with *dis*proving termination of *transition systems* (TSs), a popular intermediate representation for verification of programs written in more expressive languages.

▶ **Example 1.** Consider the TS $\mathcal{T}$, where x', y', z' represent the updated values of x, y, z, and $\overline{\overline{x}}, x{+}{+}, x{-}{-}$ abbreviate $x' = x$, $x' = x{+}1$, and $x' = x{-}1$. The first two transitions are a variant of `chc-LIA-Lin_052` from the *CHC Competition '22* (https://chc-comp.github.io) and the last two are a variant of `flip2_rec.jar-obl-8` from *TermComp* [9].

$$\mathsf{init} \to \ell_1 \,[\![x' \leq 0 \wedge z' \geq 5000 \wedge y' \leq z']\!] \qquad (\tau_{\mathsf{i}})$$

$$\ell_1 \to \ell_1 \,[\![y \leq 2 \cdot z \wedge x{+}{+} \wedge ((x < z \wedge \overline{\overline{y}}) \vee (x \geq z \wedge y{+}{+})) \wedge \overline{\overline{z}}]\!] \qquad (\tau_{\ell_1})$$

$$\ell_1 \to \ell_2 \,[\![x = y \wedge x > 2 \cdot z \wedge \overline{\overline{x}} \wedge \overline{\overline{y}}]\!] \qquad (\tau_{\ell_1 \to \ell_2})$$

$$\ell_2 \to \ell_2 \,[\![x = y \wedge x > 0 \wedge \overline{\overline{x}} \wedge y{-}{-}]\!] \qquad (\tau_{\ell_2}^{=})$$

$$\ell_2 \to \ell_2 \,[\![x > 0 \wedge y > 0 \wedge x' = y \wedge ((x > y \wedge y' = x) \vee (x < y \wedge \overline{\overline{y}}))]\!] \qquad (\tau_{\ell_2}^{\neq})$$

At ℓ_1, x is incremented until x reaches z. Then, x and y are incremented until y reaches $2 \cdot z{+}1$. If $x = y = c$ holds for some $c > 1$ at that point, then the execution can continue at ℓ_2 as follows: $\ell_2(c, c, c_z) \longrightarrow_{\tau_{\ell_2}^{=}} \ell_2(c, c-1, c_z) \longrightarrow_{\tau_{\ell_2}^{\neq}} \ell_2(c-1, c, c_z) \longrightarrow_{\tau_{\ell_2}^{\neq}} \ell_2(c, c, c_z) \longrightarrow_{\tau_{\ell_2}^{=}} \dots$ Here, $\ell_2(c, c, c_z)$ means that the current location is ℓ_2 and the values of x, y, and z are c, c, and c_z. (Of course, the value of z could also change arbitrarily in the transitions $\tau_{\ell_2}^{=}$ and $\tau_{\ell_2}^{\neq}$.) Thus, $\mathcal{T}$ does not terminate.

Ex. 1 is challenging for state-of-the-art tools for several reasons. First, more than 5000 steps are required to reach ℓ_2. Thus, `chc-LIA-Lin_052` is beyond the capabilities of most other state-of-the-art tools for proving reachability. Second, the pattern "$\tau_{\ell_2}^{=}$, 1^{st} disjunct of $\tau_{\ell_2}^{\neq}$, 2^{nd} disjunct of $\tau_{\ell_2}^{\neq}$" must be found to prove non-termination. Therefore, `flip2_rec.jar-obl-8` cannot be solved by other state-of-the-art termination tools.

We present an approach that can prove non-termination of systems like Ex. 1 automatically. To this end, we tightly integrate non-termination techniques into our recent

Acceleration Driven Clause Learning (ADCL) calculus [6], which has originally been designed for Constrained Horn Clauses (CHCs), but it can also be used to analyze TSs.

2 Preliminaries

We assume familiarity with basics from many-sorted first-order logic. $\mathcal{V}$ is a countably infinite set of variables and $\mathcal{A}$ is a first-order theory over a k-sorted signature $\Sigma_\mathcal{A}$ with carrier $\mathcal{C}_\mathcal{A} = (\mathcal{C}_{\mathcal{A},1}, \ldots, \mathcal{C}_{\mathcal{A},k})$. $\mathsf{QF}(\Sigma_\mathcal{A})$ is the set of all quantifier-free first-order formulas over $\Sigma_\mathcal{A}$, which are w.l.o.g. assumed to be in negation normal form, and $\mathsf{QF}_\wedge(\Sigma_\mathcal{A})$ only contains conjunctions of $\Sigma_\mathcal{A}$-literals. Given a first-order formula η over $\Sigma_\mathcal{A}$, σ is a *model* of η (written $\sigma \models_\mathcal{A} \eta$) if it is a model of $\mathcal{A}$ with carrier $\mathcal{C}_\mathcal{A}$, extended with interpretations for $\mathcal{V}$ such that η is satisfied. As usual, $\eta \equiv_\mathcal{A} \eta'$ means $\models_\mathcal{A} \eta \iff \eta'$. We write $\vec{x}$ for sequences and x_i is the i^{th} element of $\vec{x}$. We use "::" for concatenation of sequences, where we identify sequences of length 1 with their elements, so we may write, e.g., $x :: xs$ instead of $[x] :: xs$.

Transition Systems: Let $d \in \mathbb{N}$ be fixed, and let $\vec{x}, \vec{x}' \in \mathcal{V}^d$ be disjoint vectors of pairwise different variables. Each $\psi \in \mathsf{QF}(\Sigma_\mathcal{A})$ induces a relation $\longrightarrow_\psi$ on $\mathcal{C}_\mathcal{A}^d$ where $\vec{s} \longrightarrow_\psi \vec{t}$ iff $\psi[\vec{x}/\vec{s}, \vec{x}'/\vec{t}]$ is satisfiable. So for the condition $\psi := (x = y \wedge x > 0 \wedge \overline{\overline{x}} \wedge y\text{--})$ of $\tau^{=}_{\ell_2}$, we have $(4,4,4) \longrightarrow_\psi (4,3,7)$. $\mathcal{L} \supseteq \{\mathsf{init}, \mathsf{err}\}$ is a finite set of *locations*. A *configuration* is a pair $(\ell, \vec{s}) \in \mathcal{L} \times \mathcal{C}_\mathcal{A}^d$, written $\ell(\vec{s})$. A *transition* is a triple $\tau = (\ell, \psi, \ell') \in \mathcal{L} \times \mathsf{QF}(\Sigma_\mathcal{A}) \times \mathcal{L}$, written $\ell \to \ell' \,[\![\psi]\!]$, and its *condition* is $\mathsf{cond}(\tau) := \psi$. W.l.o.g., we assume $\ell \neq \mathsf{err}$ and $\ell' \neq \mathsf{init}$. Then τ induces a relation $\longrightarrow_\tau$ on configurations where $\mathfrak{s} \longrightarrow_\tau \mathfrak{t}$ iff $\mathfrak{s} = \ell(\vec{s}), \mathfrak{t} = \ell'(\vec{t})$, and $\vec{s} \longrightarrow_\psi \vec{t}$. So, e.g., $\ell_2(4,4,4) \longrightarrow_{\tau^{=}_{\ell_2}} \ell_2(4,3,7)$. We call τ *recursive* if $\ell = \ell'$, *conjunctive* if $\psi \in \mathsf{QF}_\wedge(\Sigma_\mathcal{A})$, *initial* if $\ell = \mathsf{init}$, and *safe* if $\ell' \neq \mathsf{err}$. Moreover, we define $(\ell \to \ell' \,[\![\psi]\!])|_{\psi'} := \ell \to \ell' \,[\![\psi']\!]$. A *transition system* (TS) $\mathcal{T}$ is a finite set of transitions, and it induces the relation $\longrightarrow_\mathcal{T} := \bigcup_{\tau \in \mathcal{T}} \longrightarrow_\tau$.

Chaining $\tau = \ell_s \to \ell_t \,[\![\psi]\!]$ and $\tau' = \ell'_s \to \ell'_t \,[\![\psi']\!]$ yields $\mathsf{chain}(\tau, \tau') := (\ell_s \to \ell'_t \,[\![\psi_c]\!])$ where $\psi_c := \psi[\vec{x}'/\vec{x}''] \wedge \psi'[\vec{x}/\vec{x}'']$ for fresh $\vec{x}'' \in \mathcal{V}^d$ if $\ell_t = \ell'_s$, and $\psi_c := \bot$ (meaning *false*) if $\ell_t \neq \ell'_s$. So $\longrightarrow_{\mathsf{chain}(\tau,\tau')} = \longrightarrow_\tau \circ \longrightarrow_{\tau'}$, and $\mathsf{chain}(\tau_{\ell_1 \to \ell_2}, \tau^{=}_{\ell_2}) = \ell_1 \to \ell_2 \,[\![\psi]\!]$ where $\psi \equiv_\mathcal{A} (x = y \wedge x > 2 \cdot z \wedge x > 0 \wedge \overline{\overline{x}} \wedge y\text{--})$. For non-empty, finite sequences of transitions we define $\mathsf{chain}([\tau]) := \tau$ and $\mathsf{chain}([\tau_1, \tau_2] :: \vec{\tau}) := \mathsf{chain}(\mathsf{chain}(\tau_1, \tau_2) :: \vec{\tau})$. We lift notations for transitions to finite sequences via chaining. So $\mathsf{cond}(\vec{\tau}) := \mathsf{cond}(\mathsf{chain}(\vec{\tau}))$, $\vec{\tau}$ is *recursive* if $\mathsf{chain}(\vec{\tau})$ is recursive, $\longrightarrow_{\vec{\tau}} = \longrightarrow_{\mathsf{chain}(\vec{\tau})}$, etc. If τ is initial and $\mathsf{cond}(\tau :: \vec{\tau}) \not\equiv_\mathcal{A} \bot$, then $(\tau :: \vec{\tau}) \in \mathcal{T}^+$ is a *finite run*. $\mathcal{T}$ is safe if every finite run is safe. If there is a σ such that $\sigma \models_\mathcal{A} \mathsf{cond}(\vec{\tau}')$ for every finite prefix $\vec{\tau}'$ of $\vec{\tau} \in \mathcal{T}^\omega$, then $\vec{\tau}$ is an *infinite run*. If no infinite run exists, then $\mathcal{T}$ is *terminating*.

Acceleration Techniques: *Acceleration techniques* compute transitive closures of relations.

▶ **Definition 2** (Acceleration). *An* acceleration technique *is a function* $\mathsf{accel} : \mathsf{QF}_\wedge(\Sigma_\mathcal{A}) \mapsto \mathsf{QF}_\wedge(\Sigma_{\mathcal{A}'})$ *such that* $\longrightarrow^+_\psi = \longrightarrow_{\mathsf{accel}(\psi)}$, *where* $\mathcal{A}'$ *is a first-order theory. For recursive conjunctive transitions* τ, *we define* $\mathsf{accel}(\tau) := \tau|_{\mathsf{accel}(\mathsf{cond}(\tau))}$.

Def. 2 allows $\mathcal{A}' \neq \mathcal{A}$ as most theories are not "closed under acceleration". E.g., accelerating the linear formula $x'_1 = x_1 + x_2 \wedge \overline{\overline{x_2}}$ yields $n > 0 \wedge x'_1 = x_1 + n \cdot x_2 \wedge \overline{\overline{x_2}}$, which is non-linear.

3 Proving Non-Termination with ADCL

To bridge the gap between transitions τ where $\mathsf{cond}(\tau) \in \mathsf{QF}(\Sigma_\mathcal{A})$ and acceleration techniques for formulas from $\mathsf{QF}_\wedge(\Sigma_\mathcal{A})$, ADCL uses *syntactic implicants*.

▶ **Definition 3** (Syntactic Implicants [6, Def. 6]). *If $\psi \in \mathsf{QF}(\Sigma_{\mathcal{A}})$, then:*

$$\begin{aligned}
\mathsf{sip}(\psi,\sigma) &:= \bigwedge\{\pi \text{ is a literal of } \psi \mid \sigma \models_{\mathcal{A}} \pi\} && \text{if } \sigma \models_{\mathcal{A}} \psi\\
\mathsf{sip}(\psi) &:= \{\mathsf{sip}(\psi,\sigma) \mid \sigma \models_{\mathcal{A}} \psi\}\\
\mathsf{sip}(\tau) &:= \{\tau|_\psi \mid \psi \in \mathsf{sip}(\mathsf{cond}(\tau))\} && \text{for transitions } \tau\\
\mathsf{sip}(\mathcal{T}) &:= \bigcup_{\tau\in\mathcal{T}} \mathsf{sip}(\tau) && \text{for TSs } \mathcal{T}
\end{aligned}$$

Here, sip *abbreviates* syntactic implicant projection.

While $\mathsf{sip}(\psi)$ contains $\mathsf{sip}(\psi,\sigma)$ for all models σ of ψ, the set $\mathsf{sip}(\psi)$ is finite, because $\mathsf{sip}(\psi,\sigma)$ is restricted to literals from ψ. Syntactic implicants ignore the semantics of literals. So we have, e.g., $(X > 1) \notin \mathsf{sip}(X > 0 \wedge X > 1) = \{X > 0 \wedge X > 1\}$. It is easy to show $\psi \equiv_{\mathcal{A}} \bigvee \mathsf{sip}(\psi)$, and thus $\longrightarrow_{\mathcal{T}} = \longrightarrow_{\mathsf{sip}(\mathcal{T})}$.

The core idea of ADCL is to learn new, *non-redundant* transitions via acceleration.

▶ **Definition 4** (Redundancy, [6, Def. 8]). *A transition τ is* (strictly) redundant *w.r.t. τ', denoted $\tau \sqsubseteq \tau'$ ($\tau \sqsubset \tau'$) if $\longrightarrow_\tau \subseteq \longrightarrow_{\tau'}$ ($\longrightarrow_\tau \subset \longrightarrow_{\tau'}$). For a TS $\mathcal{T}$, we have $\tau \sqsubseteq \mathcal{T}$ ($\tau \sqsubset \mathcal{T}$) if $\tau \sqsubseteq \tau'$ ($\tau \sqsubset \tau'$) for some $\tau' \in \mathcal{T}$.*

To prove non-termination, we look for a corresponding *certificate*.

▶ **Definition 5** (Certificate of Non-Termination). *Let $\tau = \ell \to \ell\,[\![\ldots]\!]$. A satisfiable formula ψ* certifies non-termination of *τ, written $\psi \models^{\infty}_{\mathcal{A}} \tau$, if for any model σ of ψ, there is an infinite sequence $\ell(\sigma(\vec{x})) = \mathfrak{s}_1 \longrightarrow_\tau \mathfrak{s}_2 \longrightarrow_\tau \ldots$*

From now on, let $\mathcal{T}$ be the TS that is being analyzed with ADCL, and assume that $\mathcal{T}$ does not contain unsafe transitions. A *state* of ADCL consists of a TS $\mathcal{S}$ that augments $\mathcal{T}$ with *learned transitions*, a run $\vec{\tau}$ of $\mathcal{S}$ called the *trace*, and a sequence of sets of *blocking transitions* $[B_i]_{i=0}^k$, where transitions that are redundant w.r.t. B_k must not be appended to the trace.

▶ **Definition 6** (ADCL). *A* state *is a triple $(\mathcal{S}, [\tau_i]_{i=1}^k, [B_i]_{i=0}^k)$ where $\mathcal{S} \supseteq \mathcal{T}$ is a TS, $\bigcup_{i=0}^k B_i \subseteq \mathsf{sip}(\mathcal{S})$, and $[\tau_i]_{i=1}^k \in \mathsf{sip}(\mathcal{S})^*$. The transitions in $\mathsf{sip}(\mathcal{T})$ are called* original *and the transitions in $\mathsf{sip}(\mathcal{S}) \setminus \mathsf{sip}(\mathcal{T})$ are* learned*. A transition $\tau_{k+1} \sqsubseteq B_k$ is* blocked*, and $\tau_{k+1} \not\sqsubseteq B_k$ is* active *if $\mathsf{chain}([\tau_i]_{i=1}^{k+1})$ is an initial transition with satisfiable condition (i.e., $[\tau_i]_{i=1}^{k+1}$ is a run). Let $\mathsf{bt}(\mathcal{S}, [\tau_i]_{i=1}^k, [B_0,\ldots,B_k]) := (\mathcal{S}, [\tau_i]_{i=1}^{k-1}, [B_0,\ldots,B_{k-1} \cup \{\tau_k\}])$, where* bt *abbreviates "backtrack". Our calculus is defined by the following rules.*

$$\frac{}{\mathcal{T} \rightsquigarrow (\mathcal{T}, [], [\varnothing])}\ (\textsc{Init}) \qquad \frac{\tau \in \mathsf{sip}(\mathcal{S}) \text{ is active}}{(\mathcal{S}, \vec{\tau}, \vec{B}) \rightsquigarrow (\mathcal{S}, \vec{\tau} :: \tau, \vec{B} :: \varnothing)}\ (\textsc{Step})$$

$$\frac{\vec{\tau}^{\circlearrowleft} \text{ is recursive} \quad |\vec{\tau}^{\circlearrowleft}| = |\vec{B}^{\circlearrowleft}| \quad \mathsf{accel}(\vec{\tau}^{\circlearrowleft}) = \tau \not\sqsubseteq \mathsf{sip}(\mathcal{S})}{(\mathcal{S}, \vec{\tau} :: \vec{\tau}^{\circlearrowleft}, \vec{B} :: \vec{B}^{\circlearrowleft}) \rightsquigarrow (\mathcal{S} \cup \{\tau\}, \vec{\tau} :: \tau, \vec{B} :: \{\tau\})}\ (\textsc{Accelerate})$$

$$\frac{\vec{\tau}^{\circlearrowleft} \text{ is recursive} \quad \vec{\tau}^{\circlearrowleft} \sqsubset \mathsf{sip}(\mathcal{S}) \text{ or } \vec{\tau}^{\circlearrowleft} \sqsubseteq \mathsf{sip}(\mathcal{S}) \wedge |\vec{\tau}^{\circlearrowleft}| > 1}{s = (\mathcal{S}, \vec{\tau} :: \vec{\tau}^{\circlearrowleft}, \vec{B}) \rightsquigarrow \mathsf{bt}(s)}\ (\textsc{Covered})$$

$$\frac{\vec{\tau} \text{ is unsafe}}{(\mathcal{S}, \vec{\tau}, \vec{B}) \rightsquigarrow \mathsf{unsafe}}\ (\textsc{Refute}) \qquad \frac{\text{all transitions from } \mathsf{sip}(\mathcal{S}) \text{ are inactive} \quad \tau \text{ is safe}}{s = (\mathcal{S}, \vec{\tau} :: \tau, \vec{B}) \rightsquigarrow \mathsf{bt}(s)}\ (\textsc{Backtrack})$$

$$\frac{\mathsf{chain}(\vec{\tau}^{\circlearrowleft}) = \ell \to \ell\,[\![\ldots]\!] \quad \psi \models^{\infty}_{\mathcal{A}} \vec{\tau}^{\circlearrowleft} \quad \tau = \ell \to \mathsf{err}\,[\![\psi]\!] \not\sqsubseteq \mathsf{sip}(\mathcal{S})}{(\mathcal{S}, \vec{\tau} :: \vec{\tau}^{\circlearrowleft}, \vec{B}) \rightsquigarrow (\mathcal{S} \cup \{\tau\}, \vec{\tau} :: \vec{\tau}^{\circlearrowleft}, \vec{B})}\ (\textsc{Nonterm})$$

We write $\overset{\text{I}}{\rightsquigarrow}$, $\overset{\text{S}}{\rightsquigarrow}$, ... to indicate that the rule Init, Step, ... was used. Step adds a transition to the trace. When the trace has a recursive suffix, Accelerate allows for learning a new

transition which replaces the recursive suffix on the trace, or we may backtrack via Covered if the recursive suffix is redundant. Note that Covered does not apply if $\vec{\tau}' \sqsubseteq \mathsf{sip}(\mathcal{S})$ and $|\vec{\tau}'| = 1$, as it could immediately undo every Step, otherwise. If no further Step is possible, Backtrack applies. Note that Backtrack and Covered block the last transition from the trace so that we do not perform the same Step again. If $\vec{\tau}$ is unsafe, Refute yields unsafe. As $\mathcal{T}$ is safe, this only happens if Nonterm, which applies a non-termination technique to a recursive suffix of the trace, added an unsafe transition before.

▶ **Example 7.** We apply ADCL to Ex. 1

$$\begin{aligned}
\mathcal{T} &\overset{\mathrm{I}}{\rightsquigarrow} (\mathcal{T}, [], [\varnothing]) \overset{\mathrm{S}}{\rightsquigarrow}{}^{2} (\mathcal{T}, [\tau_{\mathsf{i}}, \tau_{\ell_1}|_{\psi_{x<z}}], [\varnothing, \varnothing, \varnothing]) && (x \le 1 \wedge z \ge 5k \wedge y \le z)\\
&\overset{\mathrm{A}}{\rightsquigarrow} (\mathcal{S}_1, [\tau_{\mathsf{i}}, \tau^+_{x<z}], [\varnothing, \varnothing, \{\tau^+_{x<z}\}]) && (x \le z \wedge z \ge 5k \wedge y \le z)\\
&\overset{\mathrm{S}}{\rightsquigarrow} (\mathcal{S}_1, [\tau_{\mathsf{i}}, \tau^+_{x<z}, \tau_{\ell_1}|_{\psi_{x\ge z}}], [\varnothing, \varnothing, \{\tau^+_{x<z}\}, \varnothing]) && (x = z+1 \wedge z \ge 5k \wedge y \le z+1)\\
&\overset{\mathrm{A}}{\rightsquigarrow} (\mathcal{S}_2, [\tau_{\mathsf{i}}, \tau^+_{x<z}, \tau^+_{x\ge z}], [\varnothing, \varnothing, \{\tau^+_{x<z}\}, \{\tau^+_{x\ge z}\}]) && (x \ge y \wedge x > z \ge 5k \wedge y \le 2\cdot z+1)\\
&\overset{\mathrm{S}}{\rightsquigarrow}{}^{4}_{\mathsf{nt}} (\mathcal{S}_2, [\tau_{\mathsf{i}}, \tau^+_{x<z}, \tau^+_{x\ge z}, \tau_{\ell_1\to\ell_2}, \tau^{=}_{\ell_2}, \tau^{\neq}_{\ell_2}|_{\psi_{x>y}}, \tau^{\neq}_{\ell_2}|_{\psi_{x<y}}], [\ldots]) && (1 \equiv_2 y = x > 10k \wedge \ldots)\\
&\overset{\mathrm{N}}{\rightsquigarrow}_{\mathsf{nt}} (\mathcal{S}_3, [\tau_{\mathsf{i}}, \tau^+_{x<z}, \tau^+_{x\ge z}, \tau_{\ell_1\to\ell_2}, \tau^{=}_{\ell_2}, \tau^{\neq}_{\ell_2}|_{\psi_{x>y}}, \tau^{\neq}_{\ell_2}|_{\psi_{x<y}}], [\ldots]) && (1 \equiv_2 y = x > 10k \wedge \ldots)\\
&\overset{\mathrm{S}}{\rightsquigarrow}_{\mathsf{nt}} (\mathcal{S}_3, [\tau_{\mathsf{i}}, \tau^+_{x<z}, \tau^+_{x\ge z}, \tau_{\ell_1\to\ell_2}, \tau^{=}_{\ell_2}, \tau^{\neq}_{\ell_2}|_{\psi_{x>y}}, \tau^{\neq}_{\ell_2}|_{\psi_{x<y}}, \tau_{\mathsf{err}}], [\ldots]) \overset{\mathrm{R}}{\rightsquigarrow}_{\mathsf{nt}} \mathsf{unsafe}
\end{aligned}$$

Here, $5k$ abbreviates 5000 and:

$$\begin{aligned}
&\psi_{x<z} := y \le 2\cdot z \wedge x{+}{+} \wedge x < z \wedge \bar{\bar{y}} \wedge \bar{\bar{z}} && \psi_{x\ge z} := y \le 2\cdot z \wedge x{+}{+} \wedge x \ge z \wedge y{+}{+} \wedge \bar{\bar{z}}\\
&\tau^+_{x<z} := \ell_1 \to \ell_1\, [\![y \le 2\cdot z \wedge n > 0 \wedge x' = x + n \wedge x + n \le z \wedge \bar{\bar{y}} \wedge \bar{\bar{z}}]\!]\\
&\tau^+_{x\ge z} := \ell_1 \to \ell_1\, [\![y + n - 1 \le 2\cdot z \wedge n > 0 \wedge x' = x + n \wedge x \ge z \wedge y' = y + n \wedge \bar{\bar{z}}]\!]\\
&\psi_{x>y} := x > 0 \wedge y > 0 \wedge x' = y \wedge x > y \wedge y' = x && \psi_{x<y} := x > 0 \wedge y > 0 \wedge x' = y \wedge x < y \wedge \bar{\bar{y}}\\
&\mathcal{S}_1 := \mathcal{T} \cup \{\tau^+_{x<z}\} \qquad \mathcal{S}_2 := \mathcal{S}_1 \cup \{\tau^+_{x\ge z}\} \qquad \mathcal{S}_3 := \mathcal{S}_2 \cup \{\tau_{\mathsf{err}}\} \qquad \tau_{\mathsf{err}} := \ell_2 \to \mathsf{err}\, [\![x = y > 1]\!]
\end{aligned}$$

On the right, we show formulas describing the configurations that are reachable with the current trace, where $1 \equiv_2 y$ means that y is odd. Every $\rightsquigarrow$-derivation starts with Init. The first two Steps add the initial transition τ_{i} and an element of $\mathsf{sip}(\tau_{\ell_1})$ to the trace. Since $x < z$ holds after applying τ_{i}, the only possible choice for the latter is $\tau_{\ell_1}|_{\psi_{x<z}}$.

As $\tau_{\ell_1}|_{\psi_{x<z}}$ is recursive, it is accelerated and replaced with $\mathsf{accel}(\tau_{\ell_1}|_{\psi_{x<z}}) = \tau^+_{x<z}$, which simulates n steps with $\tau_{\ell_1}|_{\psi_{x<z}}$. Moreover, $\tau^+_{x<z}$ is also added to the current set of blocking transitions, as we always have $\longrightarrow^2_\tau \subseteq \longrightarrow_\tau$ for learned transitions τ and thus adding them to the trace twice in a row is pointless.

Next, τ_{ℓ_1} is applicable again. As neither $x < z$ nor $x \ge z$ holds for all reachable configurations, we could continue with any element of $\mathsf{sip}(\tau_{\ell_1}) = \{\tau_{\ell_1}|_{\psi_{x<z}}, \tau_{\ell_1}|_{\psi_{x\ge z}}\}$. We choose $\tau_{\ell_1}|_{\psi_{x\ge z}}$, so that the recursive transition $\tau_{\ell_1}|_{\psi_{x\ge z}}$ can be accelerated to $\tau^+_{x\ge z}$.

After the next Step with $\tau_{\ell_1\to\ell_2}$, just $\tau^{=}_{\ell_2}$ can be used, as $\mathsf{cond}(\tau_{\ell_1\to\ell_2})$ implies $x' = y'$. While $\tau^{=}_{\ell_2}$ is recursive, Accelerate cannot be applied next, as $\longrightarrow_{\tau^{=}_{\ell_2}} = \longrightarrow^+_{\tau^{=}_{\ell_2}}$, so the learned transition would be redundant. Thus, we continue with $\tau^{\neq}_{\ell_2}$, projected to $x > y$ (as $\mathsf{cond}(\tau^{=}_{\ell_2})$ implies $x' = y' + 1$). Again, all transitions that could be learned are redundant, so Accelerate does not apply. We next use $\tau^{\neq}_{\ell_2}$ projected to $x < y$, as the previous Step swapped x and y. As the suffix $[\tau^{=}_{\ell_2}, \tau^{\neq}_{\ell_2}|_{\psi_{x>y}}, \tau^{\neq}_{\ell_2}|_{\psi_{x<y}}]$ of the trace does not terminate (see Ex. 1), Nonterm applies. So we learn the transition τ_{err}, which is added to the trace to finish the proof, afterwards.

	No		Yes	Runtime overall			Runtime No	
	solved	unique	solved	average	median	timeouts	average	median
LoAT ADCL	521	9	0	48.6 s	0.1 s	183	2.9 s	0.1 s
LoAT '22	494	2	0	7.4 s	0.1 s	0	6.2 s	0.1 s
T2	442	3	615	17.2 s	0.6 s	45	7.4 s	0.6 s
VeryMax	421	6	631	28.3 s	0.5 s	30	30.5 s	14.5 s
iRankFinder	409	0	642	32.0 s	2.0 s	93	12.3 s	1.7 s

▶ **Theorem 8.** *If* $\mathcal{T} \rightsquigarrow_{\mathsf{nt}}^{*} \mathsf{unsafe}$, *then* $\mathcal{T}$ *does not terminate.*

See [8] for a discussion of obstacles regarding an adaption of ADCL for *proving* termination.

4 Implementation and Experiments

So far, our implementation in our tool LoAT is restricted to integer arithmetic. It uses the technique from [5] for acceleration and finding certificates of non-termination, the SMT solvers Z3 [11] and Yices [4], the recurrence solver PURRS [1], and libFAUDES (https://fgdes.tf.fau.de/faudes) to implement the automata-based redundancy check from [6].

To evaluate our implementation in LoAT, we used the 1222 *Integer Transition Systems* (ITSs) from the *Termination Problems Database* (https://termination-portal.org/wiki/TPDB) used in *TermComp* [9]. We compared our implementation (LoAT ADCL) with other leading termination analyzers: iRankFinder [3], T2 [2], VeryMax [10], and the previous version of LoAT [5] (LoAT '22). For T2 and VeryMax, we took the versions of their last *TermComp* participations (2015 and 2019). For iRankFinder, we used the configuration from the evaluation of [5], which is tailored towards proving non-termination. All tests were run on StarExec with 300s wallclock timeout, 1200s CPU timeout, and 128GB memory limit per example.

The table above shows the results of our experiments, where the column "unique" contains the number of examples that could be solved by the respective tool, but no others. It shows that LoAT ADCL is the most powerful tool for proving non-termination of ITSs.

If we only consider the examples where non-termination is proven, LoAT ADCL is also the fastest tool. If we consider all examples, then the *average* runtime of LoAT ADCL is significantly slower. This is not surprising, as ADCL does not terminate in general [6, Thm. 18]. So while it is very fast in most cases (as witnessed by the very fast *median* runtime), it times out more often than the other tools. Note that LoAT ADCL does not subsume LoAT '22. The reason is that LoAT '22 under-approximates more aggressively and hence solves some instances where LoAT ADCL times out.

See [7] for detailed results and a pre-compiled binary. LoAT is open-source and available on GitHub: https://github.com/LoAT-developers/LoAT

References

1 Roberto Bagnara, Andrea Pescetti, Alessandro Zaccagnini, and Enea Zaffanella. PURRS: Towards computer algebra support for fully automatic worst-case complexity analysis. *CoRR*, abs/cs/0512056, 2005.

2 Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. T2: Temporal property verification. In *TACAS '16*, LNCS 9636, 2016.

3 Jesús J. Doménech and Samir Genaim. iRankFinder. In *WST '18*, 2018.

4 Bruno Dutertre. Yices 2.2. In *CAV '14*, LNCS 8559, 2014.

5 Florian Frohn and Jürgen Giesl. Proving non-termination and lower runtime bounds with LoAT. In *IJCAR '22*, LNCS 13385, 2022.

6 Florian Frohn and Jürgen Giesl. ADCL: Acceleration Driven Clause Learning for constrained Horn clauses. In *SAS '23*, LNCS, 2023. To appear. Full version appeared in *CoRR*, abs/2303.01827.

7 Florian Frohn and Jürgen Giesl. Empirical evaluation of "Proving non-termination by Acceleration Driven Clause Learning", 2023. URL: https://loat-developers.github.io/adcl-nonterm-eval.

8 Florian Frohn and Jürgen Giesl. Proving non-termination by Acceleration Driven Clause Learning. In *CADE '23*, LNCS, 2023. To appear. Full version appeared in *CoRR*, abs/2304.10166.

9 Jürgen Giesl, Albert Rubio, Christian Sternagel, Johannes Waldmann, and Akihisa Yamada. The termination and complexity competition. In *TACAS '19*, LNCS 11429, 2019.

10 Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving non-termination using Max-SMT. In *CAV '14*, LNCS 8559, 2014.

11 Leonardo de Moura and Nikolay Bjørner. Z3: An efficient SMT solver. In *TACAS '08*, LNCS 4963, 2008.

Binary Non-Termination in Term Rewriting and Logic Programming

Étienne Payet
LIM - Université de la Réunion, France

Abstract

We present a new syntactic criterion for the automatic detection of non-termination in an abstract setting that encompasses a simplified form of term rewriting and logic programming.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Theory of computation → Rewrite systems; Theory of computation → Program analysis

Keywords and phrases Non-Termination, Term Rewriting, Logic Programming

1 Introduction

This paper is concerned with non-termination in structures where one rewrites elements using indexed binary relations. Such structures can be formalised by *abstract reduction systems* (ARSs) [3], *i.e.*, couples $(A, \Rightarrow_I)$ where A is a set and $\Rightarrow_I$ (the rewrite relation) is the union of binary relations on A, indexed by a set I, *i.e.*, $\Rightarrow_I = \bigcup\{\Rightarrow_\iota \mid \iota \in I\}$. Non-termination in these structures can be formalised as the existence of an infinite rewrite sequence $a_0 \Rightarrow_{\iota_0} a_1 \Rightarrow_{\iota_1} \cdots$. *Term rewrite systems (TRSs)* and *logic programs (LPs)* are concrete instances of ARSs: A is the set of finite terms and I indicates what rule (= a couple of finite terms) is applied at what position. A crucial difference is that the rewrite relation of TRSs relies on instantiation while that of LPs relies on narrowing, *i.e.*, on unification. In this paper, we present a new syntactic criterion for the automatic detection of non-termination in an abstract setting that encompasses a simplified form of term rewriting and logic programming. Namely, we suppose that the rewriting always takes place at the root position of terms (see Def. 4 below). There exist program transformation techniques that make it possible to place oneself in such a context, *e.g.*, the overlap closure [8] in term rewriting or the binary unfoldings [4, 6] in logic programming preserve the non-termination property of the original program.

2 Preliminaries

We let $\mathbb{N}$ denote the set of non-negative integers.

2.1 Binary Relations

If $\Rightarrow$ and $\hookrightarrow$ are binary relations on a set A, then $\Rightarrow \circ \hookrightarrow$ denotes their *composition*. We let $\Rightarrow^0$ be the identity relation and, for all $n \in \mathbb{N}$, $\Rightarrow^{n+1} = (\Rightarrow^n \circ \Rightarrow)$. Moreover, $\Rightarrow^* = \bigcup\{\Rightarrow^n \mid n \geq 0\}$ is the *reflexive and transitive closure* of $\Rightarrow$. We formalise non-termination as the existence of an infinite sequence of connected elements:

▶ **Definition 1.** *Let $\Rightarrow$ be a binary relation on a set A. A $\Rightarrow$-*chain *is a (possibly infinite) sequence $a_0, a_1, \ldots$ of elements of A such that $a_n \Rightarrow a_{n+1}$ for all $n \in \mathbb{N}$. We simply write it as $a_0 \Rightarrow a_1 \Rightarrow \cdots$.*

2.2 Terms

We use the same definitions and notations as [3] for terms. From now on, we fix a *signature* Σ (the *function symbols*) together with an infinite countable set X of *variables*, with $\Sigma \cap X = \emptyset$.

We let $\mathsf{f}, \mathsf{g}, \mathsf{s}$ be function symbols of positive arity and $\mathsf{0}$ be a constant symbol. The set of all *terms* built from Σ and X is denoted by $T(\Sigma, X)$. A *context* is a term with at least one "hole", represented by $\Box$, in it. For all terms or contexts t, we let $Var(t)$ denote the set of variables occurring in t and, for all contexts c, we let $c[t]$ denote the term or context obtained from c by replacing all the occurrences of $\Box$ by t. For all contexts c, we let $c^0 = \Box$ and, for all $n \in \mathbb{N}$, $c^{n+1} = c[c^n]$. Terms are generally denoted by a, s, t, u, v, variables by x, y and contexts by c, possibly with subscripts and quotes.

The set $S(\Sigma, X)$ of all *substitutions* consists of the functions θ from X to $T(\Sigma, X)$ such that $Dom(\theta) = \{x \in X \mid \theta(x) \neq x\}$ is finite. A substitution θ is usually written as $\{x \mapsto \theta(x) \mid x \in Dom(\theta)\}$ and its application to a term s as $s\theta$. A *renaming* is a substitution that is a bijection on X. The *composition* of substitutions σ and θ is denoted as $\sigma\theta$. We say that σ is *more general than* θ if $\theta = \sigma\eta$ for some substitution η. We let $\theta^0 = \emptyset$ (the identity substitution) and, for all $n \in \mathbb{N}$, $\theta^{n+1} = \theta^n\theta$.

A term s is an *instance* of a term t if $s = t\theta$ for some $\theta \in S(\Sigma, X)$. On the other hand, s *unifies* with t if $s\theta = t\theta$ for some $\theta \in S(\Sigma, X)$; then, θ is called a *unifier* of s and t and $mgu(s, t)$ denotes the *most general unifier* (mgu) of s and t.

2.3 Term Rewriting and Logic Programming

We refer to [3] (resp. [1]) for the basics of term rewriting (resp. logic programming).

▶ **Definition 2.** *A* program *is a subset of $T(\Sigma, X)^2$, every element (u, v) of which is called a* rule*, where u (resp. v) is the* left-hand side *(resp.* right-hand side*). For each program P, we let $\overline{P}$ denote the set of all finite, non-empty, sequences of elements of P.*

In this paper, we only consider ARSs $(A, \Rightarrow_I)$ such that $A = T(\Sigma, X)$ and I is a program. Hence the following simplified definition.

▶ **Definition 3.** *An* abstract reduction system (ARS) *is a union of binary relations on $T(\Sigma, X)$ indexed by a program, i.e., it has the form $\Rightarrow_P = \bigcup\{\Rightarrow_r \subseteq T(\Sigma, X)^2 \mid r \in P\}$ for some program P. For each ARS $\Rightarrow_P$ and each $\omega = (r_1, \ldots, r_n)$ in $\overline{P}$, we let $\Rightarrow_\omega = (\Rightarrow_{r_1} \circ \cdots \circ \Rightarrow_{r_n})$.*

The next definition introduces term rewrite systems and logic programs as concrete instances of ARSs. For all terms s and rules (u, v) and (u', v'), we write $(u, v) \ll_s (u', v')$ to denote that (u, v) is a *variant* of (u', v') *variable disjoint* with s, *i.e.*, for some renaming γ, we have $u = u'\gamma$, $v = v'\gamma$ and $Var(u) \cap Var(s) = Var(v) \cap Var(s) = \emptyset$.

▶ **Definition 4.** *For each program P, we let $\to_P = \bigcup\{\to_r \mid r \in P\}$ and $\rightsquigarrow_P = \bigcup\{\rightsquigarrow_r \mid r \in P\}$ where, for all $r \in P$,*

$$\underset{r}{\to} = \big\{\big(u\theta, v\theta\big) \in T(\Sigma, X)^2 \;\big|\; (u, v) = r,\ \theta \in S(\Sigma, X)\big\} \qquad \textit{(Term Rewriting)}$$

$$\underset{r}{\rightsquigarrow} = \big\{\big(s, v\theta\big) \in T(\Sigma, X)^2 \;\big|\; (u, v) \ll_s r,\ \theta = mgu(s, u)\big\} \qquad \textit{(Logic Programming)}$$

We say that $\to_P$ (resp. $\rightsquigarrow_P$) is a term rewrite system *(resp. a* logic program*).*

▶ **Example 5.** Let $r = \big(\mathsf{f}(x), \mathsf{s}(x)\big) = (u, v)$. Then, $\mathsf{f}^2(x) \to_r \mathsf{s}(\mathsf{f}(x))$ because $\mathsf{f}^2(x) = u\theta$ and $\mathsf{s}(\mathsf{f}(x)) = v\theta$ for $\theta = \{x \mapsto \mathsf{f}(x)\}$. Let $r' = \big(\mathsf{f}(\mathsf{g}(x, \mathsf{0})), \mathsf{f}(x)\big)$ and $s = \mathsf{f}(\mathsf{g}(x, x))$. The rule $(u', v') = (\mathsf{f}(\mathsf{g}(x', \mathsf{0})), \mathsf{f}(x'))$ is a variant of r' variable disjoint with s. Let $\theta' = \{x \mapsto \mathsf{0}, x' \mapsto \mathsf{0}\}$. Then, $\theta' = mgu\,(s, u')$ and we have $s \rightsquigarrow_{r'} v'\theta'$, *i.e.*, $\mathsf{f}(\mathsf{g}(x, x)) \rightsquigarrow_{r'} \mathsf{f}(\mathsf{0})$.

In term rewriting and in logic programming (modulo a condition), the left-hand side of a rule can be rewritten to the corresponding instance of the right-hand side.

▶ **Lemma 6.** *Let $r = (u, v)$ be a rule and θ be a substitution. We have $u\theta \to_r v\theta$ and, if $Var(v) \subseteq Var(u)$, $u\theta \rightsquigarrow_r v\theta$.*

3 Binary Non-Termination

We are interested in *binary chains*, *i.e.*, infinite chains that consist of the repetition of two sequences of rules. There are ARSs that admit such chains but no infinite chain consisting of the repetition of a single sequence (see, *e.g.*, $\to_P$ in Ex. 8 and Ex. 9 below). More precisely:

▶ **Definition 7.** *Let $\Rightarrow_P$ be an ARS and $\omega_1, \omega_2 \in \overline{P}$. A $(\omega_1, \omega_2, \Rightarrow_P)$-*chain *is an infinite $(\Rightarrow_{\omega_1}^* \circ \Rightarrow_{\omega_2})$-chain.*

▶ **Example 8.** Let $\Rightarrow_P \in \{\to_P, \rightsquigarrow_P\}$ where P is the program that consists of the rules

$$r_1 = \big(\mathsf{f}(x, \mathsf{s}(y)), \mathsf{f}(\mathsf{s}^2(x), y)\big) \qquad r_2 = \big(\mathsf{f}(x, \mathsf{0}), \mathsf{f}(\mathsf{s}(\mathsf{0}), x)\big)$$

(see [13] and `TRS_Standard/Zantema_15/ex12.xml` in [11]). We have the $(r_1, r_2, \Rightarrow_P)$-chain:

$$\mathsf{f}(\mathsf{s}(\mathsf{0}), \mathsf{0}) \underset{r_1}{\overset{0}{\Rightarrow}} \mathsf{f}(\mathsf{s}(\mathsf{0}), \mathsf{0}) \underset{r_2}{\Rightarrow} \mathsf{f}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0})) \underset{r_1}{\overset{1}{\Rightarrow}} \mathsf{f}(\mathsf{s}^3(\mathsf{0}), \mathsf{0}) \underset{r_2}{\Rightarrow} \mathsf{f}(\mathsf{s}(\mathsf{0}), \mathsf{s}^3(\mathsf{0})) \underset{r_1}{\overset{3}{\Rightarrow}} \cdots$$

▶ **Example 9.** Let $\Rightarrow_P \in \{\to_P, \rightsquigarrow_P\}$ where P is the program that consists of the rules

$$r_1 = \big(\mathsf{f}(x, \mathsf{s}(y)), \mathsf{f}(\mathsf{s}(x), y)\big) \qquad r_2 = \big(\mathsf{f}(x, \mathsf{0}), \mathsf{f}(x, \mathsf{s}(x))\big)$$

(see [13] and `TRS_Standard/Zantema_15/ex14.xml` in [11]). We have the $(r_1, r_2, \Rightarrow_P)$-chain:

$$\mathsf{f}(\mathsf{0}, \mathsf{s}(\mathsf{0})) \underset{r_1}{\overset{1}{\Rightarrow}} \mathsf{f}(\mathsf{s}(\mathsf{0}), \mathsf{0}) \underset{r_2}{\Rightarrow} \mathsf{f}(\mathsf{s}(\mathsf{0}), \mathsf{s}^2(\mathsf{0})) \underset{r_1}{\overset{2}{\Rightarrow}} \mathsf{f}(\mathsf{s}^3(\mathsf{0}), \mathsf{0}) \underset{r_2}{\Rightarrow} \mathsf{f}(\mathsf{s}^3(\mathsf{0}), \mathsf{s}^4(\mathsf{0})) \underset{r_1}{\overset{4}{\Rightarrow}} \cdots$$

Now, we present a criterion for the detection of binary chains. It is tailored to deal with specific sequences ω_1 and ω_2 that each consist of a single rule of a particular form. Intuitively, the rule r_1 of ω_1 and the rule r_2 of ω_2 are mutually recursive; in r_1, a context c is removed from the left-hand side to the right-hand side while, in r_2, c is added again. Ex. 8 and Ex. 9 are concrete instances, with $c = \mathsf{s}(\Box)$. This is formalised as follows.

▶ **Definition 10.** *A* recurrent pair *for a program P is a pair $(r_1, r_2) \in P^2$ such that*

- $r_1 = \big(\mathsf{f}(x, c[y]), \mathsf{f}(c^{n_1}[x], y)\big)$ *and* $r_2 = \big(\mathsf{f}(x, s), \mathsf{f}(c^{n_2}[t], c^{n_3}[x])\big)$
- $x \neq y$
- $\mathit{Var}(c) = \mathit{Var}(s) = \emptyset$
- $t \in \{x, s\}$

▶ **Example 11.** In Ex. 8, we have $(n_1, n_2, n_3) = (2, 1, 0)$, $c = \mathsf{s}(\Box)$ and $s = t = \mathsf{0}$. In Ex. 9, we have $(n_1, n_2, n_3) = (1, 0, 1)$, $c = \mathsf{s}(\Box)$, $s = \mathsf{0}$ and $t = x$.

We show that the existence of a recurrent pair leads to that of a binary chain (see Prop. 20), provided that property (1) below is satisfied. The rest of this section is parametric in an ARS $\Rightarrow_P$ and a recurrent pair (r_1, r_2) for P as in Def. 10, with $r_1 = (u_1, v_1)$ and $r_2 = (u_2, v_2)$. We suppose that we have

$$\forall \theta \in S(\Sigma, X)\ (u_1\theta \underset{r_1}{\Rightarrow} v_1\theta) \wedge (u_2\theta \underset{r_2}{\Rightarrow} v_2\theta) \tag{1}$$

As $\mathit{Var}(v_1) \subseteq \mathit{Var}(u_1)$ and $\mathit{Var}(v_2) \subseteq \mathit{Var}(u_2)$, by Lem. 6 both $\to_p$ and $\rightsquigarrow_P$ satisfy (1).

For the sake of readability, we introduce the following notation.

▶ **Definition 12.** *For all $m, n \in \mathbb{N}$, we let $\mathsf{f}(m, n)$ denote the term $\mathsf{f}(c^m[s], c^n[s])$.*

Then, we have the following two lemmas. Lem. 13 states that r_1 allows one to iteratively move a tower of c's from the second to the first argument of f. Conversely, Lem. 14 states that r_2 allows one to copy a tower of c's from the first to the second argument of f in just one step.

▶ **Lemma 13.** *For all $m, n \in \mathbb{N}$, $\mathsf{f}(m,n) \Rightarrow_{r_1}^{n} \mathsf{f}(n_1 \times n + m, 0)$.*

Proof. We proceed by induction on n.

- (Base: $n = 0$) Here, $\Rightarrow_{r_1}^{n}$ is the identity. Hence, for all $m \in \mathbb{N}$, we have $\mathsf{f}(m,n) \Rightarrow_{r_1}^{n} \mathsf{f}(m,n)$, where $\mathsf{f}(m,n) = \mathsf{f}(n_1 \times n + m, 0)$.
- (Induction) Suppose that for some $n \in \mathbb{N}$ we have $\mathsf{f}(m,n) \Rightarrow_{r_1}^{n} \mathsf{f}(n_1 \times n + m, 0)$ for all $m \in \mathbb{N}$. Let $m \in \mathbb{N}$. Then, $\mathsf{f}(m, n+1) = \mathsf{f}(c^m[s], c^{n+1}[s]) = u_1\{x \mapsto c^m[s], y \mapsto c^n[s]\}$. Therefore, by (1), we have $\mathsf{f}(m, n+1) \Rightarrow_{r_1} v_1\{x \mapsto c^m[s], y \mapsto c^n[s]\}$ where $v_1\{x \mapsto c^m[s], y \mapsto c^n[s]\} = \mathsf{f}(c^{n_1+m}[s], c^n[s]) = \mathsf{f}(n_1 + m, n)$. But, by induction hypothesis, we have $\mathsf{f}(n_1 + m, n) \Rightarrow_{r_1}^{n} \mathsf{f}(n_1 \times n + (n_1 + m), 0)$, *i.e.*, $\mathsf{f}(n_1 + m, n) \Rightarrow_{r_1}^{n} \mathsf{f}(n_1 \times (n+1) + m, 0)$. Finally, $\mathsf{f}(m, n+1) \Rightarrow_{r_1}^{n+1} \mathsf{f}(n_1 \times (n+1) + m, 0)$.

◀

▶ **Lemma 14.** *For all $m \in \mathbb{N}$, $\mathsf{f}(m,0) \Rightarrow_{r_2} \mathsf{f}(m' + n_2, m + n_3)$ where $m' = 0$ if $t = s$ and $m' = m$ if $t = x$.*

Proof. Let $m \in \mathbb{N}$. We have $\mathsf{f}(m,0) = \mathsf{f}(c^m[s], s) = u_2\{x \mapsto c^m[s]\}$. Hence, by (1), we have $\mathsf{f}(m,0) \Rightarrow_{r_2} v_2\{x \mapsto c^m[s]\}$.

- If $t = s$ then $v_2\{x \mapsto c^m[s]\} = \mathsf{f}(c^{n_2}[s], c^{m+n_3}[s]) = \mathsf{f}(n_2, m + n_3)$.
- If $t = x$ then $v_2\{x \mapsto c^m[s]\} = \mathsf{f}(c^{m+n_2}[s], c^{m+n_3}[s]) = \mathsf{f}(m + n_2, m + n_3)$.

◀

We consider the following polynomials in the indeterminate $i \in \mathbb{N}$. We define them in a mutually recursive way, which reflects the mutually recursive nature of r_1 and r_2 and hence facilitates the proof of the existence of a $(r_1, r_2, \Rightarrow_P)$-chain (Prop. 20 below).

▶ **Definition 15.** *We let*

- $\Pi_0(i) = n_2$ *and* $\Pi'_0(i) = n_3$
- $\Pi_{n+1}(i) = \Delta_n(i) + n_2$ *and* $\Pi'_{n+1}(i) = \Delta'_n(i) + n_3$ *for all* $n \in \mathbb{N}$

where, for all $n \in \mathbb{N}$,

- $\Delta_n(i) = 0$ *if* $t = s$ *and* $\Delta_n(i) = \Delta'_n(i)$ *if* $t = x$
- $\Delta'_n(i) = i\Pi'_n(i) + \Pi_n(i)$.

▶ **Example 16.** In Ex. 9, we have $t = x$ and $(n_1, n_2, n_3) = (1, 0, 1)$. Hence:

- $\Pi_0(i) = n_2 = 0$ and $\Pi'_0(i) = n_3 = 1$
- $\Pi_1(i) = \Delta_0(i) + n_2 = \Delta'_0(i) = i\Pi'_0(i) + \Pi_0(i) = i$
- $\Pi'_1(i) = \Delta'_0(i) + n_3 = i + 1$
- $\Pi_2(i) = \Delta_1(i) + n_2 = \Delta'_1(i) = i\Pi'_1(i) + \Pi_1(i) = i^2 + i + i = i^2 + 2i$
- $\Pi'_2(i) = \Delta'_1(i) + n_3 = i^2 + 2i + 1$

The next lemma provides a simpler form of Π and Π' for the case $t = s$ (the case $t = x$ is more intricate).

▶ **Lemma 17.** *If $t = s$ then, for all $n \in \mathbb{N}$, $\Pi_n(i) = n_2$ and $\Pi'_n(i) = n_3 i^n + \sum_{k=0}^{n-1}(n_2 + n_3)i^k$.*

Proof. Suppose that $t = s$. Then, for all $n \in \mathbb{N}$, $\Delta_n(i) = 0$, so $\Pi_{n+1}(i) = n_2$. As $\Pi_0(i) = n_2$ also, for all $n \in \mathbb{N}$ we have $\Pi_n(i) = n_2$. Now, we prove that $\Pi'_n(i) = n_3 i^n + \sum_{k=0}^{n-1}(n_2 + n_3)i^k$. We proceed by induction on n.

- (Base: $n = 0$) We have $\Pi'_n(i) = n_3 = n_3 i^n + \sum_{k=0}^{n-1}(n_2 + n_3)i^k$.
- (Induction) Suppose that the property holds for some $n \in \mathbb{N}$. We have $\Pi'_{n+1}(i) = \Delta'_n(i) + n_3 = i\Pi'_n(i) + \Pi_n(i) + n_3$. But, as $t = s$, $\Pi_n(i) = n_2$ and, by induction hypothesis, $\Pi'_n(i) = n_3 i^n + \sum_{k=0}^{n-1}(n_2 + n_3)i^k$. So, $\Pi'_{n+1}(i) = i(n_3 i^n + \sum_{k=0}^{n-1}(n_2 + n_3)i^k) + n_2 + n_3 = n_3 i^{n+1} + \sum_{k=0}^{n}(n_2 + n_3)i^k$.

◀

▶ **Example 18.** In Ex. 8, we have $t = s$ and $(n_1, n_2, n_3) = (2, 1, 0)$. Hence, by Lem. 17, we have $\Pi_n(i) = 1$ and $\Pi'_n(i) = \sum_{k=0}^{n-1} i^k$ for all $n \in \mathbb{N}$.

Using Π and Π', we define the set of terms A:

▶ **Definition 19.** *We let* $A = \{a_n = \mathsf{f}(\Pi_n(n_1), \Pi'_n(n_1)) \mid n \in \mathbb{N}\}$.

Now we prove the existence of the $(r_1, r_2, \Rightarrow_P)$-chain

$$a_0 \, (\overset{\Pi'_0(n_1)}{\underset{r_1}{\Rightarrow}} \circ \underset{r_2}{\Rightarrow}) \, a_1 \, (\overset{\Pi'_1(n_1)}{\underset{r_1}{\Rightarrow}} \circ \underset{r_2}{\Rightarrow}) \, a_2 \, (\overset{\Pi'_2(n_1)}{\underset{r_1}{\Rightarrow}} \circ \underset{r_2}{\Rightarrow}) \cdots$$

▶ **Proposition 20.** *For all* $n \in \mathbb{N}$, *we have* $a_n \, (\Rightarrow_{r_1}^{\Pi'_n(n_1)} \circ \Rightarrow_{r_2}) \, a_{n+1}$.

Proof. Let $n \in \mathbb{N}$. We have $a_n = \mathsf{f}(\Pi_n(n_1), \Pi'_n(n_1))$. By Lem. 13 and Lem. 14,

$$a_n \overset{\Pi'_n(n_1)}{\underset{r_1}{\Rightarrow}} \mathsf{f}\left(\underbrace{n_1 \times \Pi'_n(n_1) + \Pi_n(n_1)}_{\Delta'_n(n_1)}, 0\right) \underset{r_2}{\Rightarrow} \mathsf{f}\left(m, \underbrace{\Delta'_n(n_1) + n_3}_{\Pi'_{n+1}(n_1)}\right)$$

where $m = n_2 = \Pi_{n+1}(n_1)$ if $t = s$ and $m = \Delta'_n(n_1) + n_2 = \Pi_{n+1}(n_1)$ if $t = x$. Hence, $a_n \, (\Rightarrow_{r_1}^{\Pi'_n(n_1)} \circ \Rightarrow_{r_2}) \, a_{n+1}$. ◀

▶ **Example 21.** In Ex. 8, we have $\Pi_n(i) = 1$ and $\Pi'_n(i) = \sum_{k=0}^{n-1} i^k$ for all $n \in \mathbb{N}$ (see Ex. 18). We also have $n_1 = 2$ and the $(r_1, r_2, \Rightarrow_P)$-chain:

$$\underbrace{\mathsf{f}(\mathsf{s}(0), 0)}_{a_0} \overset{\Pi'_0(n_1)}{\underset{r_1}{\Rightarrow}} \mathsf{f}(\mathsf{s}(0), 0) \underset{r_2}{\Rightarrow} \underbrace{\mathsf{f}(\mathsf{s}(0), \mathsf{s}(0))}_{a_1} \overset{\Pi'_1(n_1)}{\underset{r_1}{\Rightarrow}} \mathsf{f}(\mathsf{s}^3(0), 0) \underset{r_2}{\Rightarrow} \underbrace{\mathsf{f}(\mathsf{s}(0), \mathsf{s}^3(0))}_{a_2} \overset{\Pi'_2(n_1)}{\underset{r_1}{\Rightarrow}} \cdots$$

▶ **Example 22.** In Ex. 9, we have $\Pi_0(n_1) = 0$, $\Pi'_0(n_1) = 1$, $\Pi_1(n_1) = 1$, $\Pi'_1(n_1) = 2$, $\Pi_2(n_1) = 3$, $\Pi'_2(i) = 4$, ... (see Ex. 16). We have the $(r_1, r_2, \Rightarrow_P)$-chain:

$$\underbrace{\mathsf{f}(0, \mathsf{s}(0))}_{a_0} \overset{\Pi'_0(n_1)}{\underset{r_1}{\Rightarrow}} \mathsf{f}(\mathsf{s}(0), 0) \underset{r_2}{\Rightarrow} \underbrace{\mathsf{f}(\mathsf{s}(0), \mathsf{s}^2(0))}_{a_1} \overset{\Pi'_1(n_1)}{\underset{r_1}{\Rightarrow}} \mathsf{f}(\mathsf{s}^3(0), 0) \underset{r_2}{\Rightarrow} \underbrace{\mathsf{f}(\mathsf{s}^3(0), \mathsf{s}^4(0))}_{a_2} \overset{\Pi'_2(n_1)}{\underset{r_1}{\Rightarrow}} \cdots$$

4 Future Work and Implementation

We plan to investigate how our work relates to the forms of non-termination detected by the approaches of [5, 7, 12]. We have no clear idea for the moment.

Our tool NTI (Non-Termination Inference) [9] is designed to automatically prove the existence of infinite chains in TRSs and in LPs. It first transforms the original program P into a program P': for TRSs, it uses the dependency pairs combined with a variant of the overlap closure [10] and, for LPs, it uses the binary unfolding [4, 6]. By [2, 4, 8], non-termination of P' implies that of P. Then, it detects recurrent pairs (Def. 10), hence binary chains (Prop. 20), in P'.

References

1 K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall International series in computer science. Prentice Hall, 1997.

2 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

3 F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

4 M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999. doi:10.1016/S0743-1066(99)00006-0.

5 F. Emmes, T. Enger, and J. Giesl. Proving non-looping non-termination automatically. In B. Gramlich, D. Miller, and U. Sattler, editors, *Proc. of the 6th International Joint Conference on Automated Reasoning (IJCAR'12)*, volume 7364 of *LNCS*, pages 225–240. Springer, 2012. doi:10.1007/978-3-642-31365-3_19.

6 M. Gabbrielli and R. Giacobazzi. Goal independency and call patterns in the analysis of logic programs. In H. Berghel, T. Hlengl, and J. E. Urban, editors, *Proc. of the 1994 ACM Symposium on Applied Computing (SAC'94)*, pages 394–399. ACM Press, 1994. doi:10.1145/326619.326789.

7 A. Geser and H. Zantema. Non-looping string rewriting. *RAIRO Theoretical Informatics and Applications*, 33(3):279–302, 1999. doi:10.1051/ita:1999118.

8 J. V. Guttag, D. Kapur, and D. R. Musser. On proving uniform termination and restricted termination of rewriting systems. *SIAM Journal of Computing*, 12(1):189–214, 1983.

9 NTI (Non-Termination Inference). http://lim.univ-reunion.fr/staff/epayet/Research/NTI/NTI.html and https://github.com/etiennepayet/nti.

10 É. Payet. Guided unfoldings for finding loops in standard term rewriting. In F. Mesnard and P. J. Stuckey, editors, *Proc. of the 28th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'18), Revised Selected Papers*, volume 11408 of *LNCS*, pages 22–37. Springer, 2018. doi:10.1007/978-3-030-13838-7_2.

11 Termination Problems Data Base. http://termination-portal.org/wiki/TPDB.

12 Y. Wang and M. Sakai. On non-looping term rewriting. In A. Geser and H. Søndergaard, editors, *Proc. of the 8th International Workshop on Termination (WST'06)*, pages 17–21, 2006.

13 H. Zantema and A. Geser. *Non-looping rewriting.* Universiteit Utrecht. UU-CS, Department of Computer Science. Utrecht University, Netherlands, 1996.

A Verified Efficient Implementation of the Weighted Path Order*

René Thiemann
University of Innsbruck, Austria

Elias Wenninger
University of Innsbruck, Austria

Abstract

The Weighted Path Order of Yamada is a powerful technique for proving termination. It is also supported by CeTA, a certifier for checking untrusted termination proofs. To be more precise, CeTA contains a verified function that computes for two terms whether one of them is larger than the other for a given WPO, i.e., where all parameters of the WPO have been fixed. The problem of this verified function is its exponential runtime in the worst case.

Therefore, in this work we develop a polynomial time implementation of WPO that is based on memoization. It also improves upon an earlier verified implementation of the Recursive Path Order: the RPO-implementation uses full terms as keys for the memory, a design which simplified the soundness proofs, but has some runtime overhead. In this work, keys are just numbers, so that the lookup in the memory is faster. Although trivial on paper, this change introduces some challenges for the verification task.

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

Keywords and phrases certification, Isabelle/HOL, reduction pair, termination analysis

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) [10] was introduced as a simplification order that unifies and extends classical ones.

Since implementations may be buggy, we are interested in certification, where automatically generated proofs are checked with verified implementations of the various orders. Since the verification task is usually time-intensive, one wants to minimize the number of supported orders. Here, WPO is a perfect candidate for integration into a checker, since it covers many simplification orders. Note that a direct implementation of WPO as recursive function has an exponential worst case runtime. Therefore, we will develop a non-trivial implementation that only requires polynomial time. It is based on memoization and its verification is not immediate.

We perform our formalization using Isabelle/HOL [4], based on IsaFoR, the **Isa**belle **Fo**rmalization of **R**ewriting [7], which also contains soundness proofs of the certifier CeTA. As a result of this work we improved the execution times of CeTA when it comes to checking proofs of WPO. We further replaced the existing implementation of RPO by instantiating the new efficient implementation of WPO, again leading to faster implementation. The

* The authors are listed in alphabetical order regardless of individual contributions or seniority. This research was supported by the Austrian Science Fund (FWF) project I 5943.

formalization of the new implementation is available in the archive of formal proofs (AFP) [8] and the new implementation is integrated in CeTA version 2.45.

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 reduction pair is a pair $(\succ, \succsim)$ of two relations on terms that satisfies the following requirements: $\succ$ is well-founded, $\succsim$ and $\succ$ are compatible (i.e., $\succsim \circ \succ \circ \succsim \subseteq \succ$), and the relations are closed under certain operations.

A precedence is a relation $>$ on $\mathcal{F}$ that is both transitive and well-founded.

We write $\succ^{\mathsf{lex}}$ and $\succsim^{\mathsf{lex}}$ for the strict- and non-strict lexicographic extension of $(\succ, \succsim)$. They are defined by a comparison from left to right, e.g., $[s_1, \ldots, s_n] \succ^{\mathsf{lex}} [t_1, \ldots, t_n]$ is satisfied iff there is some $1 \leq i \leq n$ such that $s_j \succsim t_j$ for all $1 \leq j < i$, and $s_i \succ t_i$.

3 The Weighted Path Order

In this section we provide a recursive definition of a variant of WPO. We deviate from the original definition by dropping the status function π of WPO, since this feature is not important to illustrate the results of this paper. Actually, the formalization of WPO [5] includes several currently known extensions of WPO, e.g., quasi-precedences, comparison of variables with least elements, the support of multiset comparisons as an alternative to lexicographic comparisons [9], and Refinements (2c) and (2d) of WPO [10, Section 4.2].

Let $>$ be some precedence. Let $(\succ, \succsim)$ be some reduction pair such that $\succ$ is transitive, $\succsim$ is a preorder, and $t \succsim s$ for all terms t and subterms s of t. Then WPO is defined by a strict and non-strict relation on terms ($\succ_{\mathsf{WPO}}$ and $\succsim_{\mathsf{WPO}}$) as follows: $s \succ_{\mathsf{WPO}} t$ iff

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

The relation $s \succsim_{\mathsf{WPO}} t$ is defined in the same way, where $\succ^{\mathsf{lex}}_{\mathsf{WPO}}$ in case **2.b.ii.B** is replaced by $\succsim^{\mathsf{lex}}_{\mathsf{WPO}}$, and there is an additional subcase that includes $x \succsim^{\mathsf{lex}}_{\mathsf{WPO}} x$. Yamada proved that $(\succ_{\mathsf{WPO}}, \succsim_{\mathsf{WPO}})$ is a reduction pair, and can thus be used for termination proofs.

The previous verified implementation of WPO just implements the recursive definition of WPO directly. This can lead to exponentially many recursive calls.

▶ **Example 1.** We consider a WPO where $\succ = \emptyset$ and $\succsim$ relates all terms. Let us now evaluate $f^{n+1}(x) \succ_{\mathsf{WPO}} f^{n+1}(x)$. We have to consider case **2.a** where we arrive at $f^n(x) \succ_{\mathsf{WPO}} f^{n+1}(x)$ and then in case **2.b.i** a recursive call to $f^n(x) \succ_{\mathsf{WPO}} f^n(x)$ is triggered. For evaluating $f^{n+1}(x) \succ_{\mathsf{WPO}} f^{n+1}(x)$ we further have to apply case **2.b.ii.B** to arrive at $[f^n(x)] \succ^{\mathsf{lex}}_{\mathsf{WPO}} [f^n(x)]$ which in turn will again evaluate $f^n(x) \succ_{\mathsf{WPO}} f^n(x)$.

So, evaluation of $f^{n+1}(x) \succ_{\mathsf{WPO}} f^{n+1}(x)$ will result in at least two invocations of $f^n(x) \succ_{\mathsf{WPO}} f^n(x)$, which in turn leads to four invocations of $f^{n-1}(x) \succ_{\mathsf{WPO}} f^{n-1}(x)$, then eight invocations of $f^{n-2}(x) \succ_{\mathsf{WPO}} f^{n-2}(x)$, etc.

4 The Efficient Implementation of the Weighted Path Order

The exponential runtime of WPO as seen in Example 1 is also present in other term orders. For instance, a memoized implementation of RPO was developed by Nagele to circumvent these problems [6]. There, for an RPO comparison of terms s and t a dictionary was used to lookup and store the results of RPO comparisons of subterms. The keys of the dictionary were pairs of terms (s_i, t_j) where s_i is a subterm of s and t_j is a subterm of t. Consequently, each lookup and store operation on the dictionary (implemented via red-black trees) requires several term comparisons, which clearly is not optimal when it comes to execution time. And from the formalization viewpoint it also had its costs, namely the approach required the definition of a linear order on terms, which was not automatic at that time.

Similarly, our approach also adds memoization to WPO, but avoiding the problem of term comparisons in dictionary operations. We use pairs of integer indices as keys instead, which have the advantage of being easy and fast to compare. The indices are uniquely assigned to all subterms of s and t at the very beginning using an index function. This index function adds a consecutive integer value to every subterm in the term.

Technically this is done as follows. Terms are represented by a datatype ('f,'v)term, where 'f is a type variable for function symbols and 'v for variables. Indexed terms are then defined as terms with function symbols and variables adjusted.

```
type_synonym ('f,'v)indexed_term = ('f × ('f,'v)term × index, 'v × ('f,'v)term × index)term
```

In this way an indexed term is a term, so one can recurse on its structure. Additionally one has constant time access to the index of a term (index :: ('f,'v)indexed_term ⇒ index) and to the original term (stored :: ('f,'v)indexed_term ⇒ ('f,'v)term) by accessing the additional informations that are stored in the root symbols, i.e., variables and function symbols.

The crucial property of the function to create an index (index_term) is formulated as:

```
lemma index_term: ∃ ri. ∀ t. index_term s ⊵ t ⟶ ri (index t) = unindex t ∧ stored t = unindex t
```

Here, unindex is the function to recursively flatten an indexed term to a term by removing the extra informations in the variables and function symbols.[1] The existentially quantified function ri (**r**everse **i**ndex) is important to show that each index uniquely identifies a subterm. The equality unindex (index_term t) = t is also available.

We can now define a memoized implementation of WPO as follows: it takes two indexed terms s and t and a dictionary d as input and then returns the result $(s \succ_{\mathsf{WPO}} t, s \succsim_{\mathsf{WPO}} t)$ in combination with an updated dictionary. Here, wpo_mem is the function which tries to perform a lookup, and if this is not successful, stores the result of the computation of wpo_main, which in turn implements the inference rules of WPO and recurses via wpo_mem.

```
fun wpo_mem and wpo_main where
wpo_mem d (s,t) = (let i = index s; j = index t in
  case lookup d (i,j) of Some result ⇒ (result, d)      (* use memoized result if available *)
    | None ⇒ case wpo_main mem (s,t)                    (* otherwise compute result *)
      of (result, d') ⇒ (result, update (i,j) result d'))   (* and memoize it *)

wpo_main d (s,t) = (if stored s ≻ stored t then ((True, True), d)      (* WPO case 1 *)
```

[1] A previous name of the unindex function was flatten. This old name is still present in AFP 2022 and CeTA version 2.45, but it will be unindex in future versions of the AFP and CeTA.

```
  else if stored s ≿ stored t then ((False, False), d)          (* WPO case 2 not applicable *)
  else case s of Fun f ss ⇒ case exists_mem (λ s_i. (s_i,t)) wpo_mem snd d ss
      of (wpo_2a, d') ⇒ if wpo_2a then ((True, True), d')              (* WPO case 2.a *)
        else ...)
```

Observe that in the (full) definition of wpo_main several higher-order functions have been replaced by higher-order memoized functions, e.g., the existential quantification over list elements in WPO case **2.a** is implemented via the memoized version exists_mem; similarly there are forall_mem for case **2.b.i** and lex_ext_mem for case **2.b.ii.B**.

In the memoized implementation of WPO each pair of subterms $s_i \trianglelefteq s$ and $t_j \trianglelefteq t$ is compared at most once, leading to at most $|s| \times |t|$ many invocations of wpo_main to compute $s \succ_{\mathsf{WPO}} t$. If we assume that $\succ$ and $\succsim$ can be evaluated in polynomial time, then we get an overall polynomial runtime for the variant of WPO that we presented here.[2]

Now the question is, how we can relate the original implementation wpo :: ('f,'v)term × ('f,'v)term ⇒ bool × bool to its memoized implementation wpo_mem :: (index × index, bool × bool) mapping ⇒ ('f,'v)indexed_term × ('f,'v)indexed_term ⇒ (bool × bool) × (index × index, bool × bool) mapping. One prerequisite is to enforce that the dictionary only stores valid results. Note, however, that in our setting the dictionary stores indices of actual inputs. Therefore, we define a valid dictionary for a function f as follows, where rev_ind is the inverse of an index function, i.e., it computes from a given index of type 'i the original value of type 'a which is then used as input to f.

```
definition valid_memory :: ('a ⇒ 'b) ⇒ ('i ⇒ 'a) ⇒ ('i, 'b) mapping ⇒ bool where
  valid_memory f rev_ind mem = (∀ i b. lookup mem i = Some b ⟶ f (rev_ind i) = b)
```

For proving soundness of wpo_mem we require—among others—the precondition valid_memory wpo ri d where ri is a combination of the two reverse index functions of s and t. Both of these exist by lemma index_term. Note that ri does not occur in the implementation.

Before we can prove soundness of wpo_mem, we also need to prove soundness of auxiliary memoized functions such as exists_mem. At this point, let us have a look a bit more closely on how exists_mem is actually defined and used. To this end, consider the type of exists_mem.

```
('a ⇒ 'b) ⇒ ('m ⇒ 'b ⇒ ('c × 'm)) ⇒ ('c ⇒ bool) ⇒ 'm ⇒ 'a list ⇒ (bool × 'm)
```

This type clearly deviates from the usual type of a function that checks whether some list element satisfies a predicate: ('a ⇒ bool) ⇒ 'a list ⇒ bool.

The second argument to exists_mem is a memoized function that, for a given input of type 'b and some input memory of type 'm, computes the result of type 'c and an updated memory. Note, however, that the input list takes elements of type 'a which first need to be converted to inputs for the memoized function. This is exactly the task of the function that is provided as the first argument of exists_mem. Once we get the result of type 'c from the memoized implementation, we further need to convert this into a Boolean to check the predicate, and this is done by the third argument of exists_mem.

In the WPO-example, we need the full flexibility of exists_mem. The memoized function is wpo_mem which works on pairs of indexed terms and computes pairs of Booleans. Here the first argument to exists_mem is a function that takes some arbitrary subterm s_i from the arguments of $s = f(ss)$ with $ss = [s_1, \ldots, s_n]$ and combines the single indexed term s_i into

[2] The version of WPO in the AFP includes several improvements. With these improvements, polynomial runtime is still ensured when fixing a finite signature, but not if arities can grow arbitrarily.

the pair (s_i, t) such that wpo_mem can be invoked on such a pair. Then the resulting pair $(s_i \succ_{\mathsf{WPO}} t, s_i \succsim_{\mathsf{WPO}} t)$ is converted by snd into $s_i \succsim_{\mathsf{WPO}} t$ so that eventually we compute $\exists s_i.\ s_i \succsim_{\mathsf{WPO}} t$, i.e., we check precisely whether WPO case **2.a** is applicable.

For making this reasoning formal, we further need a notion to express that a memoized function (e.g., wpo_mem) is an implementation of some function (e.g., wpo).

For space reasons we only provide the definition and soundness lemma here, without going into detail. Note that the handling of the indirection of indices makes the definitions and properties more complicated than they are in Nagele's formalization without this indirection.

```
definition memoize_fun impl f g rev_ind A =
  (∀ x m y m'. valid_memory f rev_ind m ⟶ impl m x = (y,m') ⟶ x ∈ A ⟶
      y = f (g x) ∧ valid_memory f rev_ind m')

lemma exists_mem: assumes valid_memory fun rev_ind m
  and exists_mem f impl h m xs = (b, m')
  and memoize_fun impl fun g rev_ind (f ' set xs)
shows b = (∃ x ∈ set xs. h (fun (g (f x)))) ∧ valid_memory fun rev_ind m'
```

We arrive at the main soundness lemma which shows that wpo_mem implements wpo.

```
lemma wpo_mem: assumes ri = (λ (i, j). (rli i, rri j))
  and ⋀ si. s ⊵ si ⟹ rli (index si) = unindex si ∧ stored si = flatten si
  and ⋀ ti. t ⊵ ti ⟹ rri (index ti) = unindex ti ∧ stored ti = unindex ti
  and valid_memory wpo ri d
  and wpo_mem d (s,t) = (result,d')
shows result = wpo (unindex s, unindex t) ∧ valid_memory wpo ri d'
```

We finally define a wrapper that invokes wpo_mem with an empty memory:

```
definition wpo_mem_impl s t = fst (wpo_mem Mapping.empty (index_term s, index_term t))
```

Soundness of wpo_mem_impl is easily proven with the help of lemmas index_term and wpo_mem. The [code] attribute tells the code generator of Isabelle to implement wpo via the defining equations of the memoized implementation.

```
lemma wpo_mem_impl[code]: wpo s t = wpo_mem_impl s t
```

Based on wpo_mem and wpo_mem_impl we also obtain rpo_mem and rpo_mem_impl, a memoized implementation of RPO. It just instantiates WPO[3] by choosing the trivial reduction pair, i.e., $\succ = \emptyset$ and $\succsim$ compares all terms. In this way, case **1** of WPO is never applicable and the condition $s \succsim t$ in case **2** is always satisfied.

5 Evaluation

The new implementations of WPO and RPO are integrated in CeTA version 2.45 and we experimentally evaluate it against CeTA version 2.44 which uses a non-memoized WPO and Nagele's implementation of RPO. For the experiments we use TRSs $\mathcal{R}_n$ whose termination

[3] Recall that the formalization contains a variant of WPO that also supports multiset comparisons. In this way RPO is fully subsumed by WPO.

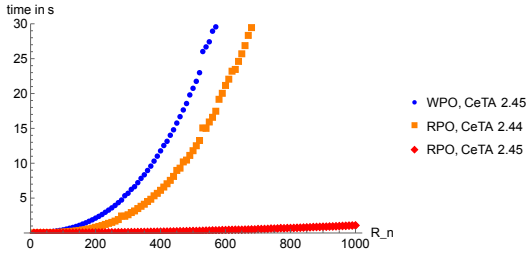


Figure 1 Results of running WPO and RPO with CeTA versions 2.44 and 2.45.

can be proven by both RPO and WPO. Eleven of the twelve rules of $\mathcal{R}_n$ are identical for each n, and there is one rule that is parametrized by n as follows.

$$\mathsf{f}(\mathit{term}(n), \mathsf{g}(\mathsf{s}(y))) \to \mathsf{f}(\mathit{term}'(n), \mathsf{s}(\mathsf{s}(\mathsf{g}(y))))$$

Here, both $\mathit{term}(n)$ and $\mathit{term}'(n)$ are terms of the form $(\mathsf{g}/\mathsf{h})^n(x)$ where each g/h is randomly replaced by either g or h. For instance, $\mathit{term}(3)$ might be $\mathsf{g}(\mathsf{h}(\mathsf{h}(x)))$ and $\mathit{term}'(3)$ is $\mathsf{h}(\mathsf{h}(\mathsf{g}(x)))$. If you download CeTA version 2.45,[4] then $\mathcal{R}_{48}$ is available as `examples/wpo_large.proof.xml`, though in a variant where the $(\mathsf{g}/\mathsf{h})^{48}(x)$ terms have a non-random structure.

We tested the implementations on $\mathcal{R}_{10}, \mathcal{R}_{20}, \ldots, \mathcal{R}_{1000}$ and the results are displayed in Figure 1. All experiments have been conducted using an 3.2 GHz 8-Core Intel Xeon W processor running macOS Ventura 13.4.

- The RPO implementation in CeTA 2.44 needs cubic time ($\approx 0.09 \cdot n^3$ μs) to certify the proofs: $\mathcal{O}(n^2)$ many term pairs are compared, and each lookup has a cost of $\mathcal{O}(n)$.
- The WPO implementation in CeTA 2.44 can only solve $\mathcal{R}_{10}$ and needs 46 seconds. The attempt to certify $\mathcal{R}_{20}$ was aborted after 10 minutes.
- The new version of RPO in CeTA 2.45 needs only quadratic time ($\approx 1.10 \cdot n^2$ μs), since now the lookup costs are negligible. So, there is a linear speedup compared to CeTA 2.44.
- The new version of WPO in CeTA 2.45 needs cubic time ($\approx 0.11 \cdot n^3$ μs), since each comparison in WPO cases **1** and **2** requires $\mathcal{O}(n)$ time for the chosen parameters.

References

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

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

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

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

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

[4] Available at `http://cl-informatik.uibk.ac.at/isafor/src/CeTA-2.45.tgz`.

6 René Thiemann, Guillaume Allais, and Julian Nagele. On the formalization of termination techniques based on multiset orderings. In *Proc. RTA 2012*, volume 15 of *LIPIcs*, pages 339–354, 2012. doi:10.4230/LIPIcs.RTA.2012.339.

7 René Thiemann and Christian Sternagel. Certification of termination proofs using CeTA. In *Proc. TPHOLs 2009*, volume 5674 of *LNCS*, pages 452–468, 2009. doi:10.1007/978-3-642-03359-9_31.

8 René Thiemann and Elias Wenninger. A verified efficient implementation of the weighted path order. *Archive of Formal Proofs*, June 2023. https://isa-afp.org/entries/Efficient_Weighted_Path_Order.html, Formal proof development.

9 René Thiemann and Akihisa Yamada. Efficient formalization of simplification orders. Proc. WST 2022, https://sws.cs.ru.nl/pmwiki/uploads/Main/wst2022proceedings.pdf, 2022.

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

Dependency Tuples for Almost-Sure Innermost Termination of Probabilistic Term Rewriting (Short WST Version)

Jan-Christoph Kassing
Jürgen Giesl
LuFG Informatik 2, RWTH Aachen University, Aachen, Germany

Abstract

Dependency pairs are one of the most powerful techniques to analyze termination of term rewrite systems (TRSs) automatically. We adapt the dependency pair framework to the probabilistic setting in order to prove almost-sure innermost termination of probabilistic TRSs. To evaluate its power, we implemented the new framework in our tool AProVE.

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

Keywords and phrases Probabilistic Term Rewriting, Dependency Pairs, Almost-Sure Termination

Related Version *See [6]. Full version, including all proofs*: `https://arxiv.org/abs/2305.11741`

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

1 Introduction

Techniques and tools to analyze innermost termination of TRSs automatically are successfully used for termination analysis of programs in many languages. While there exist several classical orderings for proving termination of TRSs, a *direct* application of these orderings is usually too weak for TRSs that result from actual programs. However, these orderings can be used successfully within the *dependency pair* (DP) framework [1, 5], which allows for modular termination proofs and is one of the most powerful techniques for termination analysis of TRSs that is used in essentially all current termination tools for TRSs.

On the other hand, *probabilistic* programs are used to describe randomized algorithms and probability distributions. To use TRSs also for such programs, *probabilistic term rewrite systems* (PTRSs) were introduced in [4]. A probabilistic program is *almost-surely terminating* (AST) if the probability for termination is 1. While there exist automatic approaches to prove AST for probabilistic programs on numbers, this is the first approach to prove AST for PTRSs. The only other related tool was presented in [2], where orderings based on interpretations were adapted to prove *positive almost-sure termination* of PTRSs, i.e., that the expected number of rewrite steps is finite.

In this paper, we adapt DPs to the probabilistic setting and present the first DP framework for probabilistic term rewriting. We also present an adaption of the technique from [2] for the direct application of polynomial interpretations in order to prove AST of PTRSs.

2 Probabilistic Term Rewriting

We assume familiarity with term rewriting [3] and the DP framework [1, 5]. In this paper, we restrict ourselves to *innermost* rewriting. In contrast to TRSs, a PTRS [2, 4] has finite multi-distributions on the right-hand side of rewrite rules. A finite *multi-distribution* μ on a set $A \neq \varnothing$ is a finite multiset of pairs $(p : a)$, where $0 < p \leq 1$ is a probability and $a \in A$,

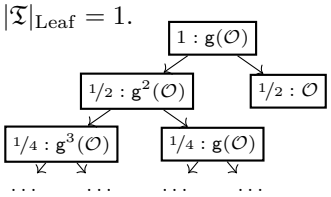
such that $\sum_{(p:a)\in\mu} p = 1$. $\mathrm{FDist}(A)$ is the set of all finite multi-distributions on A. For $\mu \in \mathrm{FDist}(A)$, its *support* is the multiset $\mathrm{Supp}(\mu) = \{a \mid (p:a) \in \mu \text{ for some } p\}$.

▶ **Definition 1** (PTRS). *A* probabilistic rewrite rule $\ell \to \mu \in \mathcal{T}(\Sigma,\mathcal{V}) \times \mathrm{FDist}(\mathcal{T}(\Sigma,\mathcal{V}))$ *is a pair such that* $\ell \notin \mathcal{V}$ *and* $\mathcal{V}(r) \subseteq \mathcal{V}(\ell)$ *for every* $r \in \mathrm{Supp}(\mu)$. *A* probabilistic TRS *(PTRS) is a finite set* $\mathcal{R}$ *of probabilistic rewrite rules. Similar to TRSs, the PTRS* $\mathcal{R}$ *induces a* rewrite relation $\to_{\mathcal{R}} \subseteq \mathcal{T}(\Sigma,\mathcal{V}) \times \mathrm{FDist}(\mathcal{T}(\Sigma,\mathcal{V}))$ *where* $s \to_{\mathcal{R}} \{p_1 : t_1, \dots, p_k : t_k\}$ *if there is a position* π*, a rule* $\ell \to \{p_1 : r_1, \dots, p_k : r_k\} \in \mathcal{R}$*, and a substitution* σ *such that* $s|_\pi = \ell\sigma$ *and* $t_j = s[r_j\sigma]_\pi$ *for all* $1 \le j \le k$*. We call* $s \to_{\mathcal{R}} \mu$ *an* innermost *rewrite step (denoted* $s \xrightarrow{\mathsf{i}}_{\mathcal{R}} \mu$*) if every proper subterm of the used redex* $\ell\sigma$ *is in normal form w.r.t.* $\mathcal{R}$.

▶ **Example 2.** As an example, consider the PTRS $\mathcal{R}_{\mathsf{rw}}$ with the only rule $\mathsf{g}(x) \to \{1/2 : x,\ 1/2 : \mathsf{g}(\mathsf{g}(x))\}$ over a signature with a unary symbol g and a constant symbol $\mathcal{O}$, which corresponds to a symmetric random walk.

To track all possible rewrite sequences (up to non-determinism) with their corresponding probabilities, we *lift* $\to_{\mathcal{R}}$ to *rewrite sequence trees (RST)*. An $\mathcal{R}$-RST is a tree whose nodes v are labeled by pairs $(p_v : t_v)$ of a probability p_v and a term t_v. For each node v with the successors $w_1, \dots, w_k$, the edge relation represents a probabilistic rewrite step, i.e., $t_v \xrightarrow{\mathsf{i}}_{\mathcal{R}} \{\frac{p_{w_1}}{p_v} : t_{w_1}, \dots, \frac{p_{w_k}}{p_v} : t_{w_k}\}$. The root of an RST is always labeled with the probability 1. For an $\mathcal{R}$-RST $\mathfrak{T}$ we define $|\mathfrak{T}|_{\mathrm{Leaf}} = \sum_{v\in\mathrm{Leaf}} p_v$, where Leaf is the set of all leaves, and we say that a PTRS $\mathcal{R}$ is *almost-surely innermost terminating (iAST)* if $|\mathfrak{T}|_{\mathrm{Leaf}} = 1$ holds for all $\mathcal{R}$-RSTs $\mathfrak{T}$. While we have $|\mathfrak{T}|_{\mathrm{Leaf}} = 1$ for every finite RST $\mathfrak{T}$, for infinite RSTs $\mathfrak{T}$ we may have $|\mathfrak{T}|_{\mathrm{Leaf}} < 1$ or even $|\mathfrak{T}|_{\mathrm{Leaf}} = 0$ if $\mathfrak{T}$ has no leaf at all. This notion of AST is the same as the one in [2], where AST is defined using a lifting of $\to_{\mathcal{R}}$ to multisets instead of trees.

▶ **Example 3.** For the infinite $\mathcal{R}_{\mathsf{rw}}$-RST $\mathfrak{T}$ on the side we have $|\mathfrak{T}|_{\mathrm{Leaf}} = 1$.



Thm. 4 introduces a novel technique to prove AST automatically by a direct application of polynomial interpretations. The proof idea is based on [7], but extends it from while-programs on integers to terms. A *polynomial interpretation* Pol is a Σ-algebra with carrier $\mathbb{N}$ mapping every function symbol $f \in \Sigma$ to a polynomial $f_{\mathrm{Pol}} \in \mathbb{N}[\mathcal{V}]$. For a term $t \in \mathcal{T}(\Sigma,\mathcal{V})$, $\mathrm{Pol}(t)$ is the interpretation of t by the Σ-algebra Pol. An inequation $\mathrm{Pol}(t_1) > \mathrm{Pol}(t_2)$ *holds* if it is true for all instantiations of its variables by natural numbers.

▶ **Theorem 4** (Proving AST with Polynomial Interpretations). *Let* $\mathcal{R}$ *be a PTRS and let* $\mathrm{Pol} : \mathcal{T}(\Sigma,\mathcal{V}) \to \mathbb{N}[\mathcal{V}]$ *be a monotonic, multilinear*[1] *polynomial interpretation. If for every rule* $\ell \to \{p_1 : r_1, \dots, p_k : r_k\} \in \mathcal{R}$*,*

1. *there exists a* $1 \le j \le k$ *with* $\mathrm{Pol}(\ell) > \mathrm{Pol}(r_j)$ *and*
2. $\mathrm{Pol}(\ell) \ge \sum_{1\le j\le k} p_j \cdot \mathrm{Pol}(r_j)$,

then $\mathcal{R}$ *is AST.*

▶ **Example 5.** To prove that $\mathcal{R}_{\mathsf{rw}}$ is AST with Thm. 4, we can use the polynomial interpretation that maps $\mathsf{g}(x)$ to $x + 1$ and $\mathcal{O}$ to 0.

[1] Multilinearity means that for all $f \in \Sigma$, all monomials of $f_{\mathrm{Pol}}(x_1, \dots, x_n)$ have the form $c \cdot x_1^{e_1} \cdot \ldots \cdot x_n^{e_n}$ with $c \in \mathbb{N}$ and $e_1, \dots, e_n \in \{0, 1\}$. As in [2], multilinearity ensures "monotonicity" w.r.t. expected values, since multilinearity implies $f_{\mathrm{Pol}}(\dots, \sum_{1\le j\le k} p_j \cdot \mathrm{Pol}(r_j), \dots) = \sum_{1\le j\le k} p_j \cdot \mathrm{Pol}(f(\dots, r_j, \dots))$.

3 Dependency Tuples and Chains for Probabilistic Term Rewriting

As in the non-probabilistic DP framework, we decompose the signature Σ into defined symbols Σ_D and constructor symbols Σ_C. For every $f \in \Sigma_D$, we introduce a fresh *tuple symbol* $f^\#$ of the same arity. $\Sigma^\#$ is the set of tuple symbols and we often write F instead of $\mathsf{f}^\#$. For any term $t = f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$ with $f \in \Sigma_D$, let $t^\# = f^\#(t_1, \ldots, t_n)$. While multiple occurrences of the same subterm $f(\ldots)$ in a right-hand side of a rule can be ignored when defining DPs for TRSs, this is not true when analyzing PTRSs. Hence, as in the adaption of DPs for complexity analysis in [8], we work with *dependency tuples* instead of pairs.

For any $t \in \mathcal{T}(\Sigma, \mathcal{V})$, if $\{t_1, \ldots, t_n\}$ is the *multiset* of all subterms of t with defined root symbols, then we define $dp(t) = \mathsf{c}_n(t_1^\#, \ldots, t_n^\#)$. To make $dp(t)$ unique, we use a total order on positions. Here, we extend Σ_C by a fresh *compound* constructor symbol c_n of arity n for every $n \in \mathbb{N}$. When rewriting a subterm $t_i^\#$ of $\mathsf{c}_n(t_1^\#, \ldots, t_n^\#)$ with a dependency tuple, one obtains terms with nested compound symbols. To flatten nested compound symbols and to abstract from the order of their arguments, we always *normalize* terms implicitly. So for example, $\mathsf{c}_3(x, x, y)$ is a normalization of $\mathsf{c}_2(\mathsf{c}_1(x), \mathsf{c}_2(x, y))$.

Instead of considering a rule $\ell \to \{p_1 : r_1, \ldots, p_k : r_k\}$ from $\mathcal{R}$ and its corresponding dependency tuple $\ell^\# \to \{p_1 : dp(r_1), \ldots, p_k : dp(r_k)\}$ separately, we couple them together to $\langle \ell^\#, \ell \rangle \to \{p_1 : \langle dp(r_1), r_1 \rangle, \ldots, p_k : \langle dp(r_k), r_k \rangle\}$. So in the "second component", we can access the original rewrite rule used to create the dependency tuple. The resulting type of rewrite system is called a *probabilistic pair term rewrite system (PPTRS)*. Our new DP framework operates on *DP problems* $(\mathcal{P}, \mathcal{S})$, where $\mathcal{P}$ is a PPTRS and $\mathcal{S}$ is a PTRS.

▶ **Definition 6** (Coupled Dependency Tuple). *Let $\mathcal{R}$ be a PTRS. For every $\ell \to \{p_1 : r_1, \ldots, p_k : r_k\} \in \mathcal{R}$, its* (coupled) dependency tuple *(*DT*) is $\langle \ell^\#, \ell \rangle \to \{p_1 : \langle dp(r_1), r_1 \rangle, \ldots, p_k : \langle dp(r_k), r_k \rangle\}$. The set of all coupled dependency tuples of $\mathcal{R}$ is denoted by $\mathcal{DT}(\mathcal{R})$.*

A PPTRS can rewrite a tuple term $t^\#$ and simultaneously use the original rewrite rule (that was used to create the dependency tuple) in order to rewrite all "copies" t of $t^\#$. We also introduce an analogous new rewrite relation for PTRSs, where we can apply the same rule simultaneously to the same subterms in a single rewrite step.

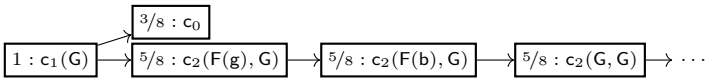
▶ **Example 7.** Consider a PTRS $\mathcal{R}_\mathsf{a}$ that (also) contains the rule $\mathsf{a} \to \{1/2 : \mathsf{s}(\mathsf{b}_1), 1/2 : \mathsf{s}(\mathsf{b}_2)\}$ and has the defined symbols $\Sigma_D = \{\mathsf{f}, \mathsf{a}, \mathsf{b}_1, \mathsf{b}_2\}$. The corresponding DT is $\langle \mathsf{A}, \mathsf{a} \rangle \to \{1/2 : \langle \mathsf{c}_1(\mathsf{B}_1), \mathsf{s}(\mathsf{b}_1) \rangle, 1/2 : \langle \mathsf{c}_1(\mathsf{B}_2), \mathsf{s}(\mathsf{b}_2) \rangle\}$. To obtain a sound termination criterion, it must be possible to mimic every rewrite step by a corresponding chain of DTs. With our notion of PTRSs, we can indeed mimic the rewrite step $\mathsf{f}(\mathsf{a}) \to_{\mathcal{R}_\mathsf{a}} \{1/2 : \mathsf{f}(\mathsf{s}(\mathsf{b}_1)), 1/2 : \mathsf{f}(\mathsf{s}(\mathsf{b}_2))\}$ using a single step with the PPTRS $\mathcal{DT}(\mathcal{R}_\mathsf{a})$: In the corresponding chain we start with $\mathsf{c}_2(\mathsf{F}(\mathsf{a}), \mathsf{A})$ and rewrite it with $\mathcal{DT}(\mathcal{R}_\mathsf{a})$ to $\{1/2 : \mathsf{c}_2(\mathsf{F}(\mathsf{s}(\mathsf{b}_1)), \mathsf{B}_1), 1/2 : \mathsf{c}_2(\mathsf{F}(\mathsf{s}(\mathsf{b}_2)), \mathsf{B}_2)\}$. So here we have to access the original rewrite rule to rewrite the "copy" a of A to $\mathsf{s}(\mathsf{b}_1)$ and $\mathsf{s}(\mathsf{b}_2)$, respectively.

The $(\mathcal{P}, \mathcal{S})$-chains in the probabilistic setting are now defined as *chain trees (CTs)*, where the edges either describe steps with the PPTRS $\mathcal{P}$ or with the PTRS $\mathcal{S}$. Regarding the paths in this tree allows us to adapt the idea of chains, i.e., that one uses only finitely many $\mathcal{S}$-steps before the next step with a DT from $\mathcal{P}$. We define $|\mathfrak{T}|_{\text{Leaf}}$ and iAST for CTs as for RSTs. With this new type of DTs and CTs, we can now mimic every possible RST using a CT. More precisely, from every RST we can create a CT with the same tree structure, and for each node v we have $dp(t_v^{RST}) = t_v^{CT}$, i.e., the term t_v^{CT} at node v in the CT corresponds to the dp transformation of the term t_v^{RST} in the RST (except for some irrelevant normal forms). Hence, we get an analogous chain criterion to the non-probabilistic setting.

▶ **Theorem 8** (Chain Criterion). *A PTRS $\mathcal{R}$ is iAST if $(\mathcal{DT}(\mathcal{R}), \mathcal{R})$ is iAST.*

In contrast to the non-probabilistic case, our chain criterion is *sound* but not *complete* (i.e., we do not have "iff" in Thm. 8). However, we also developed a refinement where our chain criterion is made complete by also storing the positions of the defined symbols in $dp(r)$.

▶ **Example 9.** The PTRS $\mathcal{R}_{\mathsf{incompl}}$ with the three rules $\mathsf{g} \to \{5/8 : \mathsf{f}(\mathsf{g}), 3/8 : \mathsf{stop}\}$, $\mathsf{g} \to \{1 : \mathsf{b}\}$, and $\mathsf{f}(\mathsf{b}) \to \{1 : \mathsf{g}\}$ shows that the chain criterion of Thm. 8 is not complete. AProVE can prove AST via Thm. 4 with the polynomial interpretation that maps $\mathsf{f}(x)$ to $x+2$, g to 4, b to 3, and stop to 0. On the other hand, the dependency tuples for this PTRS are $\langle \mathsf{G}, \mathsf{g}\rangle \to \{5/8 : \langle \mathsf{c}_2(\mathsf{F}(\mathsf{g}), \mathsf{G}), \mathsf{f}(\mathsf{g})\rangle, 3/8 : \langle \mathsf{c}_0, \mathsf{stop}\rangle\}$, $\langle \mathsf{G}, \mathsf{g}\rangle \to \{1 : \langle \mathsf{c}_0, \mathsf{b}\rangle\}$, and $\langle \mathsf{F}(\mathsf{b}), \mathsf{f}(\mathsf{b})\rangle \to \{1 : \langle \mathsf{c}_1(\mathsf{G}), \mathsf{g}\rangle\}$. The DP problem $(\mathcal{DT}(\mathcal{R}_{\mathsf{incompl}}), \mathcal{R}_{\mathsf{incompl}})$ is not iAST. To see this, consider the following chain tree:



Here, we first use the first dependency tuple, then the rule $\mathsf{g} \to \{1 : \mathsf{b}\}$, and finally the third dependency tuple. We essentially end up with a biased random walk where the number of gs is increased with probability $5/8$ and decreased with probability $3/8$. Hence, $(\mathcal{DT}(\mathcal{R}_{\mathsf{incompl}}), \mathcal{R}_{\mathsf{incompl}})$ is not iAST. The problem here is that the terms G and $\mathsf{F}(\mathsf{g})$ can both be rewritten to G. When creating the dependency tuples, we lose the information that the g inside the term $\mathsf{F}(\mathsf{g})$ corresponds to the second argument G of the compound symbol c_2. In order to obtain a complete chain criterion, one has to extend dependency tuples by positions. Then the second argument G of the compound symbol c_2 would be augmented by the position of g in the right-hand side of the first rule, because this rule was used to generate the dependency tuple. When rewriting g by a rule from $\mathcal{R}_{\mathsf{incompl}}$, then terms like G that belong to the same (or a lower) position have to be removed.

Our notion of DTs and chain trees is only suitable for *innermost* evaluation. To see this, consider the PTRSs $\mathcal{R}_1$ and $\mathcal{R}_2$ which both contain $\mathsf{g} \to \{1/2 : \mathcal{O}, 1/2 : \mathsf{h}(\mathsf{g})\}$, but in addition $\mathcal{R}_1$ has the rule $\mathsf{h}(x) \to \{1 : \mathsf{f}(x, x)\}$ and $\mathcal{R}_2$ has the rule $\mathsf{h}(x) \to \{1 : \mathsf{f}(x, x, x)\}$. Note that when considering full rewriting, $\mathcal{R}_1$ is AST while $\mathcal{R}_2$ is not. In contrast, both $\mathcal{R}_1$ and $\mathcal{R}_2$ are iAST, since the innermost evaluation strategy prevents the application of the h-rule to terms containing g. Our DP framework handles $\mathcal{R}_1$ and $\mathcal{R}_2$ in the same way, as both have the same DT $\langle \mathsf{G}, \mathsf{g}\rangle \to \{1/2 : \langle \mathsf{c}_0, \mathcal{O}\rangle, 1/2 : \langle \mathsf{c}_2(\mathsf{H}(\mathsf{g}), \mathsf{G}), \mathsf{h}(\mathsf{g})\rangle\}$ and a DT $\langle \mathsf{H}(x), \mathsf{h}(x)\rangle \to \{1 : \langle \mathsf{c}_0, \mathsf{f}(\ldots)\rangle\}$. Even if we allowed the application of the second DT to terms of the form $\mathsf{H}(\mathsf{g})$, we would still obtain $|\mathfrak{T}|_{\mathrm{Leaf}} = 1$ for every chain tree $\mathfrak{T}$. So a DP framework to analyze "full" instead of innermost AST would be considerably more involved.

4 The Probabilistic DP Framework

Now we introduce the probabilistic DP framework which keeps the core ideas of the non-probabilistic framework. So instead of applying one ordering for a PTRS directly as in Thm. 4, we want to benefit from modularity. A *DP processor* Proc is of the form $\mathrm{Proc}(\mathcal{P}, \mathcal{S}) = \{(\mathcal{P}_1, \mathcal{S}_1), \ldots, (\mathcal{P}_n, \mathcal{S}_n)\}$, where $\mathcal{P}, \mathcal{P}_1, \ldots, \mathcal{P}_n$ are PPTRSs and $\mathcal{S}, \mathcal{S}_1, \ldots, \mathcal{S}_n$ are PTRSs. A processor Proc is *sound* if $(\mathcal{P}, \mathcal{S})$ is iAST whenever $(\mathcal{P}_i, \mathcal{S}_i)$ is iAST for all $1 \le i \le n$. It is *complete* if $(\mathcal{P}_i, \mathcal{S}_i)$ is iAST for all $1 \le i \le n$ whenever $(\mathcal{P}, \mathcal{S})$ is iAST.

The (innermost) $(\mathcal{P}, \mathcal{S})$-*dependency graph* indicates which DTs from $\mathcal{P}$ can rewrite to each other using the PTRS $\mathcal{S}$. The possibility of rewriting with $\mathcal{S}$ is not related to the probabilities. Thus, for the dependency graph, we can use the *non-probabilistic variant* $\mathrm{np}(\mathcal{S}) = \{\ell \to r_j \mid \ell \to \{p_1 : r_1, \ldots, p_k : r_k\} \in \mathcal{S}, 1 \le j \le k\}$.

▶ **Definition 10** (Dependency Graph). *The node set of the* $(\mathcal{P},\mathcal{S})$-dependency graph *is* $\mathcal{P}$ *and there is an edge from* $\langle \ell_1^\#, \ell_1\rangle \to \{p_1 : \langle d_1, r_1\rangle, \dots, p_k : \langle d_k, r_k\rangle\}$ *to* $\langle \ell_2^\#, \ell_2\rangle \to \dots$ *if there are substitutions* σ_1, σ_2*, and a* $t^\#$ *occurring in* d_j *for some* $1 \le j \le k$*, such that* $t^\#\sigma_1 \xrightarrow{\mathsf{i}}{}^*_{\mathrm{np}(\mathcal{S})} \ell_2^\#\sigma_2$ *and both* $\ell_1^\#\sigma_1$ *and* $\ell_2^\#\sigma_2$ *are in normal form w.r.t.* $\mathcal{S}$*.*

In the non-probabilistic DP framework, every step from one DP to the next in a chain corresponds to an edge in the dependency graph. Similarly, in the probabilistic setting, for every path from a node where a step with $\mathcal{P}$ is used to the next such node in a $(\mathcal{P},\mathcal{S})$-CT, there is a corresponding edge in the $(\mathcal{P},\mathcal{S})$-dependency graph. Since every infinite path in a CT contains infinitely many such nodes, when tracking the arguments of the compound symbols, every such path traverses a cycle of the dependency graph infinitely often. Thus, it again suffices to consider the SCCs of the dependency graph separately. To automate the following processor, the same over-approximation techniques as for the non-probabilistic dependency graph can be used.

▶ **Theorem 11** (Probabilistic Dependency Graph Processor). *For the SCCs* $\mathcal{P}_1, \dots, \mathcal{P}_n$ *of the* $(\mathcal{P},\mathcal{S})$*-dependency graph,* $\mathrm{Proc}_{\mathtt{DG}}(\mathcal{P},\mathcal{S}) = \{(\mathcal{P}_1,\mathcal{S}), \dots, (\mathcal{P}_n,\mathcal{S})\}$ *is sound and complete.*

▶ **Example 12.** As an example, consider the following PTRS $\mathcal{R}_{\mathsf{div}}$ which adapts a well-known example from [1] to the probabilistic setting.

$$\begin{aligned} \mathsf{minus}(x,\mathcal{O}) &\to \{1 : x\} & \mathsf{div}(\mathcal{O},\mathsf{s}(y)) &\to \{1 : \mathcal{O}\}\\ \mathsf{minus}(\mathsf{s}(x),\mathsf{s}(y)) &\to \{1 : \mathsf{minus}(x,y)\} & \mathsf{div}(\mathsf{s}(x),\mathsf{s}(y)) &\to \{1/2 : \mathsf{div}(\mathsf{s}(x),\mathsf{s}(y)), 1/2 : \mathsf{s}(\mathsf{div}(\mathsf{minus}(x,y),\mathsf{s}(y)))\}\end{aligned}$$

We get $\mathcal{DT}(\mathcal{R}_{\mathsf{div}}) = \{(1), \dots, (4)\}$:

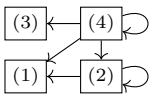
$$\langle \mathsf{M}(x,\mathcal{O}), \mathsf{minus}(x,\mathcal{O})\rangle \to \{1 : \langle \mathsf{c}_0,\ x\rangle\} \tag{1}$$

$$\langle \mathsf{M}(\mathsf{s}(x),\mathsf{s}(y)), \mathsf{minus}(\mathsf{s}(x),\mathsf{s}(y))\rangle \to \{1 : \langle \mathsf{c}_1(\mathsf{M}(x,y)),\ \mathsf{minus}(x,y)\rangle\} \tag{2}$$

$$\langle \mathsf{D}(\mathcal{O},\mathsf{s}(y)), \mathsf{div}(\mathcal{O},\mathsf{s}(y))\rangle \to \{1 : \langle \mathsf{c}_0,\ \mathcal{O}\rangle\} \tag{3}$$

$$\langle \mathsf{D}(\mathsf{s}(x),\mathsf{s}(y)), \mathsf{div}(\mathsf{s}(x),\mathsf{s}(y))\rangle \to \{1/2 : \langle \mathsf{c}_1(\mathsf{D}(\mathsf{s}(x),\mathsf{s}(y))),\ \mathsf{div}(\mathsf{s}(x),\mathsf{s}(y))\rangle, \ 1/2 : \langle \mathsf{c}_2(\mathsf{D}(\mathsf{minus}(x,y),\mathsf{s}(y)), \mathsf{M}(x,y)),\ \mathsf{s}(\mathsf{div}(\mathsf{minus}(x,y),\mathsf{s}(y)))\rangle\} \tag{4}$$

The $(\mathcal{DT}(\mathcal{R}_{\mathsf{div}}),\mathcal{R}_{\mathsf{div}})$-dependency graph is shown on the side. So we obtain $\mathrm{Proc}_{\mathtt{DG}}(\mathcal{DT}(\mathcal{R}_{\mathsf{div}}),\mathcal{R}_{\mathsf{div}}) = \{(\{(2)\},\mathcal{R}_{\mathsf{div}}), (\{(4)\},\mathcal{R}_{\mathsf{div}})\}$.



For the *reduction pair processor*, we use analogous constraints as in our new criterion for the direct application of polynomial interpretations to PTRSs (Thm. 4), but adapted to DP problems $(\mathcal{P},\mathcal{S})$. Moreover, as in the original reduction pair processor, the polynomials only have to be weakly monotonic. For every rule in $\mathcal{S}$ or in the first components of $\mathcal{P}$, we require that the expected value is weakly decreasing. The reduction pair processor then removes those DTs $\langle \ell^\#, \ell\rangle \to \{p_1 : \langle d_1, r_1\rangle, \dots, p_k : \langle d_k, r_k\rangle\}$ from $\mathcal{P}$ where in addition there is at least one term d_j that is strictly decreasing. Moreover, we can also rewrite with the original rule $\ell \to \{p_1 : r_1, \dots, p_k : r_k\}$ from the second component of the DT, provided that it is also contained in $\mathcal{S}$. Therefore, to remove the dependency tuple, we also have to require that the rule $\ell \to r_j$ is weakly decreasing.

▶ **Theorem 13** (Probabilistic Reduction Pair Processor). *Let* $\mathrm{Pol} : \mathcal{T}(\Sigma \uplus \Sigma^\#, \mathcal{V}) \to \mathbb{N}[\mathcal{V}]$ *be a weakly monotonic, multilinear polynomial interpretation with* $\mathsf{c}_{n\,\mathrm{Pol}}(x_1, \dots, x_n) = x_1 + \dots + x_n$*. Let* $\mathcal{P} = \mathcal{P}_{\ge} \uplus \mathcal{P}_{>}$ *with* $\mathcal{P}_{>} \neq \varnothing$ *such that:*

1. *For every* $\ell \to \{p_1 : r_1, \dots, p_k : r_k\} \in \mathcal{S}$*, we have* $\mathrm{Pol}(\ell) \ge \sum_{1\le j\le k} p_j \cdot \mathrm{Pol}(r_j)$*.*
2. *For every* $\langle \ell^\#, \ell\rangle \to \{p_1 : \langle d_1, r_1\rangle, \dots, p_k : \langle d_k, r_k\rangle\} \in \mathcal{P}$*, we have* $\mathrm{Pol}(\ell^\#) \ge \sum_{1\le j\le k} p_j \cdot \mathrm{Pol}(d_j)$*.*

3. *For every* $\langle \ell^{\#}, \ell\rangle \to \{p_1 : \langle d_1, r_1\rangle, \dots, p_k : \langle d_k, r_k\rangle\} \in \mathcal{P}_>$,
   there exists a $1 \le j \le k$ *with* $\mathrm{Pol}(\ell^{\#}) > \mathrm{Pol}(d_j)$.
   If $\ell \to \{p_1 : r_1, \dots, p_k : r_k\} \in \mathcal{S}$, *then we additionally have* $\mathrm{Pol}(\ell) \ge \mathrm{Pol}(r_j)$.

Then $\mathrm{Proc}_{\mathtt{RP}}(\mathcal{P}, \mathcal{S}) = \{(\mathcal{P}_{\ge}, \mathcal{S})\}$ *is sound and complete.*

▶ **Example 14.** The constraints of the reduction pair processor for the two DP problems from Ex. 12 are satisfied by the polynomial interpretation which maps $\mathcal{O}$ to 0, $\mathsf{s}(x)$ to $2 \cdot x + 1$, and the other function symbols to the projection on their first arguments. This results in DP problems of the form $(\varnothing, \dots)$ and subsequently, $\mathrm{Proc}_{\mathtt{DG}}(\varnothing, \dots)$ yields $\varnothing$. By the soundness of all processors, this proves that $\mathcal{R}_{\mathsf{div}}$ is iAST. In contrast, similar to the non-probabilistic setting, a direct application of polynomial interpretations via Thm. 4 fails for this example.

In addition to these two processors, we also have processors that can remove probabilistic rewrite rules from $\mathcal{S}$ or even remove terms from the first components of the right-hand sides of DTs in $\mathcal{P}$. Furthermore, if we eventually end up with dependency tuples and rules that only have the trivial probability 1, we can transform the problem into a non-probabilistic DP problem and use the non-probabilistic framework.

5 Evaluation

We implemented our contributions in our termination prover AProVE, which yields the first tool to prove almost-sure innermost termination of PTRSs on arbitrary data structures (including PTRSs that are not PAST). In our experiments, we compared the direct application of polynomials for proving AST (via our new Thm. 4) with the probabilistic DP framework. We evaluated AProVE on a collection of 67 PTRSs which includes many typical probabilistic algorithms. For example, it contains a PTRS $\mathcal{R}_{\mathsf{qs}}$ for probabilistic quicksort that has rules for choosing a pivot element that are only AST but not terminating, see [6]. Using the probabilistic DP framework, AProVE can prove iAST of $\mathcal{R}_{\mathsf{qs}}$ and many other typical programs.

61 of the 67 examples in our collection are iAST and AProVE can prove iAST for 53 (87%) of them. Here, the DP framework proves iAST for 51 examples and the direct application of polynomial interpretations via Thm. 4 succeeds for 27 examples. (In contrast, proving PAST via the direct application of polynomial interpretations as in [2] only works for 22 examples.) The average runtime of AProVE per example was 2.88 s (where no example took longer than 8 s). So our experiments indicate that the power of the DP framework can now also be used for probabilistic TRSs.

For details on our experiments and for instructions on how to run our implementation in AProVE via its *web interface* or locally, we refer to `https://aprove-developers.github.io/ProbabilisticTermRewriting/`.

References

1 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sc.*, 236(1-2):133–178, 2000.

2 M. Avanzini, U. Dal Lago, and A. Yamada. On probabilistic term rewriting. *Sci. Comput. Program.*, 185, 2020.

3 F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

4 O. Bournez and F. Garnier. Proving positive almost-sure termination. In *Proc. RTA '05*, LNCS 3467, pages 323–337, 2005.

5 J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *J. Autom. Reason.*, 37(3):155–203, 2006.

6 J.-C. Kassing and J. Giesl. Proving almost-sure innermost termination of probabilistic term rewriting using dependency pairs. In *Proc. CADE '23*, LNCS, 2023. To appear. Full version appeared in *CoRR*, abs/2305.11741.

7 A. McIver, C. Morgan, B. L. Kaminski, and J.-P. Katoen. A new proof rule for almost-sure termination. *Proc. ACM Program. Lang.*, 2(POPL), 2018.

8 L. Noschinski, F. Emmes, and J. Giesl. Analyzing innermost runtime complexity of term rewriting by dependency pairs. *J. Autom. Reason.*, 51:27–56, 2013.

Automated Termination Proofs for C Programs with Lists (Short WST Version)

Jera Hensel
Jürgen Giesl
LuFG Informatik 2, RWTH Aachen University, Aachen, Germany

Abstract

There are many techniques and tools for termination of C programs, but up to now they were not very powerful for termination proofs of programs whose termination depends on recursive data structures like lists. We present the first approach that extends powerful techniques for termination analysis of C programs (with memory allocation and explicit pointer arithmetic) to lists.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Program analysis; Software and its engineering → Data types and structures

Keywords and phrases Termination Analysis, C Programs, Lists, Symbolic Execution

Related Version *See [11]. Full version, including all proofs*: https://arxiv.org/abs/2305.12159

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

1 Introduction

In [8, 10, 16], we introduced an approach for termination analysis of C that also handles programs whose termination relies on the relation between allocated memory and the data stored at such addresses. This approach is implemented in our tool AProVE [9]. Instead of analyzing C directly, AProVE compiles the program to LLVM. Then it constructs a (finite) symbolic execution graph (SEG) such that every program run corresponds to a path through the SEG. AProVE proves memory safety during the construction of the SEG to ensure absence of undefined behavior (which would also allow non-termination). Afterwards, the SEG is transformed into an integer transition system (ITS) such that all paths through the SEG (and hence, the C program) are terminating if the ITS is terminating. To analyze termination of ITSs, AProVE applies standard techniques and tools. However, like other termination tools for C, up to now AProVE supported dynamic data structures only in a very restricted way.

In the program on the side, `nondet_uint` returns a random unsigned integer. The `for` loop creates a list of `n` random numbers if `n` > 0 and the `while` loop traverses this list via pointer arithmetic. We introduce a novel technique which can analyze termination of such C programs on lists, i.e., it can both handle the access by `curr->next` (when initializing the list) and by pointer arithmetic (when traversing the list).

```
struct list {
  unsigned int value;
  struct list* next;   };

int main() {
  // initialize length
  unsigned int n = nondet_uint();
  // initialize list of length n
  struct list* tail = NULL;
  struct list* curr;
  for (unsigned int k = 0; k < n; k++) {
    curr = malloc(sizeof(struct list));
    curr->value = nondet_uint();
    curr->next = tail;
    tail = curr;                        }
  // traverse list
  struct list* ptr = tail;
  while(ptr != NULL) {
    ptr = *((struct list**)((void*)ptr +
          offsetof(struct list, next)));}}
```

Our technique infers *list invariants* via symbolic execution. These invariants express all properties that are crucial for memory safety and termination. In our example, the list invariant contains the information that dereferencing the `next` pointer in the `while` loop is safe and that one finally reaches the null pointer.

To ease the presentation, in this paper we treat integer types as unbounded. Moreover, we assume that a program consists of a single non-recursive function and that values may be stored at any address. Our approach can also deal with bitvectors, data alignments, and programs with arbitrary many (possibly recursive) functions, see [8, 10, 16] for details. However, so far only lists without sharing can be handled by our new technique. Extending it to more general recursive data structures is one of the main challenges for future work.

2 Abstract States for Symbolic Execution

The LLVM code for the `for` loop is given on the side. To ease readability, we omitted instructions and keywords that are irrelevant for our presentation, renamed variables, and wrote `list` instead of `struct.list`. The code consists of several *basic blocks* including `cmpF` and `bodyF` (for the comparison and the body of the `for` loop).

```
list = type { i32, list* }

define i32 @main() { ...
cmpF:
  k < n
  0: k = load i32, i32* k_ad
  1: kltn = icmp ult i32 k, n
  2: br i1 kltn, label bodyF, label initPtr
bodyF:
  curr = malloc(sizeof(struct list));
  0: mem = call i8* @malloc(i64 16)
  1: curr = bitcast i8* mem to list*
  curr->value = nondet_uint();
  2: nondet = call i32 @nondet_uint()
  3: curr_val = getelementptr list,
               list* curr, i32 0, i32 0
  4: store i32 nondet, i32* curr_val
  curr->next = tail;
  5: tail = load list*, list** tail_ptr
  6: curr_next = getelementptr list,
                 list* curr, i32 0, i32 1
  7: store list* tail, list** curr_next
  tail = curr;
  8: store list* curr, list** tail_ptr
  k++
  9: kinc = add i32 k, 1
  10:store i32 kinc, i32* k_ad
  11:br label cmpF     ...                  }
```

We now recapitulate the *abstract states* of [16] used for symbolic execution and extend them by a component LI for list invariants. The first component is a *program position* $(\mathtt{b}, i)$, indicating that instruction i of block `b` is executed next.

The second component is a partial injective function $LV\colon \mathcal{V}_\mathcal{P} \rightharpoonup \mathcal{V}_{sym}$, which maps *local program variables* $\mathcal{V}_\mathcal{P}$ of the program $\mathcal{P}$ to an infinite set $\mathcal{V}_{sym}$ of symbolic variables with $\mathcal{V}_{sym} \cap \mathcal{V}_\mathcal{P} = \varnothing$.

The third component is a set AL of allocations $[\![v_1, v_2]\!]$ with $v_1, v_2 \in \mathcal{V}_{sym}$, indicating that $v_1 \leq v_2$ and all addresses between v_1 and v_2 are allocated.

The fourth and fifth components PT and LI model the memory. PT contains "points-to" entries $v_1 \hookrightarrow_{\mathtt{ty}} v_2$ where $v_1, v_2 \in \mathcal{V}_{sym}$ and `ty` is an LLVM type, meaning that the address v_1 of type `ty` points to v_2. The set LI of *list invariants* (which is new compared to [16]) contains invariants $v_{ad} \overset{v_\ell}{\hookrightarrow}_{\mathtt{ty}} [(\mathit{off}_i : \mathtt{ty}_i : v_i..\hat{v}_i)]_{i=1}^{m}$ where $m \in \mathbb{N}_{>0}$, $v_{ad}, v_\ell, v_i, \hat{v}_i \in \mathcal{V}_{sym}$, $\mathit{off}_i \in \mathbb{N}$ for all $1 \leq i \leq m$, `ty` and $\mathtt{ty}_i$ are LLVM types for all $1 \leq i \leq m$, and there is exactly one "recursive field" $1 \leq j \leq m$ such that $\mathtt{ty}_j = \mathtt{ty*}$. Such an invariant represents a `struct ty` with m fields that corresponds to a recursively defined list of length v_ℓ. Here, v_{ad} points to the first list element, the i-th field starts at address $v_{ad} + \mathit{off}_i$ and has type $\mathtt{ty}_i$, and the values of the i-th fields of the first and last list element are v_i and $\hat{v}_i$, respectively. For example, the list invariant (1) represents all lists of length x_ℓ and type `list` whose elements store a 32-bit integer in their first field and the pointer to the next element in their second field with offset 8. The first list element starts at address $x_{\mathtt{mem}}$, the second starts at address $x_{\mathtt{next}}$, and the last element contains the null pointer. Moreover, the first element stores the integer value $x_{\mathtt{nd}}$ and the last list element stores the integer $\hat{x}_{\mathtt{nd}}$.

$$x_{\mathtt{mem}} \overset{x_\ell}{\hookrightarrow}_{\mathtt{list}} [(0 : \mathtt{i32} : x_{\mathtt{nd}}..\hat{x}_{\mathtt{nd}}), (8 : \mathtt{list*} : x_{\mathtt{next}}..0)] \tag{1}$$

The last component is a *knowledge base KB* of formulas that express arithmetic properties of $\mathcal{V}_{sym}$. A special state *ERR* is reached if we cannot prove absence of undefined behavior.

As an example, the abstract state (2) represents concrete states at the beginning of the block `cmpF`, where the program variable `curr` is assigned the symbolic variable $x_{\mathtt{mem}}$, the

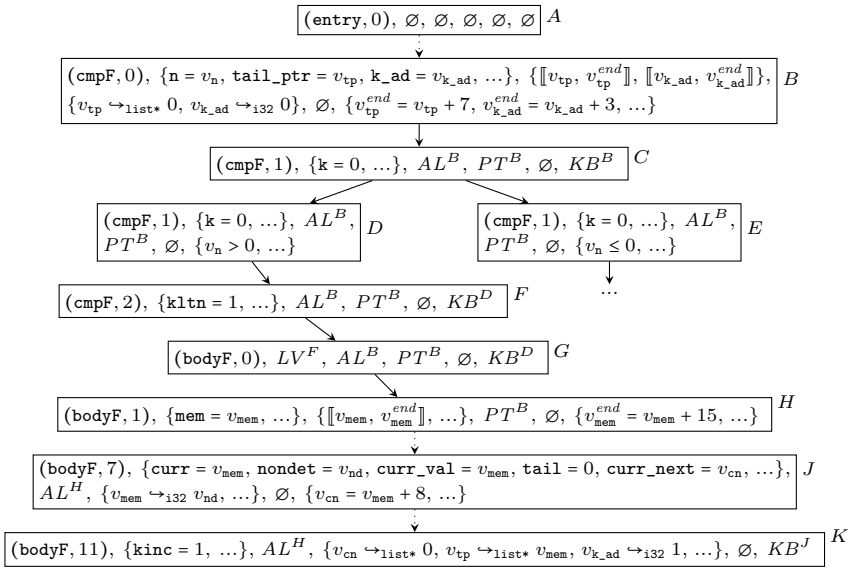


Figure 1 SEG for the First Iteration of the `for` Loop

allocation $[\![x_{\texttt{k\_ad}}, x_{\texttt{k\_ad}}^{end}]\!]$ consisting of 4 bytes stores the value $x_{\texttt{kinc}}$, and $x_{\texttt{mem}}$ points to the first element of a list of length x_ℓ (equal to $x_{\texttt{kinc}}$) that satisfies the list invariant (1). (This state will later be obtained during the symbolic execution, see State O in Fig. 3.)

$$(\texttt{cmpF}, 0),\ \{\texttt{curr} = x_{\texttt{mem}}, \texttt{kinc} = x_{\texttt{kinc}}, \ldots\},\ \{[\![x_{\texttt{k\_ad}}, x_{\texttt{k\_ad}}^{end}]\!], \ldots\},\ \{x_{\texttt{k\_ad}} \hookrightarrow_{\texttt{i32}} x_{\texttt{kinc}}, \ldots\},\ \{x_{\texttt{mem}} \overset{x_\ell}{\hookrightarrow}_{\texttt{list}} [(0 : \texttt{i32} : x_{\texttt{nd}}..\hat{x}_{\texttt{nd}}), (8 : \texttt{list*} : x_{\texttt{next}}..0)]\},\ \{x_{\texttt{k\_ad}}^{end} = x_{\texttt{k\_ad}} + 3,\ x_\ell = x_{\texttt{kinc}}, \ldots\} \tag{2}$$

3 Symbolic Execution with List Invariants

Symbolic execution starts with a state A at the first instruction of the first block (`entry` in our example), see Fig. 1. Dotted arrows indicate that we omitted some steps. For every state, we perform symbolic execution by applying the corresponding inference rule to compute its successor(s) and repeat this until all paths end in return states. Such an SEG is *complete*.

In our example, the `entry` block comprises the first three lines of the C program and the initialization of the pointer to the loop variable `k`: First, a non-deterministic integer is assigned to `n`, i.e., $(\texttt{n} = v_{\texttt{n}}) \in LV^B$, where $v_{\texttt{n}}$ is not restricted. Moreover, memory for the pointers `tail_ptr` and `k_ad` is allocated and they point to `tail = NULL` and `k = 0`, respectively ($\texttt{tail\_ptr} = v_{\texttt{tp}}$ and $\texttt{k\_ad} = v_{\texttt{k\_ad}}$ with $(v_{\texttt{tp}} \hookrightarrow_{\texttt{list*}} 0), (v_{\texttt{k\_ad}} \hookrightarrow_{\texttt{i32}} 0) \in PT^B$). For simplicity, in Fig. 1 we use concrete values directly instead of introducing fresh variables for them.

State C results from B by evaluating the `load` instruction at $(\texttt{cmpF}, 0)$, where the value `0` stored at $v_{\texttt{k\_ad}}$ is loaded to the program variable `k`. The next instruction is an integer comparison which checks whether the unsigned value of `k` is less than the one of `n`. If we cannot decide a comparison, we refine the state into two successor states like D and E (with $(v_{\texttt{n}} > 0) \in KB^D$ and $(v_{\texttt{n}} \leq 0) \in KB^E$). Evaluating D yields $\texttt{kltn} = 1$ in F. Therefore, the branch instruction leads to the block `bodyF` in State G. State E is evaluated to a state with $\texttt{kltn} = 0$. This path branches to the block `initPtr` and terminates as `tail_ptr` points to an empty list.

The instruction at $(\texttt{bodyF}, 0)$ allocates 16 bytes of memory starting at $v_{\texttt{mem}}$ in State H. Next, the pointer to the allocation is cast from `i8*` to `list*` and assigned to `curr`. Now the allocated area can be treated as a list element. Then `nondet_uint()` assigns a 32-bit integer to `nondet`. The `getelementptr` instruction computes the address of the integer field of the

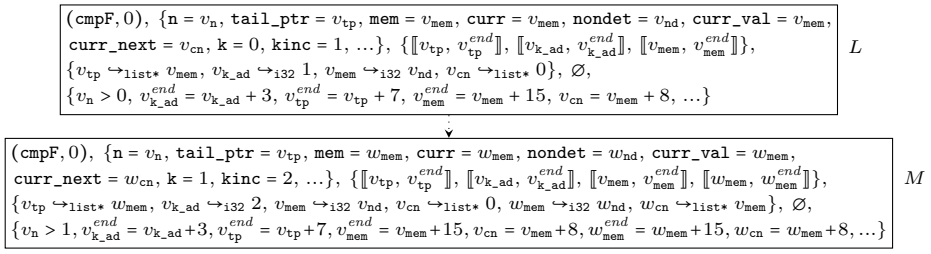
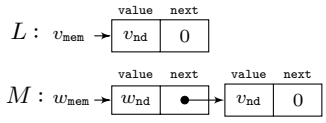


Figure 2 Second Iteration of the `for` Loop

list element by indexing this field (the second `i32 0`) based on the start address (`curr`). Since the address of this integer value coincides with the start address of the list element, this instruction assigns $v_{\texttt{mem}}$ to `curr_val`. Afterwards, the value of `nondet` is stored at `curr_val` ($v_{\texttt{mem}} \hookrightarrow_{\texttt{i32}} v_{\texttt{nd}}$), the value `0` stored at $v_{\texttt{tp}}$ is loaded to `tail`, and a second `getelementptr` instruction computes the address of the recursive field of the current list element ($v_{\texttt{cn}} = v_{\texttt{mem}}+8$) and assigns it to `curr_next`, leading to State J. In the path to K, the values of `tail` and `curr` are stored at `curr_next` and `tail_ptr`, respectively ($v_{\texttt{cn}} \hookrightarrow_{\texttt{list*}} 0$, $v_{\texttt{tp}} \hookrightarrow_{\texttt{list*}} v_{\texttt{mem}}$). Finally, the incremented value of `k` is assigned to `kinc` and stored at `k_ad` ($v_{\texttt{k\_ad}} \hookrightarrow_{\texttt{i32}} 1$).

To ensure a finite graph construction, when a program position is reached for the second time, we try to merge the states at this position to a *generalized* state. However, this is only meaningful if the domains of the LV functions of the two states coincide (i.e., the states consider the same program variables). Therefore, after the branch from the loop body back to `cmpF` (State L in Fig. 2), we evaluate the loop a second time and reach M. Here, a second list element with value $w_{\texttt{nd}}$ and a `next` pointer $w_{\texttt{cn}}$ pointing to $v_{\texttt{mem}}$ has been stored at a new allocation $[\![w_{\texttt{mem}}, w_{\texttt{mem}}^{end}]\!]$. Now, `curr` points to the new element and `k` has been incremented again, so `k_ad` points to 2.



We want to merge L and M to a more general state O that represents all states which are represented by L or M. The challenging part during generalization is to find loop invariants that provide sufficient information to prove termination of the loop. For O, we can neither use that `curr` points to a struct whose `next` field contains the null pointer (as in L), nor that its `next` field points to another struct whose `next` field contains the null pointer (as in M).

We solve this problem by introducing *list invariants*. In our example, we will infer an invariant stating that `curr` points to a list of length $x_\ell \geq 1$. This invariant also implies that all struct fields are allocated and that there is no sharing.

To merge two states s and s' at the same program position with $domain(LV^s) = domain(LV^{s'})$, we introduce a fresh symbolic variable $x_{\texttt{var}}$ for each program variable `var` and use instantiations μ_s and $\mu_{s'}$ which map $x_{\texttt{var}}$ to the corresponding symbolic variables of s and s'. For the merged state $\overline{s}$, we set $LV^{\overline{s}}(\texttt{var}) = x_{\texttt{var}}$. Moreover, we identify corresponding variables that only occur in the memory components and extend μ_s and $\mu_{s'}$ accordingly. In a second step, we check which constraints from the memory components and the knowledge base hold in both states in order to find invariants that we can add to the memory components and the knowledge base of $\overline{s}$. For example, if $[\![\mu_s(x), \mu_s(x^{end})]\!] \in AL^s$ and $[\![\mu_{s'}(x), \mu_{s'}(x^{end})]\!] \in AL^{s'}$ for $x, x^{end} \in \mathcal{V}_{sym}$, then $[\![x, x^{end}]\!]$ is added to $AL^{\overline{s}}$. To extend this heuristic to lists, we have to regard several memory entries together. If there is an $\texttt{ad} \in \mathcal{V}_{\mathcal{P}}$ such that $\mu_s(x_{\texttt{ad}})$ and $\mu_{s'}(x_{\texttt{ad}})$ both point to lists of type `ty` but of different lengths $\ell_s \neq \ell_{s'}$ with $\ell_s, \ell_{s'} \geq 1$, then we create a list invariant.

In our example, L and M contain lists of length $\ell_L = 1$ and $\ell_M = 2$. Thus, when merging L and M to a new state O (see Fig. 3), the lists are merged to a list invariant of variable

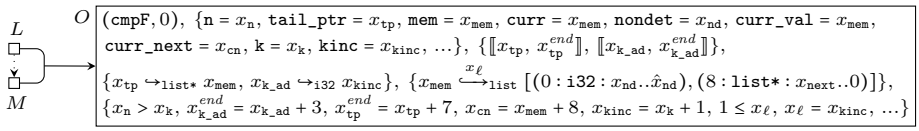


Figure 3 Merging of States

length x_ℓ and our technique adds the formulas $1 \le x_\ell$ and $x_\ell = x_{\texttt{kinc}}$ to KB^O. Moreover, the `i32` value of the first element is identified with $x_{\texttt{nd}}$, since $\mu_L(x_{\texttt{nd}})$ is equal to the first value of the first list element in L and $\mu_M(x_{\texttt{nd}})$ is equal to the first value of the first list element in M. Similarly, the values of the last list elements are identified with 0, as in L and M.

After merging s and s', we continue symbolic execution from the generalized state $\overline{s}$. The next time we reach the same program position, we might have to merge the corresponding states again. As described in [16], we use a heuristic which ensures that after a finite number of iterations, a state is reached that is represented by an *already existing* state in the SEG. Then symbolic execution can continue from this more general state instead. So the construction always ends in a complete SEG or an SEG containing the state *ERR*.

To prove termination of a program $\mathcal{P}$, as in [16] the cycles of the SEG are translated to an ITS whose termination implies termination of $\mathcal{P}$. In our example, the first cycle of the SEG (corresponding to the `for` loop of the C program) yields transitions which terminate since `k` is increased until it reaches `n`. The second cycle (corresponding to the `while` loop) terminates since the length of the list decreases. Although there is no program variable for the length, due to our list invariants the states of the SEG contain variables for this length, which are also passed to the ITS. Thus, the resulting transitions clearly terminate.

4 Conclusion and Evaluation

We presented a new approach for proving memory safety and termination of C programs on lists. The main idea is to extend the states in the SEG by *list invariants*. We developed techniques to infer and modify list invariants automatically during the symbolic execution.

List invariants abstract from a concrete number of memory allocations to a list of allocations of variable length while preserving knowledge about some of the contents and the list shape. They also contain information on the memory arrangement of the list fields which is needed to prove memory safety for programs that access fields via pointer arithmetic. The symbolic variables for the list length and the first and last values of list elements are preserved when generating an ITS from the SEG. Thus, they can be used in its termination proof.

In [3, 4, 14] we developed a technique for termination analysis of Java, based on a program transformation to *integer term rewrite systems* instead of ITSs. This approach does not require specific list invariants as recursive data structures on the heap are abstracted to terms. However, these terms are unsuitable for C, since they cannot express memory allocations and the connection to their contents.

For every abstract state, we define a corresponding *separation logic* formula to define which concrete states are represented by the abstract state, see [11]. To extend this formula to list invariants, we use a fragment similar to *quantitative* separation logic [2], extending conventional separation logic by list predicates. Based on separation logic with inductive predicates, [7] also uses a representation of lists with offsets which is similar to our list invariants, and [5] presents an abstract domain to represent lists in the presence of pointer arithmetic. However, in contrast to our work, [5, 7] are not concerned with termination analysis.

Separation logic predicates for termination of list programs were also used in [1], but

their list predicates only consider the list length and the recursive field, but no other fields or offsets. The tools Cyclist [15] and HipTNT+ [12] are integrated in separation logic systems which also allow to define heap predicates. However, they require annotations and hints which parameters of the list predicates are needed as a termination measure. The tool 2LS [13] also provides basic support for dynamic data structures. But all these approaches are not suitable if termination depends on the contents or the shape of data structures combined with pointer arithmetic. In [6], programs can be annotated with arithmetic and structural properties to reason about termination. In contrast, our approach does not need hints or annotations, but finds termination arguments fully automatically.

We implemented our approach in AProVE [16]. Since existing tools can hardly prove termination of C programs with lists, the current benchmarks for termination analysis contain almost no list programs. In 2017, a set of 18 C programs on lists was added to the *Termination* category of the *Competition on Software Verification* (*SV-COMP*),[1] where nine of them are terminating. Two of these nine programs do not need list invariants, because they just create a list without operating on it afterwards. The remaining seven terminating programs create a list and then traverse it, search for a value, or append lists and compute the length afterwards. Our new version of AProVE is the only termination prover that succeeds for five of these seven programs, since here termination depends on the shape or contents of a list after its creation.

For the *Termination Competition* 2022,[2] we submitted 18 terminating C programs on lists (different from the ones at *SV-COMP*), where AProVE succeeds on 17 of them. Two of these programs just create a list. Three traverse it afterwards (by a loop or recursion), and ten search for a value, where for nine, also the list contents are relevant for termination. Three programs perform common operations like inserting or deleting an element.

Without list invariants, in each collection AProVE only proves termination of two examples that just create a list without traversing it afterwards, and non-termination for one example in the *SV-COMP* collection. AProVE and UAutomizer were the most powerful C termination tools in *SV-COMP* and the *Termination Competition* 2022, with UAutomizer winning the former and AProVE winning the latter. To download AProVE, its web interface, and details on our experiments, see https://aprove-developers.github.io/recursive_structs.

	SV-C T.	SV-C Non-T.	TermCmp T.
AProVE	7 (of 9)	5 (of 9)	17 (of 18)
UAutomizer	2 (of 9)	7 (of 9)	1 (of 18)

References

1 J. Berdine, B. Cook, D. Distefano, and P. W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proc. CAV '06*, LNCS 4144, pages 386–400, 2006.

2 M. Bozga, R. Iosif, and S. Perarnau. Quantitative separation logic and programs with lists. *J. Aut. Reasoning*, 45(2):131–156, 2010.

3 M. Brockschmidt, C. Otto, and J. Giesl. Modular termination proofs of recursive Java Bytecode programs by term rewriting. In *Proc. RTA '11*, LIPIcs 10, pages 155–170, 2011.

4 M. Brockschmidt, R. Musiol, C. Otto, and J. Giesl. Automated termination proofs for Java programs with cyclic data. In *Proc. CAV '12*, LNCS 7358, pages 105–122, 2012.

5 C. Calcagno, D. Distefano, P. W.' O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *Proc. SAS '06*, LNCS 4134, pages 182–203, 2006.

[1] https://sv-comp.sosy-lab.org/2022/
[2] https://termination-portal.org/wiki/Termination_Competition_2022

6 C. David, D. Kroening, M. Lewis, and J. Vitek. Propositional reasoning about safety and termination of heap-manipulating programs. In *Proc. ESOP '15*, LNCS 9032, pages 661–684, 2015.

7 K. Dudka, P. Peringer, and T. Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In *Proc. CAV '11*, LNCS 6806, pages 372–378, 2011.

8 F. Emrich, J. Hensel, and J. Giesl. AProVE: Modular termination analysis of memory-manipulating C programs. *CoRR*, abs/2302.02382, 2023.

9 J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *J. Aut. Reasoning*, 58(1):3–31, 2017.

10 J. Hensel, J. Giesl, F. Frohn, and T. Ströder. Termination and complexity analysis for programs with bitvector arithmetic by symbolic execution. *J. Log. Alg. Meth. Prog.*, 97:105–130, 2018.

11 J. Hensel and J. Giesl. Proving termination of C programs with lists. In *Proc. CADE '23*, LNCS, 2023. To appear. Full version appeared in *CoRR*, abs/2305.12159.

12 T. C. Le, S. Qin, and W. Chin. Termination and non-termination specification inference. In *Proc. PLDI '15*, pages 489–498, 2015.

13 V. Malík, Š. Martiček, P. Schrammel, M. Srivas, T. Vojnar, and J. Wahlang. 2LS: Memory safety and non-termination. In *Proc. TACAS '18*, LNCS 10806, pages 417–421, 2018.

14 C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *Proc. RTA '10*, LIPIcs 6, pages 259–276, 2010.

15 R. N. S. Rowe and J. Brotherston. Automatic cyclic termination proofs for recursive procedures in separation logic. In *Proc. CPP '17*, pages 53–65, 2017.

16 T. Ströder, J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, P. Schneider-Kamp, and C. Aschermann. Automatically proving termination and memory safety for programs with pointer arithmetic. *J. Aut. Reasoning*, 58(1):33–65, 2017.

Higher-Order LCTRSs and Their Termination*

Liye Guo
Radboud University, Netherlands

Cynthia Kop
Radboud University, Netherlands

Abstract

Logically constrained term rewriting systems (LCTRSs) are a program analyzing formalism with native support for data types which are not (co)inductively defined. As a first-order formalism, LCTRSs have accommodated only analysis of imperative programs so far. In this paper, we present a higher-order variant of the LCTRS formalism, which can be used to analyze functional programs. Then we study the termination problem and define a higher-order recursive path ordering (HORPO) for this new formalism.

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

Logically constrained term rewriting systems (LCTRSs) [4, 1] are a formalism for analyzing programs. In real-world programming, data types such as integers, as opposed to natural numbers, and arrays are prevalent. Any practical technique for program analysis should be prepared to handle these. One of the defining features of the LCTRS formalism is its native support for such data types, which are not (co)inductively defined and need to be encoded if handled by more traditional TRSs. Another benefit of the formalism is its separation between logical constraints modeling the control flow and other terms representing the program states.

So far, program analysis on the basis of LCTRSs has concerned imperative programs since LCTRSs were introduced as a first-order formalism. We are naturally curious to see if functional programs can also be analyzed by constrained rewriting. What we present here is our ongoing exploration in this direction: First, we define a higher-order variant of the LCTRS formalism, which, despite the absence of lambda abstractions, is capable of representing some real-world functional programs straightforwardly. Then we approach the termination problem for this new formalism by defining a higher-order recursive path ordering (HORPO).

2 LCSTRS

We start defining *logically constrained simply-typed term rewriting systems* (LCSTRSs) with types and terms. We postulate a set $\mathcal{S}$, whose elements we call *sorts*, and a subset $\mathcal{S}_\vartheta$ of $\mathcal{S}$, whose elements we call *theory sorts*. The set $\mathcal{T}$ of *types* and its subset $\mathcal{T}_\vartheta$, called the set of *theory types*, are generated as follows: $\mathcal{T} ::= \mathcal{S} \mid (\mathcal{T} \to \mathcal{T})$ and $\mathcal{T}_\vartheta ::= \mathcal{S}_\vartheta \mid (\mathcal{S}_\vartheta \to \mathcal{T}_\vartheta)$. Right-associativity is assigned to $\to$ so we can omit some parentheses in types. We assume given disjoint sets $\mathcal{F}$ and $\mathcal{V}$, whose elements we call *function symbols* and *variables*, respectively. The grammar $\mathbb{T} ::= \mathcal{F} \mid \mathcal{V} \mid (\mathbb{T}\ \mathbb{T})$ generates the set $\mathbb{T}$ of *pre-terms*. Left-associativity is assigned to the juxtaposition operation in the above grammar so $t_0\ t_1\ t_2$ stands for $((t_0\ t_1)\ t_2)$,

* This paper was presented at HOR 2023 on July 4 in Rome, Italy.

for example. We assume that every function symbol and variable is assigned a unique type. Typing works as expected: if pre-terms t_0 and t_1 have types $A \to B$ and A, respectively, $t_0\ t_1$ has type B. Pre-terms having a type are called *terms*. We write $t : A$ if a term t has type A. We postulate a subset $\mathcal{F}_\vartheta$ of $\mathcal{F}$, whose elements we call *theory (function) symbols*, and assume that theory symbols have theory types. Terms constructed with only theory symbols and variables are called *theory terms*. The set of variables in a term t, denoted by $\mathrm{Var}(t)$, is defined as follows: $\mathrm{Var}(f) = \emptyset$, $\mathrm{Var}(x) = \{\, x \,\}$ and $\mathrm{Var}(t_0\ t_1) = \mathrm{Var}(t_0) \cup \mathrm{Var}(t_1)$. A term t is called a *ground term* if $\mathrm{Var}(t) = \emptyset$. Note that ground theory terms always have theory types.

Theory terms are distinguished because they will be treated specially when we define the rewrite relation. First, let us define the interpretation of ground theory terms. We postulate an $\mathcal{S}_\vartheta$-indexed family of sets $(\mathfrak{X}_A)_{A\in\mathcal{S}_\vartheta}$, and extend it to a $\mathcal{T}_\vartheta$-indexed family of sets by letting $\mathfrak{X}_{A\to B}$ be the set of maps from $\mathfrak{X}_A$ to $\mathfrak{X}_B$. Now we assume given a $\mathcal{T}_\vartheta$-indexed family of maps $(\llbracket\cdot\rrbracket_A)_{A\in\mathcal{T}_\vartheta}$ where $\llbracket\cdot\rrbracket_A$ assigns to each theory symbol whose type is A an element of $\mathfrak{X}_A$ and is bijective if $A \in \mathcal{S}_\vartheta$. Theory symbols whose type is a theory sort are called *values*. We extend each indexed map $\llbracket\cdot\rrbracket_B$ to a map that assigns to each **ground theory term** whose type is B an element of $\mathfrak{X}_B$ by letting $\llbracket t_0\ t_1\rrbracket_B$ be $\llbracket t_0\rrbracket_{A\to B}(\llbracket t_1\rrbracket_A)$. We omit the type and write just $\llbracket\cdot\rrbracket$ whenever the type can be deduced from the context. $\llbracket t\rrbracket$ is called the *interpretation* of t.

A *substitution* is a type-preserving map from variables to terms. Every substitution σ extends to a type-preserving map $\bar\sigma$ from terms to terms. We write $t\sigma$ for $\bar\sigma(t)$ and define it as follows: $f\sigma = f$, $x\sigma = \sigma(x)$ and $(t_0\ t_1)\sigma = (t_0\sigma)\ (t_1\sigma)$. Now we postulate a theory sort $\mathbb{B}$ and theory symbols $\bot : \mathbb{B}$ and $\top : \mathbb{B}$. Let $\mathfrak{X}_\mathbb{B}$ be $\{\, \mathfrak{o}, \mathfrak{1} \,\}$ and assume $\llbracket\bot\rrbracket = \mathfrak{o}$ and $\llbracket\top\rrbracket = \mathfrak{1}$. A *rewrite rule* $\ell \to r\ [\varphi]$ is a triple where

- ℓ and r are terms which have the same type,
- ℓ is not a theory term,
- φ is a *logical constraint*, i.e., φ is a theory term whose type is $\mathbb{B}$ and the type of each variable in $\mathrm{Var}(\varphi)$ is a theory sort, and
- the type of each variable in $\mathrm{Var}(r) \setminus \mathrm{Var}(\ell)$ is a theory sort.

A substitution σ is said to *respect* a rewrite rule $\ell \to r\ [\varphi]$ if $\sigma(x)$ is a value for all $x \in \mathrm{Var}(\varphi) \cup (\mathrm{Var}(r) \setminus \mathrm{Var}(\ell))$ and $\llbracket\varphi\sigma\rrbracket = \mathfrak{1}$. A set $\mathcal{R}$ of rewrite rules induces a *rewrite relation* $\to_\mathcal{R}$ on terms such that $t \to_\mathcal{R} t'$ if and only if one of the following conditions is true:

- $t = \ell\sigma$ and $t' = r\sigma$ for some $\ell \to r\ [\varphi] \in \mathcal{R}$ and some substitution σ that respects $\ell \to r\ [\varphi]$.
- $t = f\ v_1 \cdots v_n$ where f is a theory symbol but not a value while v_i is a value for all i, the type of t is a theory sort, and t' is the unique value such that $\llbracket f\ v_1 \cdots v_n\rrbracket = \llbracket t'\rrbracket$.
- $t = t_0\ t_1$, $t' = t_0'\ t_1$ and $t_0 \to_\mathcal{R} t_0'$.
- $t = t_0\ t_1$, $t' = t_0\ t_1'$ and $t_1 \to_\mathcal{R} t_1'$.

Logical constraints are essentially first-order—higher-order variables are excluded and theory symbols take only first-order arguments. We adopt this restriction because many conditions in functional programs are still first-order and solving higher-order constraints is hard. That is not to say that higher-order constraints are of no interest; we simply leave them out of the scope of LCSTRSs.

Below is an example LCSTRS:

$$\begin{array}{llll} \mathsf{init} \to \mathsf{fact}\ n\ \mathsf{exit} & [\top] & \mathsf{fact}\ n\ k \to k\ 1 & [n \le 0] \\ \mathsf{comp}\ g\ f\ x \to g\ (f\ x) & [\top] & \mathsf{fact}\ n\ k \to \mathsf{fact}\ (n-1)\ (\mathsf{comp}\ k\ (*\ n)) & [n > 0] \end{array}$$

Here init and exit denote the start and the end of the program, respectively. The core of the program is fact, which computes the factorial function in continuation-passing style, and comp is an auxiliary function for function composition. Integer literals and operators are theory symbols and are interpreted in the standard way. Note that we use infix notation to improve readability. The occurrence of n in the rewrite rule defining init is an example of a variable that occurs on the right-hand side but not on the left-hand side of a rewrite rule. Such variables can be used to model user input.

Let $\mathcal{R}$ denote the set of rewrite rules in the example and consider the rewrite sequence

$$\begin{aligned}\mathsf{fact}\ 1\ \mathsf{exit} &\to_{\mathcal{R}} \mathsf{fact}\ (1-1)\ (\mathsf{comp}\ \mathsf{exit}\ (*\ 1))\\ &\to_{\mathcal{R}} \mathsf{fact}\ 0\ (\mathsf{comp}\ \mathsf{exit}\ (*\ 1))\\ &\to_{\mathcal{R}} \mathsf{comp}\ \mathsf{exit}\ (*\ 1)\ 1\\ &\to_{\mathcal{R}} \cdots\end{aligned}$$

In the second step, no rewrite rule is invoked. Such rewrite steps are called *calculation steps*. We can write $\to_\emptyset$ for a calculation step. Terms s and t are said to be *joinable* by $\to_\emptyset$, written as $s \downarrow_\emptyset t$, if there exists a term r such that $s \to_\emptyset^* r$ and $t \to_\emptyset^* r$.

3 Termination

In order to prove that a given (unconstrained) TRS $\mathcal{R}$ is terminating, we usually look for a stable, monotonic and well-founded relation $\succ$ which orients every rewrite rule in $\mathcal{R}$, i.e., $\ell \succ r$ for all $\ell \to r \in \mathcal{R}$. This standard technique, however, requires a few tweaks to be applied to LCSTRSs. First, stability should be tightly coupled with rule orientation because every rewrite rule in an LCSTRS is equipped with a logical constraint, which decides what substitutions are expected when the rewrite rule is invoked. Therefore, we say that a type-preserving relation $\succ$ on terms *orients* a rewrite rule $\ell \to r\ [\varphi]$ if $\ell\sigma \succ r\sigma$ for each substitution σ that **respects** the rewrite rule. Second, the monotonicity requirement can be weakened because ℓ is never a theory term in a rewrite rule $\ell \to r\ [\varphi]$. We say that a type-preserving relation $\succ$ on terms is *rule-monotonic* if $t_0 \succ t_0{}'$ implies $t_0\ t_1 \succ t_0{}'\ t_1$ when t_0 is not a theory term, and $t_1 \succ t_1{}'$ implies $t_0\ t_1 \succ t_0\ t_1{}'$ when t_1 is not a theory term.

We present a tentative definition of HORPO [2] on LCSTRSs. For each theory sort A, we postulate theory symbols $\sqsupset_A : A \to A \to \mathbb{B}$ and $\sqsupseteq_A : A \to A \to \mathbb{B}$ such that $[\![\sqsupset_A]\!]$ is a well-founded ordering on $\mathfrak{X}_A$ and $[\![\sqsupseteq_A]\!]$ is the reflexive closure of $[\![\sqsupset_A]\!]$. We omit the sort and write just $\sqsupset$ and $\sqsupseteq$ whenever the sort can be deduced from the context. Given the *precedence* $\blacktriangleright$, a well-founded ordering on function symbols such that $f \blacktriangleright g$ for all $f \in \mathcal{F} \setminus \mathcal{F}_\vartheta$ and $g \in \mathcal{F}_\vartheta$, and the *status* stat, a map from $\mathcal{F}$ to $\{\mathfrak{l}, \mathfrak{m}_2, \mathfrak{m}_3, \dots\}$, the *higher-order recursive path ordering* (HORPO) $(\succsim_\varphi, \succ_\varphi)$ is a family of type-preserving relation pairs on terms indexed by logical constraints and defined as follows:

- $s \succsim_\varphi t$ if and only if one of the following conditions is true:
  - s and t are theory terms whose type is a theory sort, $\mathrm{Var}(\varphi) \supseteq \mathrm{Var}(s) \cup \mathrm{Var}(t)$ and $\varphi \models \sqsupseteq\ s\ t$.
  - $s \succ_\varphi t$.
  - $s \downarrow_\emptyset t$.
  - s is not a theory term, $s = s_0\ s_1$, $t = t_0\ t_1$, $s_0 \succsim_\varphi t_0$ and $s_1 \succsim_\varphi t_1$.
- $s \succ_\varphi t$ if and only if one of the following conditions is true:
  - s and t are theory terms whose type is a theory sort, $\mathrm{Var}(\varphi) \supseteq \mathrm{Var}(s) \cup \mathrm{Var}(t)$ and $\varphi \models \sqsupset\ s\ t$.

  - s and t have the same type and $s \rhd_\varphi t$.
  - s is not a theory term, $s = f\ s_1 \cdots s_n$ where f is a function symbol, $t = f\ t_1 \cdots t_n$, $s_i \succsim_\varphi t_i$ for all i and there exists i such that $s_i \succ_\varphi t_i$.
  - s is not a theory term, $s = x\ s_1 \cdots s_n$ where x is a variable, $t = x\ t_1 \cdots t_n$, $s_i \succsim_\varphi t_i$ for all i and there exists i such that $s_i \succ_\varphi t_i$.
- $s \rhd_\varphi t$ if and only if s is not a theory term, $s = f\ s_1 \cdots s_m$ where f is a function symbol, and one of the following conditions is true:
  - $s_i \succsim_\varphi t$ for some i.
  - $t = t_0\ t_1 \cdots t_n$ and $s \rhd_\varphi t_i$ for all i.
  - $t = g\ t_1 \cdots t_n$, $f \blacktriangleright g$ and $s \rhd_\varphi t_i$ for all i.
  - $t = f\ t_1 \cdots t_n$, $\mathsf{stat}(f) = \mathsf{l}$, $s_1 \cdots s_m \succ_\varphi^{\mathsf{l}} t_1 \cdots t_n$ and $s \rhd_\varphi t_i$ for all i.
  - $t = f\ t_1 \cdots t_n$, $\mathsf{stat}(f) = \mathsf{m}_k$, $k \leq n$, $s_1 \cdots s_{\min(m,k)} \succ_\varphi^{\mathsf{m}} t_1 \cdots t_k$ and $s \rhd_\varphi t_i$ for all i.
  - t is a value or $t \in \mathrm{Var}(\varphi)$.

In the above, $s_1 \cdots s_m \succ_\varphi^{\mathsf{l}} t_1 \cdots t_n$ if and only if $\exists i \leq \min(m,n)\ (s_i \succ_\varphi t_i \wedge \forall j < i\ s_j \succsim_\varphi t_j)$, $\succ_\varphi^{\mathsf{m}}$ is the generalized multiset extension of $(\succsim_\varphi, \succ_\varphi)$ (see [3]), and $\varphi \models \varphi'$ denotes, on the assumption that φ and φ' are logical constraints such that $\mathrm{Var}(\varphi) \supseteq \mathrm{Var}(\varphi')$, that for each substitution σ which maps variables in $\mathrm{Var}(\varphi)$ to values, $[\![\varphi\sigma]\!] = \mathtt{1}$ implies $[\![\varphi'\sigma]\!] = \mathtt{1}$.

In an unconstrained setting, we make a relation $\succ$ orient a rewrite rule $\ell \to r$ by asserting $\ell \succ r$; now with the above definition of HORPO on LCSTRSs, if $\ell \succ_\varphi r$, it is $\succ_\top$ that should orient the rewrite rule $\ell \to r\ [\varphi]$. Once a combination of $\sqsupset$, $\blacktriangleright$ and stat that guarantees $\ell \succ_\varphi r$ for all $\ell \to r\ [\varphi] \in \mathcal{R}$ is present, we can conclude that the LCSTRS $\mathcal{R}$ is terminating. The soundness of this method relies on the following properties of $\succ_\varphi$, which we must prove:

- $\succ_\top$ orients $\ell \to r\ [\varphi]$ if $\ell \succ_\varphi r$.
- $\succ_\top$ is rule-monotonic.
- $\succ_\top$ is well-founded.
- $\to_\emptyset\ ;\ \succ_\top\ \subseteq\ \succ_\top$.

Note that $\to_\emptyset$ is well-founded because the size strictly decreases through every calculation step.

Consider the example LCSTRS given in the previous section. Any combination of $\sqsupset$, $\blacktriangleright$ and stat that satisfies the following properties would witness the well-foundedness of $\to_{\mathcal{R}}$: $[\![\sqsupset]\!] = \lambda xy.\ x > 0 \wedge x > y$, $\mathsf{init} \blacktriangleright \mathsf{fact} \blacktriangleright \mathsf{comp}$, $\mathsf{init} \blacktriangleright \mathsf{exit}$ and $\mathsf{stat}(\mathsf{fact}) = \mathsf{l}$.

4 Future Work

LCSTRSs are still a work in progress. While the formalism itself is in a somewhat stable state, the above method for termination analysis is in active development. First and foremost, we need to prove that HORPO on LCSTRSs has the expected properties. When the theory is complete, we would like to make a tool to automate the finding of HORPO on LCSTRSs. It would also be interesting to explore other methods for termination analysis on the new formalism, including StarHorpo [3] (a transitive variant of HORPO), interpretation-based methods and dependency pairs. Another direction is to go beyond LCSTRSs by augmenting the formalism with lambda abstractions or higher-order constraints.

References

1 C. Fuhs, C. Kop, and N. Nishida. Verifying procedural programs via constrained rewriting induction. *ACM Transactions on Computational Logic*, 18(2):14:1–14:50, 2017. doi:10.1145/3060143.

2 J.-P. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *Proc. LICS*, pages 402–411, 1999. doi:10.1109/LICS.1999.782635.

3 C. Kop. *Higher Order Termination*. PhD thesis, VU Amsterdam, 2012.

4 C. Kop and N. Nishida. Term rewriting with logical constraints. In *Proc. FroCoS*, pages 343–358, 2013. doi:10.1007/978-3-642-40885-4_24.

Logical Methods in Computer Science
Volume 21, Issue 2, 2025, pp. 29:1–29:22
https://lmcs.episciences.org/

Submitted Mar. 06, 2024
Published Jun. 27, 2025

HYDRA BATTLES AND AC TERMINATION

NAO HIROKAWA [a] AND AART MIDDELDORP [b]

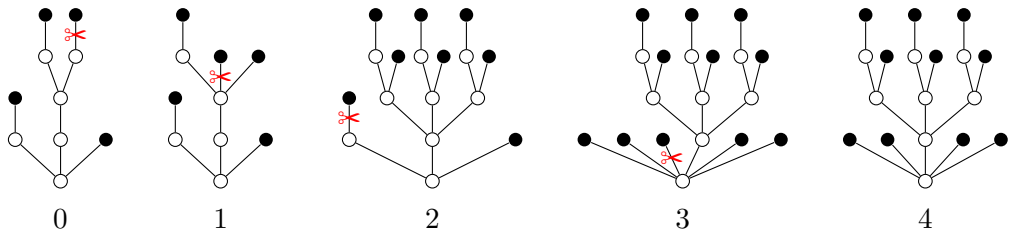
[a] School of Information Science, JAIST, Japan
e-mail address: hirokawa@jaist.ac.jp

[b] Department of Computer Science, University of Innsbruck, Austria
e-mail address: aart.middeldorp@uibk.ac.at

Abstract. We present a new encoding of the Battle of Hercules and Hydra as a rewrite system with AC symbols. Unlike earlier term rewriting encodings, it faithfully models any strategy of Hercules to beat Hydra. To prove the termination of our encoding, we employ type introduction in connection with many-sorted semantic labeling for AC rewriting and AC-MPO, a new AC compatible reduction order that can be seen as a much weakened version of AC-RPO.

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 corresponds to the number of decapitations so far. Hydra dies if there are no heads left, in that case Hercules wins. The following sequence shows an example fight:



Though the number of heads can grow considerably in one step, it turns out that the fight always terminates, and Hercules will win independent of his strategy. Proving termination

Key words and phrases: battle of Hercules and Hydra, term rewriting, AC termination.

The first author is supported by JSPS KAKENHI Grant Number JP22K11900. Part of this work was performed when the second author was employed at the Future Value Creation Research Center of Nagoya University, Japan.

DOI:10.46298/LMCS-21(2:29)2025

of the Battle is challenging since Kirby and Paris proved in their landmark paper [KP82] that termination for an arbitrary (computable) strategy is independent of Peano arithmetic. In [KP82] a termination argument based on ordinals is used.

Starting with [DJ90, p. 271], several TRS encodings of the Battle of Hercules and Hydra have been proposed and studied [Buc06, DM07, FZ96, Mos09, Tou98]. Touzet [Tou98] was the first to give a rigorous termination proof and in [ZWM15] the automation of ordinal interpretations is discussed. In this article we present yet another encoding. In contrast to earlier TRS encodings that model a specific strategy, it uses AC matching to represent *arbitrary* battles. To prove its termination, we adapt existing termination methods for AC rewriting.

The remainder of the article is organized as follows. After recalling some basic definitions in Section 2, we present our new encoding of the Battle in Section 3. We give a rigorous proof that our encoding faithfully represents the Battle. In Section 4 we present many-sorted semantic labeling for AC rewriting and apply it to our encoding. This results in an infinite AC rewrite system, which can be shown terminating by Rubio's AC-RPO [Rub02]. As a matter of fact, we do not need the full power of AC-RPO. Inspired by Steinbach's AC-KBO [Ste90], in Section 5 we introduce AC-MPO, a much weakened version of AC-RPO, and show that it is powerful enough for our purpose. Some of the properties of AC-MPO are proved in the appendix.

Related work is discussed in Section 6. In particular, we comment on earlier encodings of the Battle. We conclude in Section 7 with suggestions for future research.

A preliminary version of this article appeared in the proceedings of the 8th International Conference on Formal Structures for Computation and Deduction [HM23]. AC-MPO is a new result. New examples provide further illustration of the simulation of the Battle of Hercules and Hydra.

2. Preliminaries

Let $\mathcal{S}$ be a set of *sorts*. An *$\mathcal{S}$-sorted signature* $\mathcal{F}$ consists of function symbols f having a sort declaration $S_1 \times \cdots \times S_n \to S$. Here $S_1, \ldots, S_n$ and S are sorts in $\mathcal{S}$ and n is the *arity* of f. By $f^{(n)}$ we indicate that f has arity n. Let $\mathcal{V}$ be a countably infinite set of variables, where every variable has its own sort. We assume the existence of infinitely many variables of each sort. *Terms* of sort S are inductively defined as usual: Every variable of sort S is a term of sort S and if f has sort declaration $S_1 \times \cdots \times S_n \to S$ and t_i is a term of sort S_i for all $1 \leqslant i \leqslant n$ then $f(t_1, \ldots, t_n)$ is a term of sort S. *Ground terms* are terms without variables. The *root symbol* $\mathsf{root}(t)$ of a term t is t if it is a variable, and f if $t = f(t_1, \ldots, t_n)$. For every sort S we introduce a fresh constant $\Box_S$, called the *hole*. A term over $\mathcal{F} \uplus \{\Box_S \mid S \in \mathcal{S}\}$ is a *context* over $\mathcal{F}$ if it contains exactly one hole. Given a context C and a term t, we write $C[t]$ for the term resulting from replacing the hole in C by t. We write $s \trianglelefteq t$ if $t = C[s]$ for some context C. We write $s \triangleleft t$ if $s \trianglelefteq t$ and $s \neq t$. A mapping σ that associates each variable to a term of the same sort is a *substitution* if its domain $\{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ is finite. The application $t\sigma$ of σ to a term t is defined as $\sigma(t)$ if t is a variable and $f(t_1\sigma, \ldots, t_n\sigma)$ if $t = f(t_1, \ldots, t_n)$. A binary relation $\to$ on terms is *closed under substitutions* if $s\sigma \to t\sigma$ whenever $s \to t$, for all substitutions σ. It is *closed under contexts* if $C[s] \to C[t]$ whenever $s \to t$, for all contexts C. It has the *subterm property* if the inclusion $\triangleright \subseteq \to$ holds. Moreover, the relation $\to$ is said to be a *rewrite relation* if it is

closed under contexts and substitutions. *Rewrite orders* are rewrite relations that are strict orders, and *reduction orders* are rewrite orders that are well-founded.

A *rewrite rule* $\ell \to r$ consists of two terms ℓ and r of the same sort such that all variables in r occur in ℓ. A (many-sorted) *term rewrite system* (TRS) is a set of rewrite rules. We denote by $\to_{\mathcal{R}}$ the smallest rewrite relation that contains the pairs of the TRS $\mathcal{R}$. A rule $\ell \to r$ is *non-collapsing* if r is not a variable. A TRS is called *non-collapsing* if all rules are non-collapsing. A TRS $\mathcal{R}$ is *terminating* if $\to_{\mathcal{R}}$ is well-founded.

Let $\mathcal{F}_{\mathsf{AC}}$ be a subset of the binary function symbols in $\mathcal{F}$ that have sort declarations of the form $S \times S \to S$. We denote by AC the set of equations

$$f(f(x,y),z) \approx f(x,f(y,z)) \qquad\qquad f(x,y) \approx f(y,x)$$

expressing the associativity and commutativity of each $f \in \mathcal{F}_{\mathsf{AC}}$. Since equations in AC are rewrite rules, we can view AC as a TRS. Using this fact, we define the relation $=_{\mathsf{AC}}$ as the reflexive, transitive, and symmetric closure of $\to_{\mathsf{AC}}$. Let $\mathcal{R}$ be a TRS. The relation $=_{\mathsf{AC}} \cdot \to_{\mathcal{R}} \cdot =_{\mathsf{AC}}$ is called AC rewriting and abbreviated by $\to_{\mathcal{R}/\mathsf{AC}}$. We say that $\mathcal{R}$ is *AC terminating* if $\to_{\mathcal{R}/\mathsf{AC}}$ is well-founded. A reduction order $>$ is *AC-compatible* if the inclusion $=_{\mathsf{AC}} \cdot > \cdot =_{\mathsf{AC}} \subseteq >$ holds. AC termination of a TRS $\mathcal{R}$ can be shown by finding an AC-compatible reduction order such that $\mathcal{R} \subseteq >$ holds.

The above definitions specialize to the usual unsorted setting when the set of sorts is a singleton set.

Finally, we recall two order extensions. Let $>$ be a strict order on a set A. The *lexicographic extension* $>^{\mathsf{lex}}$ of $>$ is defined on tuples over A as follows: $(a_1,\dots,a_m) >^{\mathsf{lex}} (b_1,\dots,b_n)$ if $n = m$ and there exists an index $1 \leqslant k \leqslant n$ such that $a_k > b_k$ and $a_i = b_i$ for all $i < k$. The *multiset extension* $>^{\mathsf{mul}}$ of $>$ is defined on multisets over A as follows: $M >^{\mathsf{mul}} N$ if there exist multisets X and Y such that $N = (M - X) \uplus Y$, $\varnothing \neq X \subseteq M$, and every $b \in Y$ admits an element $a \in X$ with $a > b$.

3. Encoding

First we give a formal account of the Hydra Battle.

Definition 3.1. To represent Hydras, we use a signature containing a constant symbol h representing a head, a binary symbol $\mid$ for siblings, and a unary function symbol i representing the internal nodes. We use infix notation for $\mid$ and declare it to be an AC symbol. We write $\mathcal{T}_{\mathcal{H}}$ for the set of ground terms over $\{\mathsf{h},\mathsf{i},\mid\}$. *Encodings* of Hydras are terms t in $\mathcal{T}_{\mathcal{H}}$ with $\mathsf{root}(t) \in \{\mathsf{h},\mathsf{i}\}$.

To improve readability we omit parentheses in terms with nested $\mid$ symbols in examples.

Example 3.2. The Hydras in the above example fight are represented by the terms

$$H_0 = \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{i}(\mathsf{h}))) \mid \mathsf{h})$$
$$H_1 = \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h} \mid \mathsf{h})) \mid \mathsf{h})$$
$$H_2 = \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h}) \mid \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h}) \mid \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h})) \mid \mathsf{h})$$
$$H_3 = \mathsf{i}(\mathsf{h} \mid \mathsf{h} \mid \mathsf{h} \mid \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h}) \mid \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h}) \mid \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h})) \mid \mathsf{h} \mid \mathsf{h})$$
$$H_4 = \mathsf{i}(\mathsf{h} \mid \mathsf{h} \mid \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h}) \mid \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h}) \mid \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{h})) \mid \mathsf{h} \mid \mathsf{h})$$

and they are encodings of Hydras. The term $\mathsf{h} \mid \mathsf{h}$ is included in $\mathcal{T}_{\mathcal{H}}$ but not regarded as an encoding of a Hydra.

Definition 3.3. Let n be a natural number. The TRS $\mathcal{R}_n$ operates on encodings of Hydras and consists of the following four rules:

$$\mathsf{i}(\mathsf{i}(\mathsf{h})) \xrightarrow{1} \mathsf{i}(\mathsf{h}^{n+2}) \qquad \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid y) \xrightarrow{3} \mathsf{i}(\mathsf{h}^{n+2} \mid y)$$
$$\mathsf{i}(\mathsf{i}(\mathsf{h} \mid x)) \xrightarrow{2} \mathsf{i}(\mathsf{i}(x)^{n+2}) \qquad \mathsf{i}(\mathsf{i}(\mathsf{h} \mid x) \mid y) \xrightarrow{4} \mathsf{i}(\mathsf{i}(x)^{n+2} \mid y)$$

Here t^k for $k \geqslant 1$ is defined inductively as follows:

$$t^k = \begin{cases} t & \text{if } k = 1 \\ t^{k-1} \mid t & \text{if } k > 1 \end{cases}$$

The transition relation $\Rightarrow_n$ on encodings of Hydras is defined as follows: $H \Rightarrow_n H'$ if

(1) $H = \mathsf{i}(\mathsf{h})$ and $H' = \mathsf{h}$, or
(2) $H =_{\mathsf{AC}} \mathsf{i}(\mathsf{h} \mid t)$ and $H' = \mathsf{i}(t)$ for some term t, or
(3) $H \to_{\mathcal{R}_n/\mathsf{AC}} H'$.

That H and H' are encodings of successive Hydras at stages n and $n+1$ in a battle is expressed as $H \Rightarrow_n H'$. So fights with Hydras are represented by finite or infinite sequences of the form $H_0 \Rightarrow_0 H_1 \Rightarrow_1 H_2 \Rightarrow_2 \cdots$.

Example 3.4 (continued from Example 3.2)**.** We have the following sequence:

$$H_0 \Rightarrow_0 H_1 \Rightarrow_1 \cdots \Rightarrow_3 H_4$$

For instance, the first step is verified as follows: Let $\ell \to r$ be the third rule in $\mathcal{R}_0$, and let $C = \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{i}(\Box) \mid \mathsf{h})$ and $\sigma = \{y \mapsto \mathsf{i}(\mathsf{h})\}$. Since $H_0 = C[\ell\sigma]$ and $H_1 =_{\mathsf{AC}} C[r\sigma]$ hold, we have $H_0 \to_{\mathcal{R}_0/\mathsf{AC}} H_1$ and so $H_0 \Rightarrow_0 H_1$ is obtained.

Now we present our TRS encoding of the Hydra Battle. We represent natural numbers n by $\mathsf{s}^n(\mathsf{0})$, which is abbreviated to n.

Definition 3.5. Let $\mathcal{F}$ be the signature consisting of the constant $\mathsf{0}$, the unary symbol s, the five binary symbols A–E as well as the three symbols h, i, and $\mid$ in Definition 3.1. The TRS $\mathcal{H}$ over $\mathcal{F}$ consists of the following 14 rewrite rules:

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

The Battle is started with the term $\mathsf{A}(\mathsf{0}, H)$ where H is the encoding of the initial Hydra. Rule 1 takes care of the dying Hydra ○—●. An application of the rule ends the battle with a term of the form $\mathsf{A}(n, \mathsf{h})$. The h denotes here a dead Hydra; a Hydra with only one head is represented by the term $\mathsf{i}(\mathsf{h})$. 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

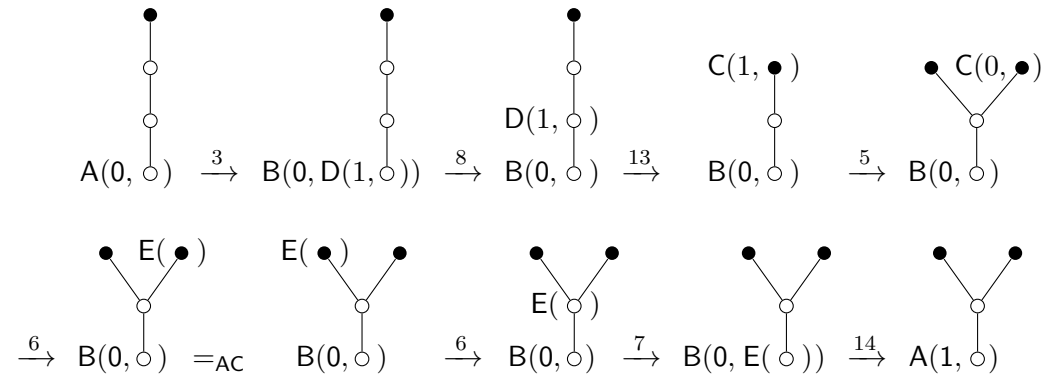


Figure 1: Rewriting from $\mathsf{A}(0, \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h}))))$ to $\mathsf{A}(1, \mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{h})))$.

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 3.6. *Let n be a natural number. If $H \Rightarrow_n H'$ then $\mathsf{A}(\mathsf{n}, H) \to^+_{\mathcal{H}/\mathsf{AC}} \mathsf{A}(\mathsf{s}(\mathsf{n}), H')$.*

Before presenting the proof, we illustrate how AC rewriting of $\mathcal{H}$ simulates fights with Hydras.

Example 3.7. Consider a fight with the Hydra of shape $\mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h})))$. The fight starts with the transition from $\mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h})))$ to $\mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{h}))$. This is simulated by the rewrite sequence

$$\begin{aligned}
&\mathsf{A}(0, \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h})))) \xrightarrow{3} \mathsf{B}(0, \mathsf{D}(\mathsf{s}(0), \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h}))))) \xrightarrow{8} \mathsf{B}(0, \mathsf{i}(\mathsf{D}(\mathsf{s}(0), \mathsf{i}(\mathsf{i}(\mathsf{h})))))\\
&\xrightarrow{13} \mathsf{B}(0, \mathsf{i}(\mathsf{i}(\mathsf{C}(\mathsf{s}(0), \mathsf{h})))) \xrightarrow{5} \mathsf{B}(0, \mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{C}(0, \mathsf{h})))) \xrightarrow{4} \mathsf{B}(0, \mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{E}(\mathsf{h}))))\\
&=_{\mathsf{AC}} \mathsf{B}(0, \mathsf{i}(\mathsf{i}(\mathsf{E}(\mathsf{h}) \mid \mathsf{h}))) \xrightarrow{6} \mathsf{B}(0, \mathsf{i}(\mathsf{E}(\mathsf{i}(\mathsf{h} \mid \mathsf{h})))) \xrightarrow{7} \mathsf{B}(0, \mathsf{E}(\mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{h}))))\\
&\xrightarrow{14} \mathsf{A}(\mathsf{s}(0), \mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{h})))
\end{aligned}$$

which is visualized in Figure 1. Rules 9–12 are variations of 8 and 13, which are used for handling nodes that have siblings. To illustrate these, consider the first step $H_0 \Rightarrow_0 H_1$ in the example fight in the introduction. The step is simulated by the following rewrite sequence:

$$\begin{aligned}
\mathsf{A}(0, H_0) &\xrightarrow{3} \mathsf{B}(0, \mathsf{D}(\mathsf{s}(0), H_0))\\
&=_{\mathsf{AC}} \cdot \xrightarrow{9} \mathsf{B}(0, \mathsf{i}(\mathsf{D}(\mathsf{s}(0), \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{i}(\mathsf{h})))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h}))\\
&\xrightarrow{8} \mathsf{B}(0, \mathsf{i}(\mathsf{i}(\mathsf{D}(\mathsf{s}(0), \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{i}(\mathsf{h})))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h}))\\
&\xrightarrow{12} \mathsf{B}(0, \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{C}(\mathsf{s}(0), \mathsf{h}) \mid \mathsf{i}(\mathsf{h}))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h}))\\
&\xrightarrow{5} \mathsf{B}(0, \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{C}(0, \mathsf{h}) \mid \mathsf{i}(\mathsf{h}))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h}))\\
&\xrightarrow{4} \mathsf{B}(0, \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{E}(\mathsf{h}) \mid \mathsf{i}(\mathsf{h}))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h}))
\end{aligned}$$

$$
\begin{aligned}
=_{\mathsf{AC}} \cdot &\xrightarrow{6} \mathsf{B}(0, \mathsf{i}(\mathsf{i}(\mathsf{E}(\mathsf{i}(\mathsf{h} \mid \mathsf{h} \mid \mathsf{i}(\mathsf{h})))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h})) \\
&\xrightarrow{7} \mathsf{B}(0, \mathsf{i}(\mathsf{E}(\mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{h} \mid \mathsf{i}(\mathsf{h})))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h})) \\
&\xrightarrow{6} \mathsf{B}(0, \mathsf{E}(\mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{h} \mid \mathsf{i}(\mathsf{h}))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h}))) \\
&\xrightarrow{14} \mathsf{A}(\mathsf{s}(0), \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{h} \mid \mathsf{i}(\mathsf{h}))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h})) =_{\mathsf{AC}} \mathsf{A}(\mathsf{s}(0), H_1)
\end{aligned}
$$

It is important to note that the TRS $\mathcal{H}$ defined above is *unsorted* and we establish in this article the result that it is AC terminating on all terms. When simulating a battle, like in the statement of the Theorem 3.6, we deal with well-behaved terms adhering to the sort discipline introduced shortly. The restriction to sorted terms is crucial for our termination proof, but entails no loss of generality. This is due to the following result, which is a special case of [MO00, Corollary 3.9].

Theorem 3.8. *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 idea of using sorts to simplify termination proof goes back to Zantema [Zan94]. The TRS $\mathcal{H}$ can be seen as a TRS over the many-sorted signature $\mathcal{F}'$:

$$
\begin{array}{llll}
\mathsf{h} : \mathsf{O} & \mathsf{i}, \mathsf{E} : \mathsf{O} \to \mathsf{O} & \mid : \mathsf{O} \times \mathsf{O} \to \mathsf{O} & \mathsf{A}, \mathsf{B} : \mathsf{N} \times \mathsf{O} \to \mathsf{S} \\
0 : \mathsf{N} & \mathsf{s} : \mathsf{N} \to \mathsf{N} & & \mathsf{C}, \mathsf{D} : \mathsf{N} \times \mathsf{O} \to \mathsf{O}
\end{array}
$$

where N, O and S are sort symbols. Since $\mathcal{H}$ is non-collapsing, Theorem 3.8 guarantees that AC termination of $\mathcal{H}$ follows from AC termination of well-sorted terms over $\mathcal{F}'$.

In the remainder of this section we present a proof of Theorem 3.6 and its converse.

Lemma 3.9. *If $n > 0$ then $\mathsf{C}(\mathsf{n}, t) \to^*_{\mathcal{H}/\mathsf{AC}} t^n \mid \mathsf{E}(t)$ for all terms t.*

Proof. We use induction on n. If $n = 1$ then

$$\mathsf{C}(\mathsf{n}, t) \xrightarrow{5} t \mid \mathsf{C}(0, t) \xrightarrow{6} t \mid \mathsf{E}(t) = t^n \mid \mathsf{E}(t)$$

Suppose the result holds for $n \geqslant 1$ and consider $n + 1$. The induction hypothesis yields $\mathsf{C}(\mathsf{n}, t) \to^*_{\mathcal{H}/\mathsf{AC}} t^n \mid \mathsf{E}(t)$. Hence

$$\mathsf{C}(\mathsf{s}(\mathsf{n}), t) \xrightarrow{5} t \mid \mathsf{C}(\mathsf{n}, t) \to^*_{\mathcal{H}/\mathsf{AC}} t \mid (t^n \mid \mathsf{E}(t)) =_{\mathsf{AC}} t^{n+1} \mid \mathsf{E}(t)$$ □

Lemma 3.10. *Let n be a natural number. If $H \Rightarrow_n H'$ then $\mathsf{D}(\mathsf{s}(\mathsf{n}), H) \to^*_{\mathcal{H}/\mathsf{AC}} \mathsf{E}(H')$.*

Proof. We use structural induction on H and consider the following two cases.

- First suppose $H \to_{\mathcal{R}_n/\mathsf{AC}} H'$ is a root step. If the first rule of $\mathcal{R}_n$ is used then $H = \mathsf{i}(\mathsf{i}(\mathsf{h}))$ and $H' =_{\mathsf{AC}} \mathsf{i}(h^{n+2})$. We have $\mathsf{D}(\mathsf{s}(\mathsf{n}), H) \xrightarrow{13} \mathsf{i}(\mathsf{C}(\mathsf{s}(\mathsf{n}), \mathsf{h}))$. Using Lemma 3.9 we obtain

  $$\mathsf{i}(\mathsf{C}(\mathsf{s}(\mathsf{n}), \mathsf{h})) \to^*_{\mathcal{H}/\mathsf{AC}} \mathsf{i}(\mathsf{h}^{n+1} \mid \mathsf{E}(\mathsf{h})) =_{\mathsf{AC}} \cdot \xrightarrow{6} \mathsf{E}(\mathsf{i}(\mathsf{h} \mid \mathsf{h}^{n+1})) =_{\mathsf{AC}} \mathsf{E}(H')$$

  If the second rule of $\mathcal{R}_n$ is used then $H =_{\mathsf{AC}} \mathsf{i}(\mathsf{i}(\mathsf{h} \mid t))$ and $H' =_{\mathsf{AC}} \mathsf{i}(\mathsf{i}(t)^{n+2})$ for some term t. We have $\mathsf{D}(\mathsf{s}(\mathsf{n}), H) =_{\mathsf{AC}} \cdot \xrightarrow{11} \mathsf{i}(\mathsf{C}(\mathsf{s}(\mathsf{n}), \mathsf{i}(t)))$. Using Lemma 3.9 we obtain

  $$\mathsf{i}(\mathsf{C}(\mathsf{s}(\mathsf{n}), \mathsf{i}(t))) \to^*_{\mathcal{H}/\mathsf{AC}} \mathsf{i}(\mathsf{i}(t)^{n+1} \mid \mathsf{E}(\mathsf{i}(t))) =_{\mathsf{AC}} \cdot \xrightarrow{6} \mathsf{E}(\mathsf{i}(\mathsf{i}(t) \mid \mathsf{i}(t)^{n+1})) =_{\mathsf{AC}} \mathsf{E}(H')$$

  If the third rule of $\mathcal{R}_n$ is used then $H =_{\mathsf{AC}} \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid t)$ and $H' =_{\mathsf{AC}} \mathsf{i}(\mathsf{h}^{n+2} \mid t)$ for some term t. We have $\mathsf{D}(\mathsf{s}(\mathsf{n}), H) =_{\mathsf{AC}} \cdot \xrightarrow{12} \mathsf{i}(\mathsf{C}(\mathsf{s}(\mathsf{n}), \mathsf{h}) \mid t)$. The remaining argument is the same

as in the preceding cases. If the fourth rule of $\mathcal{R}_n$ is used then $H =_{\mathsf{AC}} \mathsf{i}(\mathsf{i}(\mathsf{h} \mid s) \mid t)$ and $H' =_{\mathsf{AC}} \mathsf{i}(\mathsf{i}(s)^{n+2} \mid t)$ for some terms s and t. Using Lemma 3.9 we obtain

$$\begin{aligned}\mathsf{D}(\mathsf{s}(\mathsf{n}), H) &=_{\mathsf{AC}} \cdot \xrightarrow{10} \mathsf{i}(\mathsf{C}(\mathsf{s}(\mathsf{n}), \mathsf{i}(s)) \mid t) \to^*_{\mathcal{H}/\mathsf{AC}} \mathsf{i}((\mathsf{i}(s)^{n+1} \mid \mathsf{E}(\mathsf{i}(s))) \mid t)\\ &=_{\mathsf{AC}} \cdot \xrightarrow{6} \mathsf{E}(\mathsf{i}(\mathsf{i}(s) \mid (\mathsf{i}(s)^{n+1} \mid t))) =_{\mathsf{AC}} \mathsf{E}(H')\end{aligned}$$

- Otherwise, $H =_{\mathsf{AC}} \mathsf{i}(H_1 \mid H_2 \mid \cdots \mid H_m)$ and $H' =_{\mathsf{AC}} \mathsf{i}(H_1' \mid H_2 \mid \cdots \mid H_m)$ for some $m \geqslant 1$ and Hydras $H_1, \ldots, H_m, H_1'$ with $H_1 \to_{\mathcal{R}_n/\mathsf{AC}} H_1'$. We obtain $\mathsf{D}(\mathsf{s}(\mathsf{n}), H_1) \to^*_{\mathcal{H}/\mathsf{AC}} \mathsf{E}(H_1')$ from the induction hypothesis. Note that $\mathsf{root}(H_1) = \mathsf{i}$. If $m = 1$ then

$$\begin{aligned}\mathsf{D}(\mathsf{s}(\mathsf{n}), H) &=_{\mathsf{AC}} \mathsf{D}(\mathsf{s}(\mathsf{n}), \mathsf{i}(H_1)) \xrightarrow{8} \mathsf{i}(\mathsf{D}(\mathsf{s}(\mathsf{n}), H_1)) \to^*_{\mathcal{H}/\mathsf{AC}} \mathsf{i}(\mathsf{E}(H_1')) \xrightarrow{7} \mathsf{E}(\mathsf{i}(H_1'))\\ &=_{\mathsf{AC}} \mathsf{E}(H')\end{aligned}$$

and if $m > 1$ we reach the same conclusion using rules 9 and 6 instead of 8 and 7. □

Proof of Theorem 3.6. Our task is to show

$$\mathsf{A}(\mathsf{n}, H) \to^*_{\mathcal{H}/\mathsf{AC}} \mathsf{A}(\mathsf{s}(\mathsf{n}), H')$$

If $H \Rightarrow_n H'$ is derived from condition (1) or (2) in Definition 3.3, the claim is immediate by rules 1 and 2 of $\mathcal{H}$. Otherwise, $H \to_{\mathcal{R}_n/\mathsf{AC}} H'$. This implies $\mathsf{root}(H) = \mathsf{i}$. Using rules 3 and 14 together with Lemma 3.10 yields

$$\mathsf{A}(\mathsf{n}, H) \xrightarrow{3} \mathsf{B}(\mathsf{n}, \mathsf{D}(\mathsf{s}(\mathsf{n}), H)) \to^*_{\mathcal{H}/\mathsf{AC}} \mathsf{B}(\mathsf{n}, \mathsf{E}(H')) \xrightarrow{14} \mathsf{A}(\mathsf{s}(\mathsf{n}), H')$$ □

In the remaining part of this section we prove the converse of Theorem 3.6.

Theorem 3.11. *Let $H, H' \in \mathcal{T}_{\mathcal{H}}$ be encodings of Hydras and let n be a natural number. If $\mathsf{A}(\mathsf{n}, H) \to^*_{\mathcal{H}/\mathsf{AC}} \mathsf{A}(\mathsf{s}(\mathsf{n}), H')$ then $H \Rightarrow_n H'$.*

In order to show the claim we need a few auxiliary lemmata. Let $\mathcal{C}_{\mathcal{H}}$ be the set of ground contexts over $\{\mathsf{h}, \mathsf{i}, \mid\}$.

Definition 3.12. We define U as the set consisting of all terms of the forms $\mathsf{A}(\mathsf{n}, t)$, $\mathsf{B}(\mathsf{n}, C[\mathsf{C}(\mathsf{m}, t)])$, $\mathsf{B}(\mathsf{n}, C[\mathsf{D}(\mathsf{s}(\mathsf{n}), t)])$, and $\mathsf{B}(\mathsf{n}, C[\mathsf{E}(t)])$, where $n, m \in \mathbb{N}$, $t \in \mathcal{T}_{\mathcal{H}}$, and $C \in \mathcal{C}_{\mathcal{H}}$.

The set U contains all terms reachable from $\mathsf{A}(\mathsf{n}, H)$.

Lemma 3.13. *If $t \in U$ and $t \to^*_{\mathcal{H} \cup \mathsf{AC}} u$ then $u \in U$.*

Proof. The claim is easily shown by induction on the length of $t \to^*_{\mathcal{H} \cup \mathsf{AC}} u$. □

In order to analyze the rewrite sequence $\mathsf{A}(\mathsf{n}, H) \to^*_{\mathcal{H}/\mathsf{AC}} \mathsf{A}(\mathsf{s}(\mathsf{n}), H')$ we define three subsets of $\mathcal{H}$: $\mathcal{H}_1 = \{1, 2\}$, $\mathcal{H}_2 = \{3\text{–}9, 14\}$, and $\mathcal{H}_3 = \{10\text{–}13\}$. The second rewrite sequence in Example 3.7 can then be described as follows:

$$\begin{aligned}\mathsf{A}(0, H_0) &\to^*_{\mathcal{H}_2/\mathsf{AC}} \mathsf{B}(0, \mathsf{i}(\mathsf{i}(\mathsf{D}(\mathsf{s}(0), \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{i}(\mathsf{h})))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h}))\\ &\to_{\mathcal{H}_3/\mathsf{AC}} \mathsf{B}(0, \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{C}(\mathsf{s}(0), \mathsf{h}) \mid \mathsf{i}(\mathsf{h}))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h}))\\ &\to^*_{\mathcal{H}_2/\mathsf{AC}} \mathsf{A}(1, H_1)\end{aligned}$$

Definition 3.14. We define V as the extension of U with $\mathcal{T}_\mathcal{H}$ and all terms of the forms $C[\mathsf{C}(\mathsf{n},t)]$ $C[\mathsf{D}(\mathsf{n},t)]$, and $C[\mathsf{E}(t)]$ where $n \in \mathbb{N}$, $t \in \mathcal{T}_\mathcal{H}$, and $C \in \mathcal{C}_\mathcal{H}$. The mapping $\pi\colon V \to \mathcal{T}_\mathcal{H}$ is defined as follows:

$$\pi(t) = \begin{cases} \mathsf{h} & \text{if } t = \mathsf{h} \\ \mathsf{i}(\pi(u)) & \text{if } t = \mathsf{i}(u) \\ \pi(u) \mid \pi(v) & \text{if } t = u \mid v \\ u & \text{if } t = \mathsf{A}(\mathsf{n},u) \text{ or } t = \mathsf{D}(\mathsf{n},u) \text{ or } t = \mathsf{E}(u) \\ \pi(u) & \text{if } t = \mathsf{B}(\mathsf{n},u) \\ u^{n+1} & \text{if } t = \mathsf{C}(\mathsf{n},u) \end{cases}$$

Taking the role of C into account, the mapping π computes the Hydra in a given term. Applying π to the terms in the above rewrite sequence of $\mathcal{H}_2/\mathsf{AC}$ and $\mathcal{H}_3/\mathsf{AC}$, we obtain

$$\begin{aligned} H_0 = \pi(\mathsf{A}(\mathsf{0},H_0)) &=_\mathsf{AC} \pi(\mathsf{B}(\mathsf{0},\mathsf{i}(\mathsf{i}(\mathsf{D}(\mathsf{s}(\mathsf{0}),\mathsf{i}(\mathsf{i}(\mathsf{h}) \mid \mathsf{i}(\mathsf{h})))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h}))) \\ &\to_{\mathcal{R}_0/\mathsf{AC}} \pi(\mathsf{B}(\mathsf{0},\mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{C}(\mathsf{s}(\mathsf{0}),\mathsf{h}) \mid \mathsf{i}(\mathsf{h}))) \mid \mathsf{i}(\mathsf{h}) \mid \mathsf{h}))) \\ &=_\mathsf{AC} \pi(\mathsf{A}(\mathsf{1},H_1)) = H_1 \end{aligned}$$

This verifies that H_1 is a successor of H_0.

Lemma 3.15. *The following properties hold.*

(1) $\pi(t) = t$ *for all terms* $t \in \mathcal{T}_\mathcal{H}$,
(2) $\pi(C[t]) = C[\pi(t)]$ *for all terms* $t \in V$ *and contexts* $C \in \mathcal{C}_\mathcal{H}$,
(3) $\pi(C[t]) =_\mathsf{AC} \pi(D[u])$ *for all terms* $t,u \in \mathcal{T}_\mathcal{H}$ *and contexts* $C,D \in \mathcal{C}_\mathcal{H}$ *with* $t =_\mathsf{AC} u$ *and* $C =_\mathsf{AC} D$.

Proof. The first statement is proved by induction on $t \in \mathcal{T}_\mathcal{H}$. If $t = \mathsf{h}$ then $\pi(t) = \mathsf{h} = t$. If $t = \mathsf{i}(u)$ with $u \in \mathcal{T}_\mathcal{H}$ then $\pi(t) = \mathsf{i}(\pi(u)) = \mathsf{i}(u) = t$. If $t = u \mid v$ with $u,v \in \mathcal{T}_\mathcal{H}$ then $\pi(t) = \pi(u) \mid \pi(v) = u \mid v = t$. For the second statement we use induction on the context $C \in \mathcal{C}_\mathcal{H}$. If $C = \Box$ then $\pi(C[t]) = \pi(t) = C[\pi(t)]$. If $C = \mathsf{i}(D)$ then $\pi(C[t]) = \mathsf{i}(\pi(D[t])) = \mathsf{i}(D[\pi(t)]) = C[\pi(t)]$. If $C = D \mid u$ then $D \in \mathcal{C}_\mathcal{H}$ and $u \in \mathcal{T}_\mathcal{H}$ and thus $\pi(C[t]) = \pi(D[t]) \mid \pi(u) = D[\pi(t)] \mid u = C[\pi(t)]$. If $C = u \mid D$ then $D \in \mathcal{C}_\mathcal{H}$ and $u \in \mathcal{T}_\mathcal{H}$ and thus $\pi(C[t]) = \pi(u) \mid \pi(D[t]) = u \mid D[\pi(t)] = C[\pi(t)]$. The third statement follows from statements (1) and (2): $\pi(C[t]) = C[\pi(t)] = C[t] =_\mathsf{AC} D[t] =_\mathsf{AC} D[u] = D[\pi(u)] = \pi(D[u])$. □

The following lemma relates AC rewriting of $\mathcal{H}$ to rewriting of Hydras according to Definition 3.3.

Lemma 3.16. *The following statements hold for all terms* $s,t \in U$.

(1) *If* $s =_\mathsf{AC} t$ *then* $\pi(s) =_\mathsf{AC} \pi(t)$.
(2) *If* $s \to_{\mathcal{H}_2} t$ *then* $\pi(s) =_\mathsf{AC} \pi(t)$.
(3) *If* $s \to_{\mathcal{H}_3} t$ *then* $\pi(s) \to_{\mathcal{R}_n/\mathsf{AC}} \pi(t)$ *with* $s = \mathsf{B}(\mathsf{n},s')$ *for some* $n \geqslant 0$.

Proof. Let $s,t \in U$.

(1) If $s = \mathsf{A}(\mathsf{n},u)$ with $u \in \mathcal{T}_\mathcal{H}$ then $t = \mathsf{A}(\mathsf{n},v)$ for some term $v \in \mathcal{T}_\mathcal{H}$ with $u =_\mathsf{AC} v$. Since $\pi(s) = u$ and $\pi(t) = v$, $\pi(s) =_\mathsf{AC} \pi(t)$ follows. If $s = \mathsf{B}(\mathsf{n},C[\mathsf{C}(\mathsf{m},u)])$ with $n,m \in \mathbb{N}$, $C \in \mathcal{C}_\mathcal{H}$ and $u \in \mathcal{T}_\mathcal{H}$ then $t = \mathsf{B}(\mathsf{n},D[\mathsf{C}(\mathsf{m},v)])$ with $C =_\mathsf{AC} D$ and $u =_\mathsf{AC} v$. Using Lemma 3.15(1,2) we obtain $\pi(s) = \pi(C[\mathsf{C}(\mathsf{m},u)]) = C[\pi(\mathsf{C}(\mathsf{m},u))] = C[u^{m+1}]$ and $\pi(t) = D[v^{m+1}]$. From $u =_\mathsf{AC} v$ we infer $u^{m+1} =_\mathsf{AC} v^{m+1}$ and thus $\pi(s) =_\mathsf{AC} \pi(t)$ by

Lemma 3.15(3). The cases $s = \mathsf{B}(\mathsf{n}, C[\mathsf{D}(\mathsf{s}(\mathsf{n}), u)])$ and $s = \mathsf{B}(\mathsf{n}, C[\mathsf{E}(u)])$ are treated in the same way.

(2) For the second statement we make a case analysis based on the employed rule in $\mathcal{H}_2$.
- If $s \xrightarrow{3} t$ then $s = \mathsf{A}(\mathsf{n}, \mathsf{i}(u))$ and $t = \mathsf{B}(\mathsf{n}, \mathsf{D}(\mathsf{s}(\mathsf{n}), \mathsf{i}(u)))$ for some $n \geqslant 0$ and $u \in \mathcal{T}_{\mathcal{H}}$. We have $\pi(s) = \mathsf{i}(u) = \pi(\mathsf{D}(\mathsf{s}(\mathsf{n}), \mathsf{i}(u))) = \pi(t)$ by the definition of t.
- If $s \xrightarrow{4} t$ then $s = \mathsf{B}(\mathsf{n}, C[\mathsf{C}(\mathsf{0}, u)])$ and $t = \mathsf{B}(\mathsf{n}, C[\mathsf{E}(u)])$ for some $n \geqslant 0$, $C \in \mathcal{C}_{\mathcal{H}}$ and $u \in \mathcal{T}_{\mathcal{H}}$. We have $\pi(s) = \pi(C[\mathsf{C}(\mathsf{0}, u)]) = C[u^1] = C[u] = \pi(C[u]) = \pi(t)$.
- If $s \xrightarrow{5} t$ then $s = \mathsf{B}(\mathsf{n}, C[\mathsf{C}(\mathsf{s}(\mathsf{m}), u)])$ and $t = \mathsf{B}(\mathsf{n}, C[u \mid \mathsf{C}(\mathsf{m}, u)])$ for some $n \geqslant 0$, $m \geqslant 0$, $C \in \mathcal{C}_{\mathcal{H}}$ and $u \in \mathcal{T}_{\mathcal{H}}$. We have $\pi(s) = C[u^{m+2}] =_{\mathsf{AC}} C[u \mid u^{m+1}] = C[\pi(u \mid u^{m+1})] = \pi(C[u \mid \mathsf{C}(\mathsf{m}, u)]) = \pi(t)$.
- If $s \xrightarrow{6} t$ then $s = \mathsf{B}(\mathsf{n}, C[\mathsf{i}(\mathsf{E}(u)|v)])$ and $t = \mathsf{B}(\mathsf{n}, C[\mathsf{E}(\mathsf{i}(u|v))])$ for some $n \geqslant 0$, $C \in \mathcal{C}_{\mathcal{H}}$ and $u, v \in \mathcal{T}_{\mathcal{H}}$. We have $\pi(s) = \pi(C[\mathsf{i}(\mathsf{E}(u) \mid v)]) = C[\mathsf{i}(u \mid v)] = \pi(C[\mathsf{E}(\mathsf{i}(u \mid v))]) = \pi(t)$.
- If $s \xrightarrow{7} t$ then $s = \mathsf{B}(\mathsf{n}, C[\mathsf{i}(\mathsf{E}(u))])$ and $t = \mathsf{B}(\mathsf{n}, C[\mathsf{E}(\mathsf{i}(u))])$ for some $n \geqslant 0$, $C \in \mathcal{C}_{\mathcal{H}}$ and $u \in \mathcal{T}_{\mathcal{H}}$. We have $\pi(s) = \pi(C[\mathsf{i}(\mathsf{E}(u))]) = C[\mathsf{i}(u)] = \pi(C[\mathsf{E}(\mathsf{i}(u))]) = \pi(t)$.
- If $s \xrightarrow{8} t$ then $s = \mathsf{B}(\mathsf{n}, C[\mathsf{D}(\mathsf{s}(\mathsf{n}), \mathsf{i}(\mathsf{i}(u)))])$ and $t = \mathsf{B}(\mathsf{n}, C[\mathsf{i}(\mathsf{D}(\mathsf{s}(\mathsf{n}), \mathsf{i}(u)))])$ for some $n \geqslant 0$, $C \in \mathcal{C}_{\mathcal{H}}$ and $u \in \mathcal{T}_{\mathcal{H}}$. We have $\pi(s) = C[\mathsf{i}(\mathsf{i}(u))] = \pi(t)$.
- If $s \xrightarrow{9} t$ then $s = \mathsf{B}(\mathsf{n}, C[\mathsf{D}(\mathsf{s}(\mathsf{n}), \mathsf{i}(\mathsf{i}(u) \mid v))])$ and $t = \mathsf{B}(\mathsf{n}, C[\mathsf{i}(\mathsf{D}(\mathsf{s}(\mathsf{n}), \mathsf{i}(u)) \mid v)])$ for some $n \geqslant 0$, $C \in \mathcal{C}_{\mathcal{H}}$ and $u, v \in \mathcal{T}_{\mathcal{H}}$. In this case we obtain $\pi(s) = C[\mathsf{i}(\mathsf{i}(u) \mid v)] = \pi(t)$.
- If $s \xrightarrow{14} t$ then $s = \mathsf{B}(\mathsf{n}, \mathsf{E}(u))$ and $t = \mathsf{A}(\mathsf{s}(\mathsf{n}), u)$ for some $n \geqslant 0$ and $u \in \mathcal{T}_{\mathcal{H}}$. In this case we have $\pi(s) = \pi(\mathsf{E}(u)) = u = \pi(t)$.

(3) Again we make a case analysis on the applied rewrite rule.
- If $s \xrightarrow{10} t$ then $s = \mathsf{B}(\mathsf{n}, C[\mathsf{D}(\mathsf{s}(\mathsf{n}), \mathsf{i}(\mathsf{i}(\mathsf{h} \mid u) \mid v))])$ and $t = \mathsf{B}(\mathsf{n}, C[\mathsf{i}(\mathsf{C}(\mathsf{s}(\mathsf{n}), \mathsf{i}(u)) \mid v)])$ for some $n \geqslant 0$, $C \in \mathcal{C}_{\mathcal{H}}$ and $u, v \in \mathcal{T}_{\mathcal{H}}$. We obtain $\pi(s) = C[\mathsf{i}(\mathsf{i}(\mathsf{h} \mid u) \mid v)]$ and $\pi(t) = C[\mathsf{i}(\mathsf{i}(u)^{n+2} \mid v)]$. Hence $\pi(s) \to_{\mathcal{R}_n} \pi(t)$ by applying rule 4 of $\mathcal{R}_n$.
- If $s \xrightarrow{11} t$ then $s = \mathsf{B}(\mathsf{n}, C[\mathsf{D}(\mathsf{s}(\mathsf{n}), \mathsf{i}(\mathsf{i}(\mathsf{h} \mid u)))])$ and $t = \mathsf{B}(\mathsf{n}, C[\mathsf{i}(\mathsf{C}(\mathsf{s}(\mathsf{n}), \mathsf{i}(u)))])$ for some $n \geqslant 0$, $C \in \mathcal{C}_{\mathcal{H}}$ and $u, v \in \mathcal{T}_{\mathcal{H}}$. We obtain $\pi(s) = C[\mathsf{i}(\mathsf{i}(\mathsf{h} \mid u))]$ and $\pi(t) = C[\mathsf{i}(\mathsf{i}(u)^{n+2})]$. Hence $\pi(s) \to_{\mathcal{R}_n} \pi(t)$ by applying rule 2 of $\mathcal{R}_n$.
- If $s \xrightarrow{12} t$ then $s = \mathsf{B}(\mathsf{n}, C[\mathsf{D}(\mathsf{s}(\mathsf{n}), \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid v))])$ and $t = \mathsf{B}(\mathsf{n}, C[\mathsf{i}(\mathsf{C}(\mathsf{s}(\mathsf{n}), \mathsf{h}) \mid v)])$ for some $n \geqslant 0$, $C \in \mathcal{C}_{\mathcal{H}}$ and $v \in \mathcal{T}_{\mathcal{H}}$. We obtain $\pi(s) = C[\mathsf{i}(\mathsf{i}(\mathsf{h}) \mid v)]$ and $\pi(t) = C[\mathsf{i}(\mathsf{h}^{n+2} \mid v)]$. Hence $\pi(s) \to_{\mathcal{R}_n} \pi(t)$ by applying rule 3 of $\mathcal{R}_n$.
- If $s \xrightarrow{13} t$ then $s = \mathsf{B}(\mathsf{n}, C[\mathsf{D}(\mathsf{s}(\mathsf{n}), \mathsf{i}(\mathsf{i}(\mathsf{h})))])$ and $t = \mathsf{B}(\mathsf{n}, C[\mathsf{i}(\mathsf{C}(\mathsf{s}(\mathsf{n}), \mathsf{h}))])$ for some $n \geqslant 0$ and $C \in \mathcal{C}_{\mathcal{H}}$. We obtain $\pi(s) = C[\mathsf{i}(\mathsf{i}(\mathsf{h}))]$ and $\pi(t) = C[\mathsf{i}(\mathsf{h}^{n+2})]$. Hence $\pi(s) \to_{\mathcal{R}_n} \pi(t)$ by applying rule 1 of $\mathcal{R}_n$. □

So we are ready to prove the main claim.

Proof of Theorem 3.11. Suppose $s = \mathsf{A}(\mathsf{n}, H) \to^+_{\mathcal{H}/\mathsf{AC}} \mathsf{A}(\mathsf{s}(\mathsf{n}), H') = t$. Inspection of $\mathcal{H}$ reveals that one of the following two cases holds:

(a) $s \to_{\mathcal{H}_1/\mathsf{AC}} t$, or

(b) $s \to^*_{\mathcal{H}_2/\mathsf{AC}} \cdot \to_{\mathcal{H}_3/\mathsf{AC}} \cdot \to^*_{\mathcal{H}_2/\mathsf{AC}} t$.

We first consider (a). If $s \to_{\mathcal{H}_1/\mathsf{AC}} t$ is a root step using rule 1 then $H = \mathsf{i}(\mathsf{h})$ and $H' = \mathsf{h}$. If $s \to_{\mathcal{H}_1/\mathsf{AC}} t$ is a root step using rule 2 then $H =_{\mathsf{AC}} \mathsf{i}(\mathsf{h} \mid u)$ and $H' =_{\mathsf{AC}} \mathsf{i}(u)$ for some term u. Next we consider (b). We have $s \to^*_{\mathcal{H}_2/\mathsf{AC}} s' \to_{\mathcal{H}_3/\mathsf{AC}} t' \to^*_{\mathcal{H}_2/\mathsf{AC}} t$ for some s' and t'. From

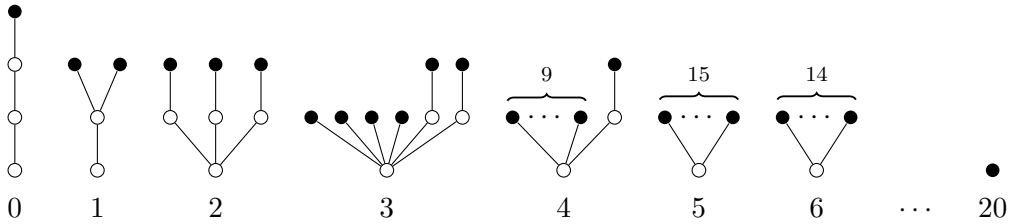
Lemma 3.13 we obtain $s, s', t', t \in U$. Hence

$$H = \pi(s) =_{\mathsf{AC}} \pi(s') \to_{\mathcal{R}_n/\mathsf{AC}} \pi(t') =_{\mathsf{AC}} \pi(t) = H'$$

is obtained by Lemma 3.16 and thus $H \to_{\mathcal{R}_n/\mathsf{AC}} H'$. Hence, we conclude $H \Rightarrow_n H'$. □

4. Many-Sorted Semantic Labeling modulo AC

Kirby and Paris [KP82] proved the termination of the Hydra Battle by associating ordinal numbers to Hydras (see [KM76, DM07] for notions and notations for ordinal numbers). Consider, for example, the following fight with the Hydra in Example 3.7:



By interpreting h, i, and $|$ as 1, the power of ω, and natural addition on ordinals, respectively, the sequence of Hydras turns into the decreasing sequence of ordinals:

$$\omega^{\omega^\omega} > \omega^{\omega^2} > \omega^{\omega\cdot 3} > \omega^{\omega\cdot 2+4} > \omega^{\omega+9} > \omega^{15} > \omega^{14} > \cdots > 1$$

One can verify that in general every transition reduces the ordinal interpretation of the Hydra. Because the order $>$ on ordinals is well-founded, the termination is concluded.

In the case of the term rewriting encoding, the mutual dependence between the function symbols A and B in rules 3 and 14 of $\mathcal{H}$ makes proving termination of $\mathcal{H}/\mathsf{AC}$ a non-trivial task. We use the technique of semantic labeling (Zantema [Zan95]) to resolve the dependence by labeling both A and B by the ordinal value of the Hydra encoded in their second arguments. Semantic labeling for rewriting modulo has been investigated in [OMG00]. We need, however, a version for many-sorted rewriting since the distinction between ordinals and natural numbers is essential for the effectiveness of semantic labeling for $\mathcal{H}/\mathsf{AC}$.

Before introducing semantic labeling, we recall some basic semantic definitions. An *algebra* $\mathcal{A}$ for an $\mathcal{S}$-sorted signature $\mathcal{F}$ is a pair $(\{S_\mathcal{A}\}_{S\in\mathcal{S}}, \{f_\mathcal{A}\}_{f\in\mathcal{F}})$, where each $S_\mathcal{A}$ is a non-empty set, called the *carrier of sort* S, and each $f_\mathcal{A}$ is a function of type $f : (S_1)_\mathcal{A} \times \cdots \times (S_n)_\mathcal{A} \to S_\mathcal{A}$, called the *interpretation function* of $f : S_1 \times \cdots \times S_n \to S$. A mapping that associates each variable of sort S to an element in $S_\mathcal{A}$ is called an *assignment*. We write $\mathcal{A}^\mathcal{V}$ for the set of all assignments. Given an assignment $\alpha \in \mathcal{A}^\mathcal{V}$, the *interpretation* of a term t is inductively defined as follows:

$$[\alpha]_\mathcal{A}(t) = \begin{cases} \alpha(t) & \text{if } t \text{ is a variable} \\ f_\mathcal{A}([\alpha]_\mathcal{A}(t_1), \ldots, [\alpha]_\mathcal{A}(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

Let $\mathcal{A} = (\{S_\mathcal{A}\}_{S\in\mathcal{S}}, \{f_\mathcal{A}\}_{f\in\mathcal{F}})$ be an $\mathcal{S}$-sorted $\mathcal{F}$-algebra. We assume that each carrier set $S_\mathcal{A}$ is equipped with a well-founded order $>_S$ such that the interpretation functions are weakly monotone. Here a function ϕ of type $A_1 \times \cdots \times A_n \to B$ is *weakly monotone* if $\phi(a_1, \ldots, a_i, \ldots, a_n) \geqslant_B \phi(a_1, \ldots, b, \ldots, a_n)$ whenever $a_i \geqslant_{A_i} b$. We call $(\mathcal{A}, \{>_S\}_{S\in\mathcal{S}})$ a weakly monotone many-sorted algebra. Given terms s and t of sort S, we write $s \geqslant_\mathcal{A} t$ ($s =_\mathcal{A} t$) if $[\alpha]_\mathcal{A}(s) \geqslant_S [\alpha]_\mathcal{A}(t)$ ($[\alpha]_\mathcal{A}(s) =_S [\alpha]_\mathcal{A}(t)$) holds for all $\alpha \in \mathcal{A}^\mathcal{V}$.

A labeling L for $\mathcal{F}$ consists of sets of labels $L_f \subseteq S_{\mathcal{A}}$ for every $f : S_1 \times \cdots \times S_n \to S$. The labeled signature $\mathcal{F}_{\mathsf{lab}}$ consists of function symbols $f_a : S_1 \times \cdots \times S_n \to S$ for every function symbol $f : S_1 \times \cdots \times S_n \to S$ in $\mathcal{F}$ and label $a \in L_f$ together with all function symbols $f \in \mathcal{F}$ such that $L_f = \varnothing$. A *labeling* (L, lab) for $(\mathcal{A}, \{>_S\}_{S\in\mathcal{S}})$ consists of a labeling L for the signature $\mathcal{F}$ together with a mapping $\mathsf{lab}_f \colon (S_1)_{\mathcal{A}} \times \cdots \times (S_n)_{\mathcal{A}} \to L_f$ for every function symbol $f : S_1 \times \cdots \times S_n \to S$ in $\mathcal{F}$ with $L_f \neq \varnothing$. We call (L, lab) *weakly monotone* if all its labeling functions lab_f are weakly monotone. The mapping lab_f determines the label of the root symbol f of a term $f(t_1, \ldots, t_n)$, based on the values of its arguments $t_1, \ldots, t_n$. Formally, for every assignment $\alpha \in \mathcal{A}^{\mathcal{V}}$ we define a mapping lab_α inductively as follows:

$$\mathsf{lab}_\alpha(t) = \begin{cases} t & \text{if } t \in \mathcal{V} \\ f(\mathsf{lab}_\alpha(t_1), \ldots, \mathsf{lab}_\alpha(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } L_f = \varnothing \\ f_a(\mathsf{lab}_\alpha(t_1), \ldots, \mathsf{lab}_\alpha(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } L_f \neq \varnothing \end{cases}$$

where $a = \mathsf{lab}_f([\alpha]_{\mathcal{A}}(t_1), \ldots, [\alpha]_{\mathcal{A}}(t_n))$. Note that $\mathsf{lab}_\alpha(t)$ and t have the same sort. Given a TRS $\mathcal{R}$ over a (many-sorted) signature $\mathcal{F}$, we define the *labeled* TRS $\mathcal{R}_{\mathsf{lab}}$ over the signature $\mathcal{F}_{\mathsf{lab}}$ as follows:

$$\mathcal{R}_{\mathsf{lab}} = \{\mathsf{lab}_\alpha(\ell) \to \mathsf{lab}_\alpha(r) \mid \ell \to r \in \mathcal{R} \text{ and } \alpha \in A^{\mathcal{V}}\}$$

Since there is no need to label the AC symbol $|$ in the encoding of the Hydra Battle, we assume for simplicity that $L_f = \varnothing$ for every AC symbol $f \in \mathcal{F}$. The TRS $\mathcal{D}\mathsf{ec}$ consists of all rewrite rules

$$f_a(x_1, \ldots, x_n) \to f_b(x_1, \ldots, x_n)$$

with $f : S_1 \times \cdots \times S_n \to S$ a function symbol in $\mathcal{F}$, $a, b \in L_f$ such that $a >_S b$, and pairwise different variables $x_1, \ldots, x_n$. A weakly monotone algebra $(\mathcal{A}, >)$ is a *quasi-model* of $\mathcal{R}/\mathsf{AC}$ if $\ell \geqslant_{\mathcal{A}} r$ for all rewrite rules $\ell \to r$ in $\mathcal{R}$ and $\ell =_{\mathcal{A}} r$ for all equations $\ell \approx r$ in AC. So in a quasi-model, AC symbols are interpreted as associative and commutative functions.

Theorem 4.1. *Let $\mathcal{R}/\mathsf{AC}$ be a* TRS *over a many-sorted signature $\mathcal{F}$, $(\mathcal{A}, \{>_S\}_{S\in\mathcal{S}})$ a quasi-model of $\mathcal{R}/\mathsf{AC}$ with a weakly monotone labeling (L, lab). If $(\mathcal{R}_{\mathsf{lab}} \cup \mathcal{D}\mathsf{ec})/\mathsf{AC}$ is terminating then $\mathcal{R}/\mathsf{AC}$ is terminating.*

Proof. We show

(1) if $t \to_{\mathcal{R}} u$ then $\mathsf{lab}_\alpha(t) \to^*_{\mathcal{D}\mathsf{ec}} \cdot \to_{\mathcal{R}_{\mathsf{lab}}} \mathsf{lab}_\alpha(u)$

(2) if $t \to_{\mathsf{AC}} u$ then $\mathsf{lab}_\alpha(t) =_{\mathsf{AC}} \mathsf{lab}_\alpha(u)$

(3) if $t =_{\mathsf{AC}} u$ then $\mathsf{lab}_\alpha(t) =_{\mathsf{AC}} \mathsf{lab}_\alpha(u)$

for all sorts S, terms $t, u \in \mathcal{T}_S(\mathcal{F}, \mathcal{V})$, and assignments $\alpha \in \mathcal{A}^{\mathcal{V}}$. The claim follows from the first and third statements. First suppose $t \to_{\mathcal{R}} u$ is a root step using the rewrite rule $\ell \to r$. So $t = \ell\sigma$ and $u = r\sigma$ for some substitution σ. Define the assignment $\beta = [\alpha]_{\mathcal{A}} \circ \sigma$ and the (labeled) substitution $\tau = \mathsf{lab}_\alpha \circ \sigma$. An easy induction proof yields $\mathsf{lab}_\alpha(s\sigma) = \mathsf{lab}_\beta(s)\tau$ for all terms s. By definition $\mathsf{lab}_\beta(\ell) \to \mathsf{lab}_\beta(r) \in \mathcal{R}_{\mathsf{lab}}$. Hence $\mathsf{lab}_\alpha(t) = \mathsf{lab}_\beta(\ell)\tau \to_{\mathcal{R}_{\mathsf{lab}}} \mathsf{lab}_\beta(r)\tau = \mathsf{lab}_\alpha(u)$. Next suppose $t \to_{\mathcal{R}} u$ takes place below the root. So $t = f(t_1, \ldots, t_i, \ldots t_n)$ and $u = f(t_1, \ldots, u_i, \ldots t_n)$ with $t_i \to_{\mathcal{R}} u_i$. Let $S_1 \times \cdots \times S_n \to S$ be the sort declaration of f. The induction hypothesis yields $\mathsf{lab}_\alpha(t_i) \to^*_{\mathcal{D}\mathsf{ec}} \cdot \to_{\mathcal{R}_{\mathsf{lab}}} \mathsf{lab}_\alpha(u_i)$.

We obtain $[\alpha]_{\mathcal{A}}(t_i) \geqslant_{S_i} [\alpha]_{\mathcal{A}}(u_i)$ from the quasi-model assumption. If $L_f = \varnothing$ then

$$\begin{aligned}\mathsf{lab}_\alpha(t) = f(\mathsf{lab}_\alpha(t_1), \dots, \mathsf{lab}_\alpha(t_i), \dots, \mathsf{lab}_\alpha(t_n)) &\to^*_{\mathcal{D}\mathsf{ec}} \cdot \to_{\mathcal{R}_{\mathsf{lab}}} \\ f(\mathsf{lab}_\alpha(t_1), \dots, \mathsf{lab}_\alpha(u_i), \dots, \mathsf{lab}_\alpha(t_n)) &= \mathsf{lab}_\alpha(u)\end{aligned}$$

Suppose $L_f \neq \varnothing$ and let

$$\begin{aligned}a &= \mathsf{lab}_f([\alpha]_{\mathcal{A}}(t_1), \dots, [\alpha]_{\mathcal{A}}(t_i), \dots, [\alpha]_{\mathcal{A}}(t_n)) \\ b &= \mathsf{lab}_f([\alpha]_{\mathcal{A}}(t_1), \dots, [\alpha]_{\mathcal{A}}(u_i), \dots, [\alpha]_{\mathcal{A}}(t_n))\end{aligned}$$

We obtain $a \geqslant_S b$ from the weak monotonicity of the labeling function lab_f. Therefore, the following rewrite sequence is constructed:

$$\begin{aligned}\mathsf{lab}_\alpha(t) = f_a(\mathsf{lab}_\alpha(t_1), \dots, \mathsf{lab}_\alpha(t_i), \dots, \mathsf{lab}_\alpha(t_n)) &\to^*_{\mathcal{D}\mathsf{ec}} \\ f_b(\mathsf{lab}_\alpha(t_1), \dots, \mathsf{lab}_\alpha(t_i), \dots, \mathsf{lab}_\alpha(t_n)) &\to^*_{\mathcal{D}\mathsf{ec}} \cdot \to_{\mathcal{R}_{\mathsf{lab}}} \\ f_b(\mathsf{lab}_\alpha(t_1), \dots, \mathsf{lab}_\alpha(u_i), \dots, \mathsf{lab}_\alpha(t_n)) &= \mathsf{lab}_\alpha(u)\end{aligned}$$

This concludes the proof of the first statement. The second statement is shown in the same way, but since AC symbols are not labeled the rules of $\mathcal{D}\mathsf{ec}$ do not come into play. The third statement is obtained from the the second statement together with the fact that $t =_{\mathsf{AC}} u$ implies $t \to^*_{\mathsf{AC}} u$. □

After these preliminaries, we are ready to put many-sorted semantic labeling to the test. Consider the many-sorted algebra $\mathcal{A}$ with carriers $\mathbb{N}$ for sort N and $\mathbb{O}$, the set of ordinal numbers smaller than ϵ_0, for sorts O and S and the following interpretation functions:

$$\begin{aligned}0_{\mathcal{A}} = \mathsf{h}_{\mathcal{A}} = 1 \qquad \mathsf{s}_{\mathcal{A}}(n) &= n+1 \qquad \mathsf{i}_{\mathcal{A}}(x) = \omega^x \\ x \mid_{\mathcal{A}} y = x \oplus y \qquad \mathsf{E}_{\mathcal{A}}(x) &= x+1 \qquad \mathsf{C}_{\mathcal{A}}(n,x) = x \otimes n + 1 \\ \mathsf{A}_{\mathcal{A}}(n,x) = \mathsf{B}_{\mathcal{A}}(n,x) &= \mathsf{D}_{\mathcal{A}}(n,x) = x\end{aligned}$$

Here $\oplus$ denotes natural addition on ordinals and $\otimes$ denotes natural product characterized by $x \otimes 0 = 0$ and $x \otimes (n+1) = (x \otimes n) \oplus x$. Both satisfy strict monotonicity.

Lemma 4.2. *The algebra $(\mathcal{A}, \{>_{\mathsf{O}}, >_{\mathsf{N}}\})$ is a quasi-model of $\mathcal{H}/\mathsf{AC}$.*

Proof. First note that the interpretation functions are weakly monotone. The rewrite rules in $\mathcal{H}$ are oriented by $\geqslant_{\mathsf{O}}$:

$$\mathsf{A}_{\mathcal{A}}(n, \mathsf{i}_{\mathcal{A}}(\mathsf{h}_{\mathcal{A}})) = \omega \;>_{\mathsf{O}}\; 1 = \mathsf{A}_{\mathcal{A}}(\mathsf{s}_{\mathcal{A}}(n), \mathsf{h}_{\mathcal{A}}) \tag{1}$$

$$\mathsf{A}_{\mathcal{A}}(n, \mathsf{i}_{\mathcal{A}}(\mathsf{h}_{\mathcal{A}} \mid_{\mathcal{A}} x)) = \omega^{x+1} \;>_{\mathsf{O}}\; \omega^x = \mathsf{A}_{\mathcal{A}}(\mathsf{s}_{\mathcal{A}}(n), \mathsf{i}_{\mathcal{A}}(x)) \tag{2}$$

$$\mathsf{A}_{\mathcal{A}}(n, \mathsf{i}_{\mathcal{A}}(x)) = \omega^x \;=_{\mathsf{O}}\; \omega^x = \mathsf{B}_{\mathcal{A}}(n, \mathsf{D}_{\mathcal{A}}(\mathsf{s}_{\mathcal{A}}(n), \mathsf{i}_{\mathcal{A}}(x))) \tag{3}$$

$$\mathsf{C}_{\mathcal{A}}(0_{\mathcal{A}}, x) = x+1 \;=_{\mathsf{O}}\; x+1 = \mathsf{E}_{\mathcal{A}}(x) \tag{4}$$

$$\mathsf{C}_{\mathcal{A}}(\mathsf{s}_{\mathcal{A}}(n), x) = (x \otimes n) \oplus x + 1 \;=_{\mathsf{O}}\; (x \otimes n) \oplus x + 1 = x \mid_{\mathcal{A}} \mathsf{C}_{\mathcal{A}}(n, x) \tag{5}$$

$$\mathsf{i}_{\mathcal{A}}(\mathsf{E}_{\mathcal{A}}(x) \mid_{\mathcal{A}} y) = \omega^{x \oplus y + 1} \;>_{\mathsf{O}}\; \omega^{x \oplus y} + 1 = \mathsf{E}_{\mathcal{A}}(\mathsf{i}_{\mathcal{A}}(x \mid_{\mathcal{A}} y)) \tag{6}$$

$$\mathsf{i}_{\mathcal{A}}(\mathsf{E}_{\mathcal{A}}(x)) = \omega^{x+1} \;>_{\mathsf{O}}\; \omega^x + 1 = \mathsf{E}_{\mathcal{A}}(\mathsf{i}_{\mathcal{A}}(x)) \tag{7}$$

$$\mathsf{D}_{\mathcal{A}}(n, \mathsf{i}_{\mathcal{A}}(\mathsf{i}_{\mathcal{A}}(x))) = \omega^{\omega^x} \;=_{\mathsf{O}}\; \omega^{\omega^x} = \mathsf{i}_{\mathcal{A}}(\mathsf{D}_{\mathcal{A}}(n, \mathsf{i}_{\mathcal{A}}(x))) \tag{8}$$

$$\mathsf{D}_{\mathcal{A}}(n, \mathsf{i}_{\mathcal{A}}(\mathsf{i}_{\mathcal{A}}(x) \mid_{\mathcal{A}} y)) = \omega^{\omega^x \oplus y} \;=_{\mathsf{O}}\; \omega^{\omega^x \oplus y} = \mathsf{i}_{\mathcal{A}}(\mathsf{D}_{\mathcal{A}}(n, \mathsf{i}_{\mathcal{A}}(x)) \mid_{\mathcal{A}} y) \tag{9}$$

$$\mathsf{D}_{\mathcal{A}}(n, \mathsf{i}_{\mathcal{A}}(\mathsf{i}_{\mathcal{A}}(\mathsf{h}_{\mathcal{A}} \mid_{\mathcal{A}} x) \mid_{\mathcal{A}} y)) = \omega^{\omega^{x+1} \oplus y} \;>_{\mathsf{O}}\; \omega^{(\omega^x \otimes n) \oplus y + 1} = \mathsf{i}_{\mathcal{A}}(\mathsf{C}_{\mathcal{A}}(n, \mathsf{i}_{\mathcal{A}}(x)) \mid_{\mathcal{A}} y) \tag{10}$$

$$\mathsf{D}_{\mathcal{A}}(n, \mathsf{i}_{\mathcal{A}}(\mathsf{i}_{\mathcal{A}}(\mathsf{h}_{\mathcal{A}} \mid_{\mathcal{A}} x))) = \omega^{\omega^{x+1}} \;>_{\mathsf{O}}\; \omega^{(\omega^x \otimes n) + 1} = \mathsf{i}_{\mathcal{A}}(\mathsf{C}_{\mathcal{A}}(n, \mathsf{i}_{\mathcal{A}}(x))) \tag{11}$$

$$\mathsf{D}_{\mathcal{A}}(n, \mathsf{i}_{\mathcal{A}}(\mathsf{i}_{\mathcal{A}}(\mathsf{h}_{\mathcal{A}}) \mid_{\mathcal{A}} y)) = \omega^{\omega \oplus y} \;>_{\mathsf{O}}\; \omega^{(n+1)\oplus y} = \mathsf{i}_{\mathcal{A}}(\mathsf{C}_{\mathcal{A}}(n, \mathsf{h}_{\mathcal{A}}) \mid_{\mathcal{A}} y) \tag{12}$$

$$\mathsf{D}_{\mathcal{A}}(n, \mathsf{i}_{\mathcal{A}}(\mathsf{i}_{\mathcal{A}}(\mathsf{h}_{\mathcal{A}}))) = \omega^{\omega} \;>_{\mathsf{O}}\; \omega^{n+1} = \mathsf{i}_{\mathcal{A}}(\mathsf{C}_{\mathcal{A}}(n, \mathsf{h}_{\mathcal{A}})) \tag{13}$$

$$\mathsf{B}_{\mathcal{A}}(n, \mathsf{E}_{\mathcal{A}}(x)) = x + 1 \;>_{\mathsf{O}}\; x = \mathsf{A}_{\mathcal{A}}(\mathsf{s}_{\mathcal{A}}(n), x) \tag{14}$$

Note that inequalities (10)—(13) use the fact that $\omega >_{\mathsf{O}} n$ holds for $n \in \mathbb{N}$. The compatibility of $\mathcal{A}$ with AC follows from the associativity and the commutativity of $\oplus$:

$$(x \mid_{\mathcal{A}} y) \mid_{\mathcal{A}} z = (x \oplus y) \oplus z \;=_{\mathsf{O}}\; x \oplus (y \oplus z) = x \mid_{\mathcal{A}} (y \mid_{\mathcal{A}} z)$$

$$x \mid_{\mathcal{A}} y = x \oplus y \;=_{\mathsf{O}}\; y \oplus x = x \mid_{\mathcal{A}} y$$

Therefore, $\mathcal{A}$ is a quasi-model of $\mathcal{H}/\mathsf{AC}$. □

We now label A and B by the value of their second argument. Let $L_{\mathsf{A}} = L_{\mathsf{B}} = \mathbb{O}$ and $L_f = \varnothing$ for the other function symbols f, and define lab as follows:

$$\mathsf{lab}_{\mathsf{A}}(n, x) = \mathsf{lab}_{\mathsf{B}}(n, x) = x$$

The labeling (L, lab) results in the infinite rewrite system $\mathcal{H}_{\mathsf{lab}} \cup \mathcal{D}\mathsf{ec}$ with $\mathcal{H}_{\mathsf{lab}}$ consisting of the rewrite rules

$$\mathsf{A}_{\omega}(n, \mathsf{i}(\mathsf{h})) \xrightarrow{1} \mathsf{A}_1(\mathsf{s}(n), \mathsf{h}) \qquad \mathsf{D}(n, \mathsf{i}(\mathsf{i}(x))) \xrightarrow{8} \mathsf{i}(\mathsf{D}(n, \mathsf{i}(x)))$$

$$\mathsf{A}_{\omega^{v+1}}(n, \mathsf{i}(\mathsf{h} \mid x)) \xrightarrow{2} \mathsf{A}_{\omega^v}(\mathsf{s}(n), \mathsf{i}(x)) \qquad \mathsf{D}(n, \mathsf{i}(\mathsf{i}(x) \mid y)) \xrightarrow{9} \mathsf{i}(\mathsf{D}(n, \mathsf{i}(x)) \mid y)$$

$$\mathsf{A}_{\omega^v}(n, \mathsf{i}(x)) \xrightarrow{3} \mathsf{B}_{\omega^v}(n, \mathsf{D}(\mathsf{s}(n), \mathsf{i}(x))) \qquad \mathsf{D}(n, \mathsf{i}(\mathsf{i}(\mathsf{h} \mid x) \mid y)) \xrightarrow{10} \mathsf{i}(\mathsf{C}(n, \mathsf{i}(x)) \mid y)$$

$$\mathsf{C}(0, x) \xrightarrow{4} \mathsf{E}(x) \qquad \mathsf{D}(n, \mathsf{i}(\mathsf{i}(\mathsf{h} \mid x))) \xrightarrow{11} \mathsf{i}(\mathsf{C}(n, \mathsf{i}(x)))$$

$$\mathsf{C}(\mathsf{s}(n), x) \xrightarrow{5} x \mid \mathsf{C}(n, x) \qquad \mathsf{D}(n, \mathsf{i}(\mathsf{i}(\mathsf{h}) \mid y)) \xrightarrow{12} \mathsf{i}(\mathsf{C}(n, \mathsf{h}) \mid y)$$

$$\mathsf{i}(\mathsf{E}(x) \mid y) \xrightarrow{6} \mathsf{E}(\mathsf{i}(x \mid y)) \qquad \mathsf{D}(n, \mathsf{i}(\mathsf{i}(\mathsf{h}))) \xrightarrow{13} \mathsf{i}(\mathsf{C}(n, \mathsf{h}))$$

$$\mathsf{i}(\mathsf{E}(x)) \xrightarrow{7} \mathsf{E}(\mathsf{i}(x)) \qquad \mathsf{B}_{v+1}(n, \mathsf{E}(x)) \xrightarrow{14} \mathsf{A}_v(\mathsf{s}(n), x)$$

for all $v \in \mathbb{O}$ and $\mathcal{D}\mathsf{ec}$ consisting of the rewrite rules

$$\mathsf{A}_v(n, x) \to \mathsf{A}_w(n, x) \qquad \mathsf{B}_v(n, x) \to \mathsf{B}_w(n, x)$$

for all $v, w \in \mathbb{O}$ with $v > w$.

Example 4.3. The first rewrite sequence in Example 3.7 is simulated as follows:

$$\begin{aligned}
\mathsf{A}_u(0, \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h})))) &\xrightarrow{3} \mathsf{B}_u(0, \mathsf{D}(\mathsf{s}(0), \mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{h})))))\\
&\xrightarrow{8} \mathsf{B}_u(0, \mathsf{i}(\mathsf{D}(\mathsf{s}(0), \mathsf{i}(\mathsf{i}(\mathsf{h}))))))\\
&\xrightarrow{13} \mathsf{B}_u(0, \mathsf{i}(\mathsf{i}(\mathsf{C}(\mathsf{s}(0), \mathsf{h})))) \to_{\mathcal{D}\mathsf{ec}} \mathsf{B}_{v+1}(0, \mathsf{i}(\mathsf{i}(\mathsf{C}(\mathsf{s}(0), \mathsf{h})))))\\
&\xrightarrow{5} \mathsf{B}_{v+1}(0, \mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{C}(0, \mathsf{h})))))\\
&\xrightarrow{4} \mathsf{B}_{v+1}(0, \mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{E}(\mathsf{h})))) =_{\mathsf{AC}} \mathsf{B}_{v+1}(0, \mathsf{i}(\mathsf{i}(\mathsf{E}(\mathsf{h}) \mid \mathsf{h})))\\
&\xrightarrow{6} \mathsf{B}_{v+1}(0, \mathsf{i}(\mathsf{E}(\mathsf{i}(\mathsf{h} \mid \mathsf{h}))))\\
&\xrightarrow{7} \mathsf{B}_{v+1}(0, \mathsf{E}(\mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{h})))))\\
&\xrightarrow{14} \mathsf{A}_v(\mathsf{s}(0), \mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{h})))
\end{aligned}$$

Here $u = \omega^{\omega^\omega}$ and $v = \omega^{\omega^2}$.

According to Theorem 4.1, the AC termination of $\mathcal{H}$ on many-sorted terms follows from the AC termination of $\mathcal{H}_{\mathsf{lab}} \cup \mathcal{D}\mathsf{ec}$.

Corollary 4.4. *If $\mathcal{H}_{\mathsf{lab}} \cup \mathcal{D}\mathsf{ec}$ is* AC *terminating, $\mathcal{H}$ is* AC *terminating on sorted terms.* □

5. AC-MPO

In order to show AC termination of $\mathcal{H}_{\mathsf{lab}} \cup \mathcal{D}\mathsf{ec}$ we use a simplified version of AC-RPO.

Definition 5.1. Let $\mathcal{F}_{\mathsf{AC}}$ be the set of AC symbols in $\mathcal{F}$. Given a non-variable term $t = f(t_1, \dots, t_n)$, the multiset $\triangledown(t)$ is defined inductively as follows:

$$\triangledown(t) = \triangledown_f(t_1) \uplus \cdots \uplus \triangledown_f(t_n)$$

$$\triangledown_f(t) = \begin{cases} \triangledown_f(t_1) \uplus \triangledown_f(t_2) & \text{if } t = f(t_1, t_2) \text{ and } f \in \mathcal{F}_{\mathsf{AC}} \\ \{t\} & \text{otherwise} \end{cases}$$

For example, if $+$ is an AC symbol, we have $\triangledown_+(\mathsf{a}+(\mathsf{b}+x)) = \{\mathsf{a}, \mathsf{b}, x\}$. If f is a non-AC symbol, we have $\triangledown(f(t_1, \dots, t_n)) = \{t_1, \dots, t_n\}$.

The multiset extension $=_{\mathsf{AC}}^{\mathsf{mul}}$ of the equivalence relation $=_{\mathsf{AC}}$ is inductively defined as follows: $\varnothing =_{\mathsf{AC}}^{\mathsf{mul}} \varnothing$ and $\{s\} \uplus M =_{\mathsf{AC}}^{\mathsf{mul}} \{t\} \uplus N$ if $s =_{\mathsf{AC}} t$ and $M =_{\mathsf{AC}}^{\mathsf{mul}} N$. It is not difficult to see that $=_{\mathsf{AC}}^{\mathsf{mul}}$ is an equivalence relation. We have $\triangledown(s) =_{\mathsf{AC}}^{\mathsf{mul}} \triangledown(t)$ whenever $s =_{\mathsf{AC}} t$.

Definition 5.2. *Precedences* are strict orders on function symbols. Let $>$ be a precedence. We define $>_{\mathsf{acmpo}}$ inductively as follows: $s >_{\mathsf{acmpo}} t$ if $s \notin \mathcal{V}$ and one of the following conditions holds:

(1) $\triangledown(s) \geqslant_{\mathsf{acmpo}}^{\mathsf{mul}} \{t\}$,

(2) $\mathsf{root}(s) > \mathsf{root}(t)$ and $\{s\} >_{\mathsf{acmpo}}^{\mathsf{mul}} \triangledown(t)$,

(3) $\mathsf{root}(s) = \mathsf{root}(t)$ and $\triangledown(s) >_{\mathsf{acmpo}}^{\mathsf{mul}} \triangledown(t)$.

In the third condition $=_{\mathsf{AC}}$ is used instead of $=$ in the definition of multiset extension. We write $\geqslant_{\mathsf{acmpo}}$ for the union of $>_{\mathsf{acmpo}}$ and $=_{\mathsf{AC}}$.

The first condition is equivalent to $s' \geqslant_{\mathsf{acmpo}} t$ for some $s' \in \triangledown(s)$ and the second condition is equivalent to the conjunction of $\mathsf{root}(s) > \mathsf{root}(t)$ and $s >_{\mathsf{acmpo}} t'$ for all $t' \in \triangledown(t)$. These equivalences will be used freely in the sequel. The multiset comparison in the third condition is spelled out as follows: $\triangledown(s) >_{\mathsf{acmpo}}^{\mathsf{mul}} \triangledown(t)$ if there exist multisets S_1, S_2, T_1 and T_2 such that $\triangledown(s) = S_1 \uplus S_2$, $\triangledown(t) = T_1 \uplus T_2$, $S_1 =_{\mathsf{AC}}^{\mathsf{mul}} T_1$, $S_2 \neq \varnothing$, and for every $t' \in T_2$ there exists a term $s' \in S_2$ such that $s' >_{\mathsf{acmpo}} t'$.

Note that if there are no AC symbols, the above definition reduces to the original recursive path order of Dershowitz [Der82], nowadays known as the *multiset path order*. Moreover, if AC symbols are minimal in a precedence, AC-RPO with the multiset status reduces to AC-MPO. Hence the simplified AC-RPO will be called AC-MPO.

The proof of the following result can be found in the appendix. *Incrementality* of AC-MPO means that for precedences $>$ and $\sqsupset$ the inclusion $>_{\mathsf{acmpo}} \subseteq \sqsupset_{\mathsf{acmpo}}$ holds whenever $> \subseteq \sqsupset$.

Theorem 5.3. *If* AC *symbols are minimal in the precedence $>$ then $>_{\mathsf{acmpo}}$ is an incremental* AC*-compatible rewrite order with the subterm property.* □

As a consequence, $>_{\mathsf{acmpo}}$ is an AC-compatible reduction order when the underlying signature is finite. This also holds for infinite signatures, provided the precedence $>$ is well-founded and there are only finitely many AC symbols. This extension is important because the signature of $\mathcal{H}_{\mathsf{lab}}$ is infinite. Below, we will formally prove the correctness of the extension, by adopting the approach of [MZ97].

A strict order $>$ on a set A is a *partial well-order* if for every infinite sequence $a_0, a_1, \ldots$ of elements in A there exist indices i and j such that $i < j$ and $a_i \leqslant a_j$. Well-founded total orders (*well-orders*) are partial well-orders. Given a partial well-order $>$ on $\mathcal{F}$, the *embedding* TRS $\mathcal{E}\mathsf{mb}(\mathcal{F}, >)$ consists of the rules $f(x_1, \ldots, x_n) \to x_i$ for every n-ary function symbol and $1 \leqslant i \leqslant n$, together with the rules $f(x_1, \ldots, x_n) \to g(x_{i_1}, \ldots, x_{i_m})$ for all function symbols f and g with arities m and n such that $f > g$, and indices $1 \leqslant i_1 < i_2 < \cdots < i_m \leqslant n$. Here $x_1, \ldots, x_n$ are pairwise distinct variables.

Theorem 5.4 [MZ97, Theorem 5.3]**.** *A rewrite order $>$ is well-founded if $\mathcal{E}\mathsf{mb}(\mathcal{F}, \sqsupset) \subseteq >$ for some partial well-order $\sqsupset$.* □

Theorem 5.5. *Consider a signature $\mathcal{F}$ with only finitely many* AC *symbols that are minimal in a given well-founded precedence $>$. The relation $>_{\mathsf{acmpo}}$ is an* AC*-compatible reduction order.*

Proof. We only need to show well-foundedness of $>_{\mathsf{acmpo}}$ because the other properties follow by Theorem 5.3. Let $\sqsupset$ be an arbitrary partial well-order that contains $>$ and in which AC symbols are minimal. The inclusion $\mathcal{E}\mathsf{mb}(\mathcal{F}, \sqsupset) \subseteq \sqsupset_{\mathsf{acmpo}}$ is easily verified. Hence the well-foundedness of $\sqsupset_{\mathsf{acmpo}}$ is obtained from Theorem 5.4. Since $> \subseteq \sqsupset$, the incrementality of AC-MPO yields $>_{\mathsf{acmpo}} \subseteq \sqsupset_{\mathsf{acmpo}}$. It follows that $>_{\mathsf{acmpo}}$ is well-founded. □

We show the termination of $\mathcal{H}_{\mathsf{lab}} \cup \mathcal{D}\mathsf{ec}$ by AC-MPO. To this end, we consider the following precedence $>$ on the labeled signature:

$$\begin{aligned}
\mathsf{A}_v &> \mathsf{A}_w && \text{for all } v, w \in \mathbb{O} \text{ with } v > w\\
\mathsf{B}_v &> \mathsf{B}_w && \text{for all } v, w \in \mathbb{O} \text{ with } v > w\\
\mathsf{B}_{v+1} &> \mathsf{A}_v > \mathsf{B}_v && \text{for all } v \in \mathbb{O}\\
\mathsf{B}_0 &> \mathsf{s} > \mathsf{D} > \mathsf{C} > \mathsf{i} > \mathsf{E} > |
\end{aligned}$$

Note that $>$ is well-founded and the only AC symbol $|$ is minimal. In order to ease the compatibility verification we employ the following simple criterion.

Lemma 5.6. *Let $\ell \to r$ be a rewrite rule and let $>$ be a precedence. If $\mathsf{root}(\ell) > g$ for all function symbols g in r then $\ell >_{\mathsf{acmpo}} r$.* □

Theorem 5.7. $\mathcal{H}_{\mathsf{lab}} \cup \mathcal{D}\mathsf{ec} \subseteq >_{\mathsf{acmpo}}$

Proof. Lemma 5.6 applies to all rules of $\mathcal{H}_{\mathsf{lab}} \cup \mathcal{D}\mathsf{ec}$, except $5-9$. We consider rule 6 here; the other rewrite rules are handled in a similar fashion. Since case (1) of Definition 5.2 yields $\mathsf{E}(x) >_{\mathsf{acmpo}} x$, we have $\triangledown(\mathsf{E}(x) \mid y) = \{\mathsf{E}(x), y\} >^{\mathsf{mul}}_{\mathsf{acmpo}} \{x, y\} = \triangledown(x \mid y)$. Thus $\mathsf{E}(x)|y >_{\mathsf{acmpo}} x|y$ follows by case (3). Using case (3) again, we obtain $\mathsf{i}(\mathsf{E}(x)|y) >_{\mathsf{acmpo}} \mathsf{i}(x|y)$. Because of $\mathsf{i} > \mathsf{E}$, the desired orientation $\mathsf{i}(\mathsf{E}(x) \mid y) >_{\mathsf{acmpo}} \mathsf{i}(x \mid y)$ is concluded by case (2). □

Theorem 5.8. *The* TRS $\mathcal{H}_{\mathsf{lab}} \cup \mathcal{D}\mathsf{ec}$ *is* AC *terminating.* □

From Theorems 3.6 and 5.7 we conclude that Hercules eventually beats Hydra in any battle. Theorems 5.7 and 3.8 in connection with Corollary 4.4 yield the AC termination of $\mathcal{H}$ on arbitrary terms.

6. Related Work

In an influential survey paper, Dershowitz and Jouannaud [DJ90, p. 270] introduced a 5-rule rewrite system to simulate the Hydra Battle. The proposed rewrite system was later shown to be erroneous. A corrected version together with a detailed termination analysis has been given by Dershowitz and Moser [DM07], see also Moser [Mos09]. Earlier, Touzet [Tou98] presented an 11-rule rewrite system that encodes a specific battle with weakened Hydras (whose height is bounded by 4) and proved total termination by a semantic termination method. It is worth noting that our rewrite system $\mathcal{H}$ is not even simply terminating on unsorted terms. In fact, we have the following cyclic sequence with respect to $\mathcal{H} \cup \mathcal{E}\mathsf{mb}(\mathcal{F}, \varnothing)$:

$$\begin{aligned}\mathsf{A}(\mathsf{E}(\mathsf{i}(x)), \mathsf{i}(x)) &\xrightarrow{3} \mathsf{B}(\mathsf{E}(\mathsf{i}(x)), \mathsf{D}(\mathsf{s}(\mathsf{E}(\mathsf{i}(x))), \mathsf{i}(x))) \to^*_{\mathcal{E}\mathsf{mb}(\mathcal{F},\varnothing)} \mathsf{B}(\mathsf{E}(\mathsf{i}(x)), \mathsf{i}(x)) \\ &\xrightarrow{14} \mathsf{A}(\mathsf{s}(\mathsf{E}(\mathsf{i}(x))), \mathsf{i}(x)) \to_{\mathcal{E}\mathsf{mb}(\mathcal{F},\varnothing)} \mathsf{A}(\mathsf{E}(\mathsf{i}(x)), \mathsf{i}(x))\end{aligned}$$

So the TRS $\mathcal{H}$ is not simply terminating (see [MZ97, Lemma 4.6]). This is the reason that our termination proof employs semantic labeling.

The rewrite systems referred to above model the so-called *standard* battle, which corresponds to a specific strategy for Hercules. In this regard it is interesting to quote Kirby and Paris [KP82], who introduced the battle as an accessible example of an independence result for Peano arithmetic (P):

> A *strategy* is a function which determines for Hercules which head to chop off at each stage of any battle. It is not hard to find a reasonably fast *winning strategy* (i.e. a strategy which ensures that Hercules wins against any hydra). More surprisingly, Hercules cannot help winning:
>
> Theorem 2. (i) *Every strategy is a winning strategy.*
>
> [...]
>
> Theorem 2. (ii) *The statement "every recursive strategy is a winning strategy" is not provable from* P.

In a recent paper [EKO21, Section 6], rules are presented to slay Hydras, independent of the strategy. These rules do not constitute a term rewrite system in the usual sense (they operate on terms with *sequence variables*). More importantly, the infinitely many rules do not faithfully represent the battle. Earlier, Ferreira and Zantema [FZ96, Section 10] presented an infinite rewrite system to model the standard strategy and gave a direct ordinal interpretation to conclude its termination. In neither of the latter two papers stages of the battle are modeled.

7. Conclusion

We presented a new TRS encoding of the Battle of Hydra and Hercules. Unlike earlier encodings, it makes use of AC symbols. This allows us to faithfully model any strategy of Hercules, as envisaged in the paper by Kirby and Paris [KP82] in which the Battle was first presented. To prove the termination of the encoding we employed many-sorted rewriting

modulo AC and we extended semantic labeling modulo AC to many-sorted TRSs. The infinite TRS produced by semantic labeling was proved terminating by suitably instantiating AC-RPO.

One of the reviewers for this article pointed out that the Hydra battle can still be simulated even if rule 6 in $\mathcal{H}$ is replaced by the simpler $\mathsf{E}(x) \mid y \to \mathsf{E}(x \mid y)$. While we expect that this variant also has the termination property, the presented termination methods are not applicable. In fact, the rule cannot be ordered by AC-MPO and the AC symbol $\mid$ cannot be labeled. The reviewer also suggested an alternative encoding of Hydras that omits i from $\mathsf{i}(t_1 \mid \cdots \mid t_n)$. For instance, H_0 in Example 3.2 is written as $\mathsf{i}(\mathsf{h}) \mid \mathsf{i}(\mathsf{i}(\mathsf{h} \mid \mathsf{h})) \mid \mathsf{h}$ in this encoding. While this simplifies representations of Hydras, it seems difficult to construct an ordinal interpretation of $\mid$ for semantic labeling. Further investigations of AC termination techniques are required.

The finite TRS $\mathcal{H}$ poses an interesting challenge for automatic termination tools. None of the tools (AProVE [GAB+17], muterm [AGLNM11]) competing in the "TRS Equational" category of the Termination Competition 2024[1] succeeds on $\mathcal{H}/\mathsf{AC}$. This is not really surprising since most methods implemented in termination tool come with a multiple recursive upper bound on the derivation height (e.g. [Hof92, Lep01, MS11]). The tools even fail to prove termination of $\mathcal{H}$ without AC. The tool TTT2 [KSZM09] has support for ordinal interpretations [ZWM15] but also fails on $\mathcal{H}$.

Formalizing the techniques used in this article in a proof assistant is an important task to ensure the correctness of the results. Interestingly, the informal paper [HM22] in which we announced our encoding also presents a termination proof, essentially extending a semantic method of Touzet [Tou98] and Zantema [Zan01] to AC rewriting. Although we believe the non-trivial extension to be correct, its use in proving the AC termination of $\mathcal{H}$ has a critical mistake, which we recently discovered.

Another topic for future research is to investigate the scope of many-sorted semantic labeling. Can the termination of earlier encodings of the battle be established with many-sorted semantic labeling followed by some standard simplification order? Variants of the battle by Buchholz [Buc87] and Lepper [Lep04] are also of interest here.

Acknowledgements

We are grateful to the anonymous reviewers for their pertinent comments, which helped to improve the presentation. We thank Teppei Saito for his thorough feedback on the proofs for AC-MPO.

References

[AGLNM11] Beatriz Alarcón, Raúl Gutiérrez, Salvador Lucas, and Rafael Navarro-Marset. Proving termination properties with MU-TERM. In *Proc. 13th Algebraic Methodology and Software Technology*, volume 6486 of *Lecture Notes in Computer Science*, pages 201–208, 2011. doi:10.1007/978-3-642-17796-5_12.

[Buc87] Wilfried Buchholz. An independence result for $(\pi^1_1 - \mathrm{CA}) + \mathrm{BI}$. *Annals of Pure and Applied Logic*, 33:131–155, 1987. doi:10.1016/0168-0072(87)90078-9.

[Buc06] Wilfried Buchholz. Another rewrite system for the standard Hydra battle. In *Proc. Mini-Workshop: Logic, Combinatorics and Independence Results*, volume 3(4) of *Oberwolfach Reports*, pages 3099–3102. European Mathematical Society, 2006.

[1] https://termcomp.github.io/Y2024/

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982. doi:10.1016/0304-3975(82)90026-3.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. *Rewrite Systems. Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, 1990.

[DM07] Nachum Dershowitz and Georg Moser. The Hydra battle revisited. In *Rewriting, Computation and Proof, Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of his 60th Birthday*, volume 4600 of *Lecture Notes in Computer Science*, pages 1–27, 2007. doi:10.1007/978-3-540-73147-4_1.

[EKO21] Jörg Endrullis, Jan Willem Klop, and Roy Overbeek. Star games and Hydras. *Logical Methods in Computer Science*, 17(2):20:1–20:32, 2021. doi:10.23638/LMCS-17(2:20)2021.

[FZ96] Maria C. F. Ferreira and Hans Zantema. Total termination of term rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 7(2):133–162, 1996. doi:10.1007/BF01191381.

[GAB+17] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning*, 58(1):3–31, 2017. doi:10.1007/s10817-016-9388-y.

[HM22] Nao Hirokawa and Aart Middeldorp. Hydra battles and AC termination. In *Proc. 18th International Workshop on Termination*, pages 21–25, 2022.

[HM23] Nao Hirokawa and Aart Middeldorp. Hydra battles and AC termination. In *Proc. 8th Formal Structures for Computation and Deduction*, volume 260 of *Leibniz International Proceedings in Informatics*, pages 12:1–12:16, 2023. doi:10.4230/LIPIcs.FSCD.2023.12.

[Hof92] Dieter Hofbauer. Termination proofs by multiset path orderings imply primitive recursive derivation lengths. *Theoretical Computer Science*, 105(1):129–140, 1992. doi:10.1016/0304-3975(92)90289-R.

[KM76] Kazimierz Kuratowski and Andrzej Mostowski. *Set Theory — With an Introduction to Descriptive Set Theory (second edition)*, volume 86 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1976.

[KP82] Laurence Kirby and Jeff Paris. Accessible independence results for Peano arithmetic. *Bulletin of the London Mathematical Society*, 14:285–325, 1982. doi:10.1112/blms/14.4.285.

[KSZM09] Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean Termination Tool 2. In *Proc. 20th International Conference on Rewriting Techniques and Applications*, volume 5595 of *Lecture Notes in Computer Science*, pages 295–304, 2009. doi:10.1007/978-3-642-02348-4_21.

[Lep01] Ingo Lepper. Derivation lengths and order types of Knuth–Bendix orders. *Theoretical Computer Science*, 269(1-2):433–450, 2001. doi:10.1016/S0304-3975(01)00015-9.

[Lep04] Ingo Lepper. Simply terminating rewrite systems with long derivations. *Archive for Mathematical Logic*, 43(1):1–18, 2004. doi:10.1007/s00153-003-0190-2.

[MO00] Aart Middeldorp and Hitoshi Ohsaki. Type introduction for equational rewriting. *Acta Informatica*, 36(12):1007–1029, 2000. doi:10.1007/PL00013300.

[Mos09] Georg Moser. The Hydra battle and Cichon's principle. *Applicable Algebra in Engineering, Communication and Computing*, 20(2):133–158, 2009. doi:10.1007/s00200-009-0094-4.

[MS11] Georg Moser and Andreas Schnabl. The derivational complexity induced by the dependency pair method. *Logical Methods in Computer Science*, 7(3), 2011. doi:10.2168/LMCS-7(3:1)2011.

[MZ97] Aart Middeldorp and Hans Zantema. Simple termination of rewrite systems. *Theoretical Computer Science*, 175(1):127–158, 1997. doi:10.1016/S0304-3975(96)00172-7.

[OMG00] Hitoshi Ohsaki, Aart Middeldorp, and Jürgen Giesl. Equational termination by semantic labelling. In *Proc. 14th EACSL Annual Conference on Computer Science Logic*, volume 1862 of *Lecture Notes in Computer Science*, pages 457–471, 2000. doi:10.1007/3-540-44622-2_31.

[Rub02] Albert Rubio. A fully syntactic AC-RPO. *Information and Computation*, 178(2):515–533, 2002. doi:10.1006/inco.2002.3158.

[Ste90] Joachim Steinbach. AC-termination of rewrite systems: A modified Knuth–Bendix ordering. In *Proc. 2nd International Conference on Algebraic and Logic Programming*, volume 463 of *Lecture Notes in Computer Science*, pages 372–386, 1990. doi:10.1007/3-540-53162-9_52.

[Tou98] Hélène Touzet. Encoding the Hydra battle as a rewrite system. In *Proc. 23rd International Symposium on Mathematical Foundations of Computer Science*, volume 1450 of *Lecture Notes in Computer Science*, pages 267–276, 1998. doi:10.1007/BFb0055776.

[Zan94] Hans Zantema. Termination of term rewriting: Interpretation and type elimination. *Journal of Symbolic Computation*, 17(1):23–50, 1994. doi:10.1006/jsco.1994.1003.

[Zan95] Hans Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995. doi:10.3233/FI-1995-24124.

[Zan01] Hans Zantema. The termination hierarchy for term rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 12(1-2):3–19, 2001. doi:10.1007/s002000100061.

[ZWM15] Harald Zankl, Sarah Winkler, and Aart Middeldorp. Beyond polynomials and Peano arithmetic—Automation of elementary and ordinal interpretations. *Journal of Symbolic Computation*, 69:129–158, 2015. doi:10.1016/j.jsc.2014.09.033.

Appendix A. Proof of Theorem 5.3

We first show the AC-compatibility of AC-MPO.

Lemma A.1. *The relation* $>_{\mathsf{acmpo}}$ *is* AC*-compatible.*

Proof. First assume $s >_{\mathsf{acmpo}} t =_{\mathsf{AC}} u$. By induction on $|s| + |t|$ we show $s >_{\mathsf{acmpo}} u$. We distinguish three cases, according to Definition 5.2.

(1) If $s' \geqslant_{\mathsf{acmpo}} t$ for some $s' \in \triangledown(s)$ then also $s' \geqslant_{\mathsf{acmpo}} u$, either by the induction hypothesis or by the transitivity of $=_{\mathsf{AC}}$. Therefore $s >_{\mathsf{acmpo}} u$ by case (1).

(2) Suppose $\mathsf{root}(s) > \mathsf{root}(t)$ and $s >_{\mathsf{acmpo}} t'$ for all $t' \in \triangledown(t)$. From $t =_{\mathsf{AC}} u$ we derive $\mathsf{root}(t) = \mathsf{root}(u)$ and $\triangledown(t) =_{\mathsf{AC}}^{\mathsf{mul}} \triangledown(u)$. So for every $u' \in \triangledown(u)$ there exists a term $t' \in \triangledown(t)$ with $t' =_{\mathsf{AC}} u'$. Because $s >_{\mathsf{acmpo}} t' =_{\mathsf{AC}} u'$ and $|t| > |t'|$, the induction hypothesis yields $s >_{\mathsf{acmpo}} u'$. Hence $\{s\} >_{\mathsf{acmpo}}^{\mathsf{mul}} \triangledown(u)$ and thus $s >_{\mathsf{acmpo}} u$ by case (2).

(3) Suppose $\mathsf{root}(s) = \mathsf{root}(t)$ and $\triangledown(s) >_{\mathsf{acmpo}}^{\mathsf{mul}} \triangledown(t)$. From $t =_{\mathsf{AC}} u$ we derive $\mathsf{root}(t) = \mathsf{root}(u)$ and $\triangledown(t) =_{\mathsf{AC}}^{\mathsf{mul}} \triangledown(u)$. As $\triangledown(s) >_{\mathsf{acmpo}}^{\mathsf{mul}} \triangledown(t)$, there exist multisets S_1, S_2, T_1, and T_2 such that $\triangledown(s) = S_1 \uplus S_2$, $\triangledown(t) = T_1 \uplus T_2$, $S_1 =_{\mathsf{AC}}^{\mathsf{mul}} T_1$, $S_2 \neq \varnothing$, and for every $t' \in T_2$ there exists a term $s' \in S_2$ with $s' >_{\mathsf{acmpo}} t'$. As $\triangledown(t) =_{\mathsf{AC}}^{\mathsf{mul}} \triangledown(u)$, we may write $\triangledown(u) = U_1 \uplus U_2$ with $T_1 =_{\mathsf{AC}}^{\mathsf{mul}} U_1$ and $T_2 =_{\mathsf{AC}}^{\mathsf{mul}} U_2$. As $S_1 =_{\mathsf{AC}}^{\mathsf{mul}} T_1 =_{\mathsf{AC}}^{\mathsf{mul}} U_1$, we obtain $S_1 =_{\mathsf{AC}}^{\mathsf{mul}} U_1$ from the transitivity of $=_{\mathsf{AC}}$. For every $u' \in U_2$ there exists a term $t' \in T_2$ with $t' =_{\mathsf{AC}} u'$. Moreover, there exists a term $s' \in S_2$ with $s' >_{\mathsf{acmpo}} t'$. Since $s' >_{\mathsf{acmpo}} t' =_{\mathsf{AC}} u'$ and $|s| + |t| > |s'| + |t'|$, the induction hypothesis yields $s' >_{\mathsf{acmpo}} u'$. Consequently, $\triangledown(s) >_{\mathsf{acmpo}}^{\mathsf{mul}} \triangledown(u)$. Hence $s >_{\mathsf{acmpo}} u$ by case (3).

Next assume $s =_{\mathsf{AC}} t >_{\mathsf{acmpo}} u$. By induction on $|t| + |u|$ we show $s >_{\mathsf{acmpo}} u$. From $s =_{\mathsf{AC}} t$ we infer $\mathsf{root}(s) = \mathsf{root}(t)$ and $\triangledown(s) =_{\mathsf{AC}}^{\mathsf{mul}} \triangledown(t)$. We distinguish three cases for $t >_{\mathsf{acmpo}} u$.

(1) Suppose $\triangledown(t) \geqslant_{\mathsf{acmpo}} \{u\}$. Since $\triangledown(s) =_{\mathsf{AC}}^{\mathsf{mul}} \triangledown(t)$, we obtain $\triangledown(s) >_{\mathsf{acmpo}}^{\mathsf{mul}} \{u\}$ by the induction hypothesis or the transitivity of $=_{\mathsf{AC}}$. Hence $s >_{\mathsf{acmpo}} u$ by case (1).

(2) Suppose $\mathsf{root}(t) > \mathsf{root}(u)$ and $t >_{\mathsf{acmpo}} u'$ for all $u' \in \triangledown(u)$. The induction hypothesis yields $s >_{\mathsf{acmpo}} u'$ for all $u' \in \triangledown(u)$. Since also $\mathsf{root}(s) > \mathsf{root}(u)$, $s >_{\mathsf{acmpo}} u$ by case (2).

(3) Suppose $\mathsf{root}(t) = \mathsf{root}(u)$ and $\triangledown(t) >_{\mathsf{acmpo}}^{\mathsf{mul}} \triangledown(u)$. Since $\triangledown(s) =_{\mathsf{AC}}^{\mathsf{mul}} \triangledown(t)$, we obtain $\triangledown(s) >_{\mathsf{acmpo}}^{\mathsf{mul}} \triangledown(u)$ by the induction hypothesis and the transitivity of $=_{\mathsf{AC}}$. Hence $s >_{\mathsf{acmpo}} u$ by case (3). □

Next we show transitivity.

Lemma A.2. *The relation $>_{\mathsf{acmpo}}$ is transitive.*

Proof. Suppose $s >_{\mathsf{acmpo}} t >_{\mathsf{acmpo}} u$. We show $s >_{\mathsf{acmpo}} u$ by induction on $|s| + |t| + |u|$. We do a case analysis on $s >_{\mathsf{acmpo}} t$.

(1) If $s' \geqslant_{\mathsf{acmpo}} t$ for some $s' \in \triangledown(s)$ then $s' >_{\mathsf{acmpo}} u$ by the induction hypothesis or the AC-compatibility of $>_{\mathsf{acmpo}}$ (Lemma A.1).
(2) Suppose $\mathsf{root}(s) > \mathsf{root}(t)$ and $\{s\} >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(t)$. We perform a second case analysis on $t >_{\mathsf{acmpo}} u$.
   - If $\triangledown(t) \geqslant_{\mathsf{acmpo}} \{u\}$ then we obtain $s >_{\mathsf{acmpo}} u$ by the induction hypothesis or the AC-compatibility of $>_{\mathsf{acmpo}}$.
   - If $\mathsf{root}(t) > \mathsf{root}(u)$ and $\{t\} >_{\mathsf{acmpo}} \triangledown(u)$ then $s >_{\mathsf{acmpo}} t >_{\mathsf{acmpo}} v$ for all $v \in \triangledown(u)$ and thus $\{s\} >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(u)$ by the induction hypothesis. Hence $s >_{\mathsf{acmpo}} u$ by case (2).
   - Suppose $\mathsf{root}(t) = \mathsf{root}(u)$ and $\triangledown(t) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(u)$. We obtain $\triangledown(s) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(u)$ from the induction hypothesis and the AC-compatibility of $>_{\mathsf{acmpo}}$. Thus, $s >_{\mathsf{acmpo}} u$ follows by case (3).
(3) Suppose $\mathsf{root}(s) = \mathsf{root}(t)$ and $\triangledown(s) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(t)$. Also in this case we perform an additional case analysis on $t >_{\mathsf{acmpo}} u$.
   - If $\triangledown(t) \geqslant_{\mathsf{acmpo}} \{u\}$ then we obtain $\triangledown(s) >_{\mathsf{acmpo}} \{u\}$ by the induction hypothesis or the AC-compatibility of $>_{\mathsf{acmpo}}$.
   - Suppose $\mathsf{root}(t) > \mathsf{root}(u)$ and $\{t\} >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(u)$. We have $\mathsf{root}(s) > \mathsf{root}(u)$. For every $v \in \triangledown(u)$ we have $s >_{\mathsf{acmpo}} t >_{\mathsf{acmpo}} v$, and thus $s >_{\mathsf{acmpo}} v$ by the induction hypothesis. Hence $\{s\} >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(u)$ and thus $s >_{\mathsf{acmpo}} u$ by case (2).
   - Suppose $\mathsf{root}(t) = \mathsf{root}(u)$ and $\triangledown(t) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(u)$. From $\triangledown(s) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(t) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(u)$ we infer $\triangledown(s) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(u)$ by the induction hypothesis, the AC-compatibility of $>_{\mathsf{acmpo}}$, and the transitivity of $=_{\mathsf{AC}}$. Hence $s >_{\mathsf{acmpo}} u$ by case (3). □

The subterm property is next.

Lemma A.3. *The relation $>_{\mathsf{acmpo}}$ has the subterm property.*

Proof. Let $t = f(t_1, \ldots, t_n)$. Fix $i \in \{1, \ldots, n\}$. We show $t >_{\mathsf{acmpo}} t_i$. The subterm property is then obtained by induction and the transitivity of $>_{\mathsf{acmpo}}$ (Lemma A.2). We distinguish two cases.

(1) If $t_i \in \triangledown(t)$ then $\triangledown(t) \geqslant_{\mathsf{acmpo}} \{t_i\}$ and thus $t >_{\mathsf{acmpo}} t_i$ by case (1).
(2) If $t_i \notin \triangledown(t)$ then f is an AC symbol, $n = 2$ and $\mathsf{root}(t_i) = f$. Since $\triangledown(t_i) \subsetneq \triangledown(t)$, $\triangledown(t) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(t_i)$ holds and thus $t >_{\mathsf{acmpo}} t_i$ by case (3). □

The preceding lemmata are used to prove irreflexivity.

Lemma A.4. *The relation $>_{\mathsf{acmpo}}$ is irreflexive.*

Proof. Assume to the contrary $t >_{\mathsf{acmpo}} t$. We derive a contradiction by induction on t. We distinguish three cases.

(1) Suppose $t' \geqslant_{\mathsf{acmpo}} t$ for some $t' \in \triangledown(t)$. Since $t \rhd t'$, the subterm property (Lemma A.3) yields $t >_{\mathsf{acmpo}} t$. So $t' \geqslant_{\mathsf{acmpo}} t >_{\mathsf{acmpo}} t'$, and thus $t' >_{\mathsf{acmpo}} t'$ is obtained by the transitivity (Lemma A.2) or AC compatibility (Lemma A.1) of $>_{\mathsf{acmpo}}$. The induction hypothesis yields the desired contraction.
(2) If $t >_{\mathsf{acmpo}} t$ is derived by case (2) then $\mathsf{root}(t) > \mathsf{root}(t)$, which contradicts the irreflexivity of the precedence $>$.

(3) Suppose $\triangledown(t) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(t)$. Let U be the set of all proper subterms of t, and $\succ$ the restriction of $>_{\mathsf{acmpo}}$ to $U \times U$. The multiset extension $\succ^{\mathsf{mul}}$ coincides with the restriction of $>^{\mathsf{mul}}_{\mathsf{acmpo}}$ to finite multisets over U. Hence $\triangledown(t) \succ^{\mathsf{mul}} \triangledown(t)$ follows from $\triangledown(t) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(t)$. The relation $\succ$ is irreflexive according to the induction hypothesis. Moreover, $\succ$ inherits transitivity from $>_{\mathsf{acmpo}}$ (Lemma A.2). Hence $\succ$ is a strict order and thus so is its multiset extension $\succ^{\mathsf{mul}}$. Since $\triangledown(t)$ is a finite multiset over U, $\triangledown(t) \succ^{\mathsf{mul}} \triangledown(t)$ cannot hold, yielding the desired contradiction. □

In the proof of closure under substitutions we use the fact that for an f-rooted term t and a substitution σ the multiset $\triangledown(t\sigma)$ is the multiset sum of $\triangledown_f(t'\sigma)$ for all $t' \in \triangledown(t)$.

Lemma A.5. *The relation $>_{\mathsf{acmpo}}$ is closed under substitutions.*

Proof. Suppose $s >_{\mathsf{acmpo}} t$ and let σ be a substitution. We show $s\sigma >_{\mathsf{acmpo}} t\sigma$ by induction on $|s| + |t|$. We distinguish three cases.

(1) Suppose $s' \geqslant_{\mathsf{acmpo}} t$ for some $s' \in \triangledown(s)$. If $s' >_{\mathsf{acmpo}} t$ then we obtain $s'\sigma >_{\mathsf{acmpo}} t\sigma$ from the induction hypothesis. If $s' =_{\mathsf{AC}} t$ then we obtain $s'\sigma =_{\mathsf{AC}} t\sigma$ from the closure under substitutions of $=_{\mathsf{AC}}$. So in both cases we have $s'\sigma \geqslant_{\mathsf{acmpo}} t\sigma$. If $s'\sigma \in \triangledown(s\sigma)$ then $\triangledown(s\sigma) >^{\mathsf{mul}}_{\mathsf{acmpo}} \{t\sigma\}$ and thus $s\sigma >_{\mathsf{acmpo}} t\sigma$ by case (1). If $s'\sigma \notin \triangledown(s\sigma)$ then $s' \in \mathcal{V}$ and thus $s' = t$ follows from $s' \geqslant_{\mathsf{acmpo}} t$. Hence $s'\sigma = t\sigma$. Since $s'\sigma \lhd s\sigma$ we obtain $s\sigma >_{\mathsf{acmpo}} t\sigma$ from the subterm property (Lemma A.3).

(2) Suppose $\mathsf{root}(s) > \mathsf{root}(t)$ and $\{s\} >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(t)$. Clearly $\mathsf{root}(s\sigma) = \mathsf{root}(s) > \mathsf{root}(t) = \mathsf{root}(t\sigma)$. Consider an arbitrary term $u \in \triangledown(t\sigma)$. Let $f = \mathsf{root}(t)$. There exists a term $t' \in \triangledown(t)$ such that $u \in \triangledown_f(t'\sigma)$. The induction hypothesis yields $s\sigma >_{\mathsf{acmpo}} t'\sigma$. Since $u \in \triangledown_f(t'\sigma)$ satisfies $u \trianglelefteq t'\sigma$, we have $t'\sigma \geqslant_{\mathsf{acmpo}} u$ by the subterm property (Lemma A.3) and thus $s\sigma >_{\mathsf{acmpo}} u$ by transitivity (Lemma A.2). Hence $\{s\sigma\} >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(t\sigma)$ and thus $s\sigma >_{\mathsf{acmpo}} t\sigma$ by case (2).

(3) Suppose $\mathsf{root}(s) = \mathsf{root}(t)$ and $\triangledown(s) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(t)$. We write f for $\mathsf{root}(s)$. Let $s' \in \triangledown(s)$ and $t' \in \triangledown(t)$ such that $s' >_{\mathsf{acmpo}} t'$ or $s' =_{\mathsf{AC}} t'$ is used in $\triangledown(s) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(t)$.

- We first consider $s' >_{\mathsf{acmpo}} t'$. Since $s' \lhd s$ and $t' \lhd t$, the induction hypothesis yields $s'\sigma >_{\mathsf{acmpo}} t'\sigma$. From $s' >_{\mathsf{acmpo}} t'$ we infer $s' \notin \mathcal{V}$ and thus $s'\sigma \notin \mathcal{V}$. Because of $s' \in \triangledown(s)$, we have $\mathsf{root}(s') \neq f$ and thus $\mathsf{root}(s'\sigma) \neq f$. Hence $\triangledown_f(s'\sigma) = \{s'\sigma\}$. Consider a term $u \in \triangledown_f(t'\sigma)$. Since $u \trianglelefteq t'\sigma$, we obtain $t'\sigma \geqslant_{\mathsf{acmpo}} u$ by the subterm property (Lemma A.3) and thus $s'\sigma >_{\mathsf{acmpo}} u$ by transitivity (Lemma A.2). Hence $\triangledown_f(s'\sigma) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown_f(t'\sigma)$ follows.
- Suppose $s' =_{\mathsf{AC}} t'$. Since $=_{\mathsf{AC}}$ is closed under substitutions, we have $s'\sigma =_{\mathsf{AC}} t'\sigma$ and thus $\triangledown_f(s'\sigma) =^{\mathsf{mul}}_{\mathsf{AC}} \triangledown_f(t'\sigma)$.

It follows that the derivation of $\triangledown(s) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(t)$ can be simulated, resulting in $\triangledown(s\sigma) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(t\sigma)$. Hence $s\sigma >_{\mathsf{acmpo}} t\sigma$ by case (3). □

The following technical result is used in the proof that AC-MPO is closed under contexts (if AC symbols are minimal in the precedence).

Lemma A.6. *If $f \in \mathcal{F}_{\mathsf{AC}}$ is minimal in $>$ and $s = f(s_1, s_2) >_{\mathsf{acmpo}} t$ then $\triangledown_f(s) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown_f(t)$.*

Proof. We have $\triangledown(s) = \triangledown_f(s) = \triangledown_f(s_1) \uplus \triangledown_f(s_2)$. We distinguish three cases.

(1) Suppose $s' \geqslant_{\mathsf{acmpo}} t$ for some $s' \in \triangledown(s)$. We obtain $\mathsf{root}(s') \neq f$ from $s' \in \triangledown(s) = \triangledown_f(s)$ and thus $\triangledown_f(s') = \{s'\}$. If also $\mathsf{root}(t) \neq f$, then $\triangledown_f(t) = \{t\}$ and thus $s' \geqslant_{\mathsf{acmpo}} t$ leads to $\triangledown_f(s) \supsetneq \triangledown_f(s') \geqslant^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown_f(t)$ and hence $\triangledown_f(s) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown_f(t)$. Suppose $\mathsf{root}(t) = f$. Let $v \in \triangledown_f(t)$. We have $t \rhd v$ and thus $t >_{\mathsf{acmpo}} v$. As $s' \geqslant_{\mathsf{acmpo}} t >_{\mathsf{acmpo}} v$, we obtain $s' >_{\mathsf{acmpo}} v$ by transitivity or AC-compatibility. Hence $\triangledown_f(s') >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown_f(t)$ and therefore $\triangledown_f(s) \supsetneq \triangledown_f(s') >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown_f(t)$ and $\triangledown_f(s) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown_f(t)$.
(2) Since f is minimal in the precedence, $s >_{\mathsf{acmpo}} t$ cannot be obtained by case (2).
(3) Suppose $\mathsf{root}(s) = \mathsf{root}(t) = f$ and $\triangledown(s) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(t)$. Since $\triangledown_f(s) = \triangledown(s)$ and $\triangledown_f(t) = \triangledown(t)$, the claim holds. □

Lemma A.7. *The relation $>_{\mathsf{acmpo}}$ is closed under contexts if* AC *symbols are minimal in the precedence $>$.*

Proof. Suppose $s >_{\mathsf{acmpo}} t$ and consider a context of the form $C = f(\dots, \Box, \dots)$. If $f \notin \mathcal{F}_{\mathsf{AC}}$ then $\triangledown(C[s]) - \triangledown(C[t]) = \{s\}$ and $\triangledown(C[t]) - \triangledown(C[s]) = \{t\}$ by the irreflexivity of $>_{\mathsf{acmpo}}$ (Lemma A.4), and thus $\triangledown(C[s]) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(C[t])$ follows from $s >_{\mathsf{acmpo}} t$. Hence $C[s] >_{\mathsf{acmpo}} C[t]$ by case (3). Suppose $f \in \mathcal{F}_{\mathsf{AC}}$. We have $C = f(\Box, u)$ or $C = f(u, \Box)$ and distinguish two cases.

- If $f = \mathsf{root}(s)$ then $\triangledown_f(s) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown_f(t)$ by Lemma A.6. So the inequality
$$\triangledown(C[s]) - \triangledown(C[t]) = \triangledown_f(s) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown_f(t) = \triangledown(C[t]) - \triangledown(C[s])$$
holds. Therefore, we obtain $C[s] >_{\mathsf{acmpo}} C[t]$ by case (3).
- If $f \neq \mathsf{root}(s)$ then $\triangledown(C[s]) = \{s\} \uplus \triangledown_f(u)$ and $\triangledown(C[t]) = \triangledown_f(t) \uplus \triangledown_f(u)$. According to case (3), it is enough to show $s >_{\mathsf{acmpo}} t'$ for all $t' \in \triangledown_f(t)$. Let $t' \in \triangledown_f(t)$. We have $t \trianglerighteq t'$ and thus $t \geqslant_{\mathsf{acmpo}} t'$ by the subterm property (Lemma A.3). As $s >_{\mathsf{acmpo}} t \geqslant_{\mathsf{acmpo}} t'$, we obtain $s >_{\mathsf{acmpo}} t'$ by transitivity or AC-compatibility. □

Incrementality is the final property in Theorem 5.3.

Lemma A.8. *The relation $>_{\mathsf{acmpo}}$ is incremental.*

Proof. Let $>$ and $\sqsupset$ be precedences with $> \subseteq \sqsupset$. Suppose $s >_{\mathsf{acmpo}} t$. We show $s \sqsupset_{\mathsf{acmpo}} t$ by induction on $|s| + |t|$. We distinguish three cases.

(1) If $\triangledown(s) \geqslant^{\mathsf{mul}}_{\mathsf{acmpo}} \{t\}$ then $s' =_{\mathsf{AC}} t$ or $s' >_{\mathsf{acmpo}} t$ for some $s' \in \triangledown(s)$. In the latter case, the induction hypothesis yields $s' \sqsupset_{\mathsf{acmpo}} t$. Hence in both cases $s \sqsupset_{\mathsf{acmpo}} t$ by case (1).
(2) If $\mathsf{root}(s) > \mathsf{root}(t)$ and $\{s\} >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(t)$ then $\{s\} \sqsupset_{\mathsf{acmpo}} \triangledown(t)$ by the induction hypothesis. Hence, $s \sqsupset_{\mathsf{acmpo}} t$ is obtained by case (2).
(3) Suppose $\mathsf{root}(s) = \mathsf{root}(t)$ and $\triangledown(s) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(t)$. Let $s' \in \triangledown(s)$ and $t' \in \triangledown(s)$ be a term pair such that $s' >_{\mathsf{acmpo}} t'$ is used in $\triangledown(s) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(t)$. Since $s' \lhd s$ and $t' \lhd t$, the induction hypothesis yields $s' \sqsupset_{\mathsf{acmpo}} t'$. Thus, the derivation of $\triangledown(s) >^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(t)$ can be simulated by $\sqsupset_{\mathsf{acmpo}}$. Therefore, $\triangledown(s) \sqsupset^{\mathsf{mul}}_{\mathsf{acmpo}} \triangledown(t)$, and hence $s \sqsupset_{\mathsf{acmpo}} t$ is obtained by case (3). □

Old and New Benchmarks for Relative Termination of String Rewrite Systems

Dieter Hofbauer
ASW Saarland

Johannes Waldmann
HTWK Leipzig

Abstract

We provide a critical assessment of the current set of benchmarks for relative SRS termination in the Termination Problems Database (TPDB): most of the benchmarks in `Waldmann_19` and `ICFP_10_relative` are, in fact, strictly terminating (i. e., terminating when non-strict rules are considered strict), so these benchmarks should be removed, or relabelled. To fill this gap, we enumerate small relative string rewrite systems. At present, we have complete enumerations for a 2-letter alphabet up to size 11, and for a 3-letter alphabet up to size 8. For some selected benchmarks, old and new, we discuss how to prove termination, automated or not.

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

Keywords and phrases termination, relative termination, string rewriting

1 Introduction

A rewrite system R is *terminating relative to* a system S if any (infinite) rewrite sequence modulo $R \cup S$ contains only finitely many R-steps. This property is denoted by $\mathsf{SN}(R/S)$. The term was coined by Klop [7]. Relative termination as a generalization of standard termination is a basis for modular termination proofs due to the fact that $\mathsf{SN}(R \cup S)$ holds if and only if both $\mathsf{SN}(R/S)$ and $\mathsf{SN}(S)$ hold. It is a research topic on its own, cf. [1, 11], and is widely used by current termination provers.

The Termination Problems Database (TPDB, see `https://termination-portal.org/wiki/TPDB`) contains relative termination benchmarks since version 2.0. For string rewriting, the directory `SRS_Relative` of the current TPDB (versions 11.*) consists of five subdirectories:

- `ICFP_2010_relative`: 160 benchmarks, cf. [10] [VBS TC22: 132].
- `Mixed_relative_SRS`: 20 benchmarks [VBS TC22: 20].
- `Waldmann_06_relative`: 11 benchmarks [VBS TC22: 11].
- `Waldmann_19`: 100 benchmarks, length-preserving (alphabet size 3, left- and right-hand sides of length 3, 2 to 7 rules), randomly chosen [VBS TC22: 91].
- `Zantema_06_relative`: 14 benchmarks, hand-crafted [VBS TC22: 9].

The number of benchmarks solved by the *Virtual Best Solver (VBS)* in the Termination Competition 2022 (TC22) are given in square brackets, see `https://termcomp.github.io/Y2022/SRS_Relative.html`. "The VBS records the best (consistent) score for each claim collected at least since 2018", according to `https://termination-portal.org/wiki/Termination_Competition_2022`.

The category `SRS_Relative` is missing an enumeration of small systems. We are starting this, and report some results and observations. The first enumeration of (strict) SRS was given by Kurth [9], and was later extended by Geser [2] and others [4]. Our ultimate goal is to cover all small relative SRSs (up to a certain size) for which termination cannot be decided

by current provers. The naive approach is to enumerate them all, and call the provers for each one. The challenge is to achieve the same result with fewer prover calls, by short-cutting the enumeration and omitting redundant cases.

There is no new theorem or method in this paper. It's just that no one has done the work before. Source code and data for the experiments reported here are available at `https://gitlab.imn.htwk-leipzig.de/waldmann/pure-matchbox/-/issues/472`.

2 How to win `SRS_Relative`, without proving Relative Termination

The approach is strikingly simple: prove (standard) termination instead for the *strictified* system, where all non-strict rules are replaced by their strict counterparts (replacing $\ell \to^= r$ by $\ell \to r$). This is based on the trivial observation that $\mathsf{SN}(R \cup S)$ implies $\mathsf{SN}(R/S)$. Surprisingly, this works for at least 210 (out of 305) benchmarks in `SRS_Relative`, in comparison to the winner of TC22, MultumNonMulta (MnM), with 203 `YES` and 8 `NO` answers.

What if the answer to the question about termination of the strictified system $R \cup S$ is `NO`? In this case we can infer $\neg\mathsf{SN}(R/S)$, provided we have a proof of $\mathsf{SN}(S)$. This is in fact the method we recommend for proving $\mathsf{SN}(R/S)$: if $\mathsf{SN}(S)$, then prove $\mathsf{SN}(R \cup S)$ using any of the methods for standard termination that are not available for relative termination (a DP transformation followed by weakly monotone matrix interpretations, or RFC-matchbounds). Only if $\neg\mathsf{SN}(S)$, check $\mathsf{SN}(R/S)$ with special methods (e. g., strictly monotone matrix interpretations).

The point is that methods for standard termination are more powerful (due to reduced monotonicity requirements, for example), and that checking $\mathsf{SN}(S)$ is straightforward (at least for the set of current benchmarks). A potential drawback is that we cannot re-use information from the proof, or disproof, of $\mathsf{SN}(S)$, to investigate $\mathsf{SN}(R \cup S)$ or $\mathsf{SN}(R/S)$.

3 Relative Non-Termination via Loops

One way of proving the statement $\neg\mathsf{SN}(R/S)$ is to exhibit a mixed looping derivation $v \to^+_{R \cup S} uvw$ that contains at least one strict R-step. In fact, all known non-terminating problems in `SRS_Relative` have been shown to be looping.

As shown by Geser and Zantema [5], loops for standard SRS can be characterized by looping forward closures. However, the corresponding statement does not hold for relative SRS: Consider the example $\{ab \to a, c \to^= bc\}$ from [3], which has a loop $(abc \to ac \to^= abc)$, but no looping forward closure. In this example, a looping forward closure can be found after reversing the system, but for the system $\{bab \to a, c \to^= cb, d \to^= bd\}$ neither the system nor its reversal has a looping forward closure, although a loop exists $(cad \to^{=2} cbabd \to cad)$.

We could use overlap closures instead, but this changes the nature of the enumeration since there is one case where we need to overlap two closures with a rule. If we want to limit ourselves to overlaps between one closure and one rule, then we can consider backward overlaps. A characterizing theorem for looping relative SRS is missing, to the best of our knowledge.

A special method for proving $\neg\mathsf{SN}(R/S)$ is this: exhibit a derivation $v \to^+_S uvw$ such that u or w contains an R-redex. As an example consider $R = \{a \to b\}$ and $S = \{c \to^= ac\}$, where $c \to^+_S ac$ and a is an R-redex. This corresponds to the so-called *R-emitting loops* for term rewriting, used by AProVE twice in TC22.

4 Experiments and Results

We did an optimized enumeration and called the prover matchbox on each relative SRS that was produced, using a simple strategy that contains weights (from GLPK), matrix interpretations of dimension 2 to 9 (with binary bitblasting for bit widths from 4 down to 1), full tiling for width 2, 3, 4 (possibly repeated), and closure enumeration, using parallel computation on up to 8 cores. Some unsolved problems will be submitted to TPDB.

4.1 Results for 2-letter alphabets

All SRS up to size 9 could be solved. Here, the size of a system is the sum of the sizes of its rules, and the size of a rule is the sum of the lengths of its left- and right hand sides. The empty string is denoted by ϵ. Unsolved for size 10 are

- $\{aa \to \epsilon, abb \to^= abbba\}$
- $\{aabba \to baab, \epsilon \to^= a\}$

4.2 Results for 3-letter alphabets

All SRS up to size 6 could be solved. Unsolved for size 7:

- $\{ac \to c, \epsilon \to^= ab, ab \to^= \epsilon\}$
- $\{ac \to c, \epsilon \to^= ab, ba \to^= \epsilon\}$

Unsolved for size 8: 24 systems, some of them looking vaguely similar, e. g.,

- $\{ac \to c, \epsilon \to^= ab, aab \to^= \epsilon\}$,

but also some that don't, like

- $\{ab \to \epsilon, \epsilon \to^= acb, acb \to^= \epsilon\}$.

5 Two Examples from the Enumeration

In this section, we give hand-crafted termination proofs for two samples from the enumeration.

5.1 Example $\{ac \to c, \epsilon \to^= ab, ab \to^= \epsilon\}$

The following interpretation M is a model for this system:

$$\begin{aligned} \epsilon_M &= 0 \\ a_M(y) &= \max(0, y-1) \\ b_M(y) &= y+1 \\ c_M(y) &= 0 \end{aligned}$$

Here, a string $w \in \{a,b\}^*$ gets value 0 iff $w \to^* a^n$ via the relative rules alone; the number n in that normal form gives the number of rewrite steps starting from wc. We compute n as the first component of the monotone interpretation I on domain $\mathbb{N}^2$ (the second component being M), ordered by $(x_1, y_1) > (x_2, y_2)$ iff $x_1 > x_2 \wedge y_1 = y_2$:

$$\begin{aligned} \epsilon_I &= (0,0) \\ a_I(x,y) &= \text{if } y > 0 \text{ then } (x, y-1) \text{ else } (x+1, 0) \\ b_I(x,y) &= (x, y+1) \\ c_I(x,y) &= (x, 0) \end{aligned}$$

Proof of Monotonicity, for $(x_1, y) > (x_2, y)$:

$$\begin{aligned} a_I(x_1, y) &= \text{if } y > 0 \text{ then } (x_1, y-1) \text{ else } (x_1+1, 0) \\ &> \text{if } y > 0 \text{ then } (x_2, y-1) \text{ else } (x_2+1, 0) \\ &= a_I(x_2, y) \end{aligned}$$

Proof of Compatibility, again for $(x_1, y) > (x_2, y)$:

$$\begin{aligned} ac_I(x, y) &= a_I(x, 0) = (x+1, 0) > (x, 0) = c_I(x, y) \\ ab_I(x, y) &= a_I(x, y+1) = (x, y) \end{aligned}$$

5.2 Example $\{ac \to c, \epsilon \to^= ab, ba \to^= \epsilon\}$

We want to use the same interpretation as in previous example, but its second component is no longer a model, since $ba_I(x, 0) = b_I(x+1, 0) = (x+1, 1) \neq (x, 0) = \epsilon_I(x, 0)$. The proof succeeds by keeping the interpretation and using different orders:

$$\begin{aligned} (x_1, y_1) > (x_2, y_2) \quad &\text{iff} \quad (x_1 > x_2) \wedge (y_1 \geq y_2) \wedge (x_1 - y_1 > x_2 - y_2) \\ (x_1, y_1) \geq (x_2, y_2) \quad &\text{iff} \quad (x_1 \geq x_2) \wedge (y_1 \geq y_2) \wedge (x_1 - y_1 \geq x_2 - y_2) \end{aligned}$$

6 Some SRS from `Zantema_06_relative`

We assume that these 13 systems were hand-crafted to be "obviously terminating", but outside of the range of (then) current methods of automation. Four of these benchmarks remain unsolved in the termination competition so far: `rel03`, `rel07`, `rel11`, and `rel12`. Meanwhile, we have automated proofs for `rel11` and `rel12`, obtained by brute-forced matrix interpretations [6, 8]. The following remain open:

- $\{ac \to cca, c \to^= baab, baab \to^= c\}$ (`rel03`)
- $\{ad \to db, a \to bbb, d \to \epsilon, a \to \epsilon, bc \to cdd, ac \to bbcd, bdb \to^= ad, ad \to^= bdb\}$ (`rel07`).

6.1 `rel11`: $\{bpb \to bapb, p \to^= apa, apaa \to^= p\}$

The following arctic-below-zero matrix interpretation of dimension 4 proves termination. Here, $-$ stands for $-\infty$:

$$a \mapsto \begin{pmatrix} 0 & - & - & - \\ - & -1 & - & - \\ - & - & 1 & - \\ - & - & - & 0 \end{pmatrix} \qquad b \mapsto \begin{pmatrix} 0 & 0 & - & - \\ 1 & 1 & - & 1 \\ 0 & 1 & - & 1 \\ - & 0 & - & 0 \end{pmatrix} \qquad p \mapsto \begin{pmatrix} 0 & - & - & 0 \\ - & - & 1 & - \\ - & - & - & - \\ - & - & - & 0 \end{pmatrix}$$

6.2 `rel12`: $\{bpb \to abapba, p \to^= apa, apa \to^= p\}$

A termination proof can be obtained by a natural matrix interpretation of dimension 5:

$$a \mapsto \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad b \mapsto \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 4 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad p \mapsto \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

7 Conclusion

With this contribution, we provide some small hard benchmarks for relative termination of string rewriting, to encourage the search for more powerful proof methods and their automation.

References

1 Alfons Geser. *Relative Termination*. Dissertation, Universität Passau, Germany, 1990. URL: http://vts.uni-ulm.de/docs/2012/8146/vts_8146_11884.pdf.

2 Alfons Geser. *Is Termination Decidable for String Rewriting With only One Rule*. Habilitationsschrift, Universität Tübingen, 2001.

3 Alfons Geser, Dieter Hofbauer, and Johannes Waldmann. Sparse tiling through overlap closures for termination of string rewriting. In Herman Geuvers, editor, *4th Intl. Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*, volume 131 of *LIPIcs*, pages 21:1–21:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.FSCD.2019.21.

4 Alfons Geser, Johannes Waldmann, and Mario Wenzel. Symbolic enumeration of one-rule string rewriting systems. In A. Middeldorp and R. Thiemann, editors, *Proc. 15th Intl. Workshop on Termination, WST-16*, pages 8:1–8:5, Obergurgl, Austria, 2016.

5 Alfons Geser and Hans Zantema. Non-looping string rewriting. *ITA*, 33(3):279–302, 1999. doi:10.1051/ita:1999118.

6 Dieter Hofbauer and Johannes Waldmann. Termination of string rewriting with matrix interpretations. In F. Pfenning, editor, *Term Rewriting and Applications, 17th Intl. Conference, RTA 2006, Seattle, WA, USA, August 12-14, 2006, Proceedings*, volume 4098 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2006. doi:10.1007/11805618_25.

7 Jan Willem Klop. Term rewriting systems: A tutorial. *Bull. EATCS*, 32:143–183, 1987.

8 Adam Koprowski and Johannes Waldmann. Arctic termination ... below zero. In A. Voronkov, editor, *Rewriting Techniques and Applications, 19th Intl. Conference, RTA 2008, Hagenberg, Austria, July 15-17, 2008, Proceedings*, volume 5117 of *Lecture Notes in Computer Science*, pages 202–216. Springer, 2008. doi:10.1007/978-3-540-70590-1_14.

9 Winfried Kurth. *Termination und Konfluenz von Semi-Thue-Systemen mit nur einer Regel*. Dissertation, Technische Universität Clausthal, 1990.

10 Johannes Waldmann and Bertram Felgenhauer. ICFP Programming Contest 2010 – International Cars and Fuels Production. Talk at Intl. Conf. Functional Programming, Baltimore, Maryland, USA, 2010. https://www.imn.htwk-leipzig.de/~waldmann/talk/10/icfp/slides.pdf.

11 Hans Zantema. Relative termination in term rewriting. In M. Codish and A. Middeldorp, editors, *Proc. 7th Intl. Workshop on Termination, WST 2004, Aachen, Germany, 2004, Aachener Informatik Berichte AIB-2004-07*, pages 51–54, 2004.

Linear Termination over $\mathbb{N}$ is Undecidable

Fabian Mitterwallner
University of Innsbruck, Innsbruck, Austria

Aart Middeldorp
University of Innsbruck, Innsbruck, Austria

René Thiemann
University of Innsbruck, Innsbruck, Austria

Abstract

Recently it was shown that it is undecidable whether a term rewrite system can be proved terminating by a polynomial interpretation in the natural numbers. In this paper we show that this is also the case when restricting the interpretations to linear polynomials, as is often done in tools, and when only considering single-rule rewrite systems. What is more, the new undecidability proof is simpler than the previous one. We further show that polynomial termination over the rationals/reals is undecidable.

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

Keywords and phrases term rewriting, polynomial termination, undecidability

Acknowledgements We thank Nao Hirokawa for posing the question about linear interpretations.

1 Introduction

In a recent paper [3] the problem of whether a finite term rewrite system (TRS) can be proved terminating by a polynomial interpretation in the naturals numbers was shown to be undecidable. The result was strengthened by restricting the instance to incremental polynomially terminating TRSs. Moreover, incremental polynomial termination over $\mathbb{N}$ is an undecidable property of terminating TRSs.

In this paper we complement these results by proving the somewhat surprising result that the problem remains undecidable if we restrict the allowed interpretation functions to linear ones, even for single-rule polynomially terminating TRSs. A second contribution is that the termination problem is undecidable if we consider polynomial interpretations over the rationals and reals. Our undecidability proofs are surprisingly simple.

The results in this paper are obtained by a reduction from a variant of Hilbert's tenth problem. Hilbert's tenth is one of 23 problems published by David Hilbert in 1900 which where all unsolved at the time. To solve the tenth problem one should find an algorithm that given a diophantine equation with integer coefficients determines if the equation has a solution in the integers [1]. As is turns out this is impossible and the underlying decision problem was proved undecidable by Matijasevic in 1970 [2].

To simplify the encoding of Hilbert's tenth problem, we first reduce it to a slightly modified decision problem. Instead of using an arbitrary integer polynomial, we consider two polynomials P and Q with only positive coefficients and ask if $P(x_1, \ldots, x_n) \geqslant Q(x_1, \ldots, x_n)$ for some arguments $x_1, \ldots, x_n \in \mathbb{N}_+$. This is also undecidable and is more easily applicable in the proofs related to polynomial termination.

▶ **Lemma 1.** *The following decision problem is undecidable:*
instance: *two polynomials P and Q with positive integer coefficients*
question: $P(x_1, \ldots, x_n) \geqslant Q(x_1, \ldots, x_n)$ *for some* $x_1, \ldots, x_n \in \mathbb{N}_+$ *?*

Proof. We proceed by a reduction from Hilbert's 10th problem. Assume the decision problem is decidable and let $R \in \mathbb{Z}[x_1, \dots, x_n]$ be a polynomial. We can modify Hilbert's 10th problem for R as follows:

$$\begin{aligned} & \exists\, x_1, \dots, x_n \in \mathbb{Z}\ R(x_1, \dots, x_n) = 0 \\ \iff\ & \exists\, x_1, \dots, x_n \in \mathbb{Z}\ R(x_1, \dots, x_n)^2 \leqslant 0 \\ \iff\ & \exists\, a_1, \dots, a_n \in \{-1, 0, 1\}\ \exists\, x_1, \dots, x_n \in \mathbb{N}_+\ R(a_1 x_1, \dots, a_n x_n)^2 \leqslant 0 \end{aligned} \tag{1}$$

We can now split $R(a_1 x_1, \dots, a_n x_n)^2$ into two polynomials $P_{\vec{a}}$ and $Q_{\vec{a}}$ containing only positive coefficients, such that $R(a_1 x_1, \dots, a_n x_n)^2 = Q_{\vec{a}}(x_1, \dots, x_n) - P_{\vec{a}}(x_1, \dots, x_n)$. Hence (1) is equivalent to

$$\exists\, a_1, \dots, a_n \in \{-1, 0, 1\}\ \exists\, x_1, \dots, x_n \in \mathbb{N}_+\ P_{\vec{a}}(x_1, \dots, x_n) \geqslant Q_{\vec{a}}(x_1, \dots, x_n)$$

The final problem is decidable by our assumption, since it consists of 3^n instances of the decision problem. This contradicts the undecidability of Hilbert's 10th problem, thereby proving the lemma. ◀

2 Undecidability of Linear Termination over $\mathbb{N}$

To prove undecidability of linear termination we define a TRS $\mathcal{R}$ which is parameterized by two polynomials P and Q containing only positive coefficients. We then prove that $\mathcal{R}$ can be shown to be terminating using a linear interpretation if and only of $P(x_1, \dots, x_n) \geqslant Q(x_1, \dots, x_n)$ for some $x_1, \dots, x_n \in \mathbb{N}_+$. For polynomials containing the indeterminates $v_1, \dots, v_n$, the signature of $\mathcal{R}$ is $\mathcal{F} = \{\mathsf{z}, \mathsf{o}, \mathsf{a}, \mathsf{f}, \mathsf{v}_1, \dots, \mathsf{v}_n\}$, where z and o are constants, $\mathsf{v}_1, \dots, \mathsf{v}_n$ are unary symbols, a is a binary symbol and f has arity four.

To this end we first define an encoding $\ulcorner \cdot \urcorner^x$, which maps polynomials with positive coefficients to terms containing the variable x.

▶ **Definition 2.** *Let P be a polynomial containing only positive coefficients, and the indeterminates $v_1, \dots, v_n$. We can then encode natural numbers as*

$$\ulcorner 0 \urcorner^x = \mathsf{z} \qquad\qquad \ulcorner m + 1 \urcorner^x = \mathsf{a}(x, \ulcorner m \urcorner^x)$$

A monomial $M = c \cdot v_1^{m_1} \cdot v_2^{m_2} \cdots v_n^{m_n}$ with $c \in \mathbb{N}_+$ and $m_1, \dots, m_n \in \mathbb{N}$ is encoded as

$$\ulcorner M \urcorner^x = \mathsf{v}_1^{m_1}(\mathsf{v}_2^{m_2}(\cdots(\mathsf{v}_\mathsf{n}^{m_n}(\ulcorner c \urcorner^x))\cdots))$$

where $v^0(t) = t$ and $v^{i+1}(t) = v(v^i(t))$ for $v \in \{\mathsf{v}_1, \dots, \mathsf{v}_n\}$. Finally the polynomial $P = M_1 + M_2 + \cdots + M_k$ is encoded as

$$\ulcorner P \urcorner^x = \mathsf{a}(\ulcorner M_1 \urcorner^x, \mathsf{a}(\ulcorner M_2 \urcorner^x, \cdots \mathsf{a}(\ulcorner M_k \urcorner^x, \mathsf{z}) \cdots))$$

▶ **Example 3.** For the polynomial $P = X^3 + 2X + 2$ we obtain the term

$$\ulcorner P \urcorner^y = \mathsf{a}(\underbrace{\mathsf{X}(\mathsf{X}(\mathsf{X}(\mathsf{a}(y, \mathsf{z}))))}_{\ulcorner X^3 \urcorner}, \mathsf{a}(\underbrace{\mathsf{X}(\mathsf{a}(y, \mathsf{a}(y, \mathsf{z})))}_{\ulcorner 2X \urcorner}, \mathsf{a}(\underbrace{\mathsf{a}(y, \mathsf{a}(y, \mathsf{z}))}_{\ulcorner 2 \urcorner}, \mathsf{z})))$$

The TRS $\mathcal{R}$ can then be defined via this encoding.

▶ **Definition 4.** *For polynomials P and Q containing only positive coefficients we obtain the* TRS $\mathcal{R}$ *consisting of the single rule*

$$\mathsf{f}(y_1, y_2, \mathsf{a}(\ulcorner P \urcorner^{y_3}, y_3), \mathsf{o}) \to \mathsf{f}(\mathsf{a}(y_1, \mathsf{z}), \mathsf{a}(\mathsf{z}, y_2), \mathsf{a}(\ulcorner Q \urcorner^{y_3}, y_3), \mathsf{z})$$

The rule serves two purposes. First it constrains any linear interpretation proving its termination to conform to a very limited shape. Secondly it uses these restricted shapes to encode the inequality $P \geqslant Q$ in the orientation of the rule $[\ell]_{\mathbb{N}} > [r]_{\mathbb{N}}$. This leads to the following result.

▶ **Theorem 5.** *Termination of $\mathcal{R}$ can be shown by a linear interpretation if and only if $P(v_1, \ldots, v_n) \geqslant Q(v_1, \ldots, v_n)$ for some $v_1, \ldots, v_n \in \mathbb{N}_+$.*

Proof. For the if direction assume $P(v_1, \ldots, v_n) \geqslant Q(v_1, \ldots, v_n)$ for some $v_1, \ldots, v_n \in \mathbb{N}_+$. We then choose the monotone interpretations

$$\mathsf{z}_{\mathbb{N}} = 0 \qquad \mathsf{a}_{\mathbb{N}}(x_1, x_2) = x_1 + x_2 \qquad \mathsf{v}_{i\mathbb{N}}(x) = v_i \cdot x \quad \text{for all } i \in \{1, \ldots, n\}$$
$$\mathsf{o}_{\mathbb{N}} = 1 \qquad \mathsf{f}_{\mathbb{N}}(x_1, x_2, x_3, x_4) = x_1 + x_2 + x_3 + x_4$$

Note that using this interpretation we have $[\ulcorner P^{\neg y_3}]_{\mathbb{N}} = P(v_1, \ldots, v_n) \cdot y_3$, and the same holds for Q. Hence we orient the rule in $\mathcal{R}$, as seen in

$$[\ell]_{\mathbb{N}} = y_1 + y_2 + (P(v_1, \ldots, v_n) + 1)y_3 + 1 > y_1 + y_2 + (Q(v_1, \ldots, v_n) + 1)y_3 = [r]_{\mathbb{N}}$$

For the only-if direction we assume a linear interpretation for all $f \in \mathcal{F}$, such that $[\ell]_{\mathbb{N}} > [r]_{\mathbb{N}}$. Hence we know that all interpretations have the shape $f_{\mathbb{N}}(x_1, \ldots, x_k) = f_0 + f_1 x_1 + \cdots + f_k x_k$ where $f_0 \in \mathbb{N}$ and $f_1, \ldots, f_k \in \mathbb{N}_+$ due to monotonicity. For any term t we write $[t]_{\mathbb{N}}^{y_i}$ for the coefficient of the indeterminate y_i of the linear polynomial $[t]_{\mathbb{N}}$. Using this notation, $[\ell]_{\mathbb{N}} > [r]_{\mathbb{N}}$ implies $[\ell]_{\mathbb{N}}^{y_i} \geqslant [r]_{\mathbb{N}}^{y_i}$ for $i \in \{1, 2, 3\}$. By the shape of the rule we deduce $\mathsf{f}_1 = [\ell]_{\mathbb{N}}^{y_1} \geqslant [r]_{\mathbb{N}}^{y_1} = \mathsf{f}_1\mathsf{a}_1$ and in combination with $\mathsf{f}_1 > 0$ and $\mathsf{a}_1 > 0$ we conclude $\mathsf{a}_1 = 1$. Similarly, from $[\ell]_{\mathbb{N}}^{y_2} \geqslant [r]_{\mathbb{N}}^{y_2}$ we infer $\mathsf{a}_2 = 1$, and in turn $\mathsf{a}_{\mathbb{N}}(x_1, x_2) = x_1 + x_2 + \mathsf{a}_0$ for some $\mathsf{a}_0 \in \mathbb{N}$. Due to the shape $\mathsf{a}_{\mathbb{N}}$ it is easy to see that $[\ulcorner m^{\neg y_3}]_{\mathbb{N}}^{y_3} = m$ for any $m \in \mathbb{N}$, $[c \cdot \ulcorner v_1^{m_1} \cdots v_n^{m_n \neg y_3}]_{\mathbb{N}}^{y_3} = c \cdot v_1^{m_1} \cdots v_n^{m_n}$ and further $[\ulcorner P^{\neg y_3}]_{\mathbb{N}}^{y_3} = P(v_1, \ldots, v_n)$ for any polynomial P. Hence

$$\mathsf{f}_3 \cdot (P(v_1, \ldots, v_n) + 1) = [\ell]_{\mathbb{N}}^{y_3} \geqslant [r]_{\mathbb{N}}^{y_3} = \mathsf{f}_3 \cdot (Q(v_1, \ldots, v_n) + 1)$$

Since $\mathsf{f}_3 > 0$, division by f_3 is possible, resulting in the desired inequality $P(v_1, \ldots, v_n) \geqslant Q(v_1, \ldots, v_n)$ for $v_1, \ldots, v_n \in \mathbb{N}_+$. ◀

▶ **Corollary 6.** *Linear termination is undecidable, even for single-rule* TRS*s.*

Proof. This follows directly from Theorem 5 and Lemma 1. ◀

Interestingly the TRS $\mathcal{R}$ is always terminating, independent of the polynomials P and Q. This can be shown using a (non-linear) polynomial interpretation.

▶ **Lemma 7.** *The* TRS $\mathcal{R}$ *is polynomially terminating.*

Proof. Use the following monotone interpretation over $\mathbb{N}$

$$\mathsf{o}_{\mathbb{N}} = Q(1, \ldots, 1) + 1 \qquad \mathsf{a}_{\mathbb{N}}(x, y) = x + y \qquad \mathsf{v}_{i\mathbb{N}}(x) = x \quad \text{for all } i \in \{1, \ldots, n\}$$
$$\mathsf{z}_{\mathbb{N}} = 0 \qquad \mathsf{f}_{\mathbb{N}}(x_1, x_2, x_3, x_4) = x_3x_4 + x_1 + x_2 + x_3 + x_4$$

Note that due to $[\mathsf{v}_i(x)]_{\mathbb{N}}^x = 1$ we have $[\ulcorner P^{\neg y_3}]_{\mathbb{N}}^{y_3} = P(1, \ldots, 1)$ and $[\ulcorner Q^{\neg y_3}]_{\mathbb{N}}^{y_3} = Q(1, \ldots, 1)$. Hence, we can orient the rule as seen in

$$\begin{aligned}[\ell]_{\mathbb{N}} &= (Q(1, \ldots, 1) + 1)(P(1, \ldots, 1) + 1)y_3 + \\ &\quad y_1 + y_2 + (P(1, \ldots, 1) + 1)y_3 + (Q(1, \ldots, 1) + 1) \\ &> y_1 + y_2 + (Q(1, \ldots, 1) + 1)y_3 = [r]_{\mathbb{N}}\end{aligned}$$ ◀

▶ **Corollary 8.** *Linear termination is undecidable, even for polynomially terminating single-rule systems.*

3 Polynomial Termination over $\mathbb{Q}$ and $\mathbb{R}$

When considering polynomial interpretations over $\mathbb{Q}$ and $\mathbb{R}$, we restrict the domain to all non-negative values. Moreover, when comparing the polynomials associated with the left- and right-hand side of a rewrite rule, we demand that the difference is at least δ, for some fixed positive value δ of the domain. This ensures termination. We refer to [4] for formal definitions as well as the relationship between the notions of polynomial termination over $\mathbb{N}$, $\mathbb{Q}$ and $\mathbb{R}$.

In the previous section we encoded polynomials as terms such that indeterminates of the polynomials correspond to coefficients of some interpretation. When dealing with polynomial termination over $\mathbb{Q}$ and $\mathbb{R}$ a new approach for proving undecidability is required, since coefficients take values in $\mathbb{Q}$ or $\mathbb{R}$. However, what does not change is that the exponents of our interpretations must still be natural numbers. We can make use of this by encoding the polynomials and the order on polynomials in the degrees of our interpretations. As long as we can represent multiplication in the interpretations we can use basic arithmetic to encode the polynomials in the degrees.

▶ **Lemma 9.** *If P and Q are univariate polynomials containing only positive coefficients then*

1. $\deg(P + Q) = \max(\deg(P), \deg(Q))$,
2. $\deg(P \cdot Q) = \deg(P) + \deg(Q)$, *and*
3. $\deg(P \circ Q) = \deg(P) \cdot \deg(Q)$. ◀

For encoding polynomials with positive coefficients as terms we use Definition 2, so using function symbols from $\{\mathsf{z}, \mathsf{a}\} \cup \{\mathsf{v}_i \mid 1 \leqslant i \leqslant n\}$. Moreover, we write $\ulcorner P \urcorner$ for $\ulcorner P \urcorner^x$ with some fixed variable x. The polynomial is then encoded in the degree of $\ulcorner P \urcorner$, as seen in the following lemma, which can be proved using a simple induction over Definition 2.

▶ **Lemma 10.** *Let $\mathcal{D} \in \{\mathbb{Q}, \mathbb{R}\}$ and suppose $\mathsf{z}_\mathcal{D} = z_0$ and $\mathsf{a}_\mathcal{D} = a_3xy + a_2x + a_1y + a_0$ for some $z_0, a_0 \in \mathcal{D}_{\geqslant 0}$ and $a_3, a_2, a_1 \in \mathcal{D}_{>0}$. If $P \in \mathbb{Z}[v_1, \dots, v_n]$ with positive coefficients then* $\deg([\ulcorner P \urcorner]_\mathcal{D}) = P(\deg([\mathsf{v}_1]_\mathcal{D}), \dots, \deg([\mathsf{v}_n]_\mathcal{D}))$.

Proof. We use induction on the definition of $\ulcorner P \urcorner$. If $P = 0$ then $[\ulcorner P \urcorner]_\mathcal{D} = \mathsf{z}$ and thus $\deg([\ulcorner P \urcorner]_\mathcal{D}) = 0 = P$. For $P = m + 1$ we obtain $\ulcorner P \urcorner = \mathsf{a}(x, \ulcorner m \urcorner)$ and thus

$$[\ulcorner P \urcorner]_\mathcal{D} = a_3 \cdot [\ulcorner m \urcorner]_\mathcal{D} \cdot x + a_2 \cdot x + a_1 \cdot [\ulcorner m \urcorner]_\mathcal{D} + a_0$$

Hence $\deg([\ulcorner P \urcorner]_\mathcal{D}) = \deg([\ulcorner m \urcorner]_\mathcal{D}) + 1 = m + 1$ by the induction hypothesis. For a monomial $P = c \cdot v_1^{m_1} \cdots v_n^{m_n}$ with $c \in \mathbb{N}_+$ and $m_1, \dots, m_n \in \mathbb{N}$ we obtain

$$\begin{aligned}\deg([\ulcorner P \urcorner]_\mathcal{D}) &= \deg([\ulcorner c \urcorner]_\mathcal{D}) \cdot \deg([\mathsf{v}_1]_\mathcal{D})^{m_1} \cdots \deg([\mathsf{v}_n]_\mathcal{D})^{m_n} \\ &= c \cdot \deg([\mathsf{v}_1]_\mathcal{D})^{m_1} \cdots \deg([\mathsf{v}_n]_\mathcal{D})^{m_n} \\ &= P(\deg([\mathsf{v}_1]_\mathcal{D}), \dots, \deg([\mathsf{v}_n]_\mathcal{D}))\end{aligned}$$

Finally, if $P = M_1 + \cdots + M_k$ then $\ulcorner P \urcorner = \mathsf{a}(\ulcorner M_1 \urcorner, \cdots \mathsf{a}(\ulcorner M_k \urcorner, \mathsf{z}) \cdots)$ and thus

$$\begin{aligned}\deg [\ulcorner P \urcorner]_\mathcal{D} &= \deg([\ulcorner M_1 \urcorner]_\mathcal{D}) + \cdots + \deg([\ulcorner M_k \urcorner]_\mathcal{D}) \\ &= M_1(\deg([\mathsf{v}_1]_\mathcal{D}), \dots, \deg([\mathsf{v}_n]_\mathcal{D})) + \cdots + M_k(\deg([\mathsf{v}_1]_\mathcal{D}), \dots, \deg([\mathsf{v}_n]_\mathcal{D})) \\ &= P(\deg([\mathsf{v}_1]_\mathcal{D}), \dots, \deg([\mathsf{v}_k]_\mathcal{D}))\end{aligned}$$ ◀

▶ **Definition 11.** *Given polynomials P and Q containing only positive coefficients and containing the indeterminates $v_1, \dots, v_n$, the* TRS $\mathcal{Q}$ *is defined over the signature* $\mathcal{F} = \{\mathsf{z}, \mathsf{a}, \mathsf{h}, \mathsf{q}, \mathsf{g}\} \cup \{\mathsf{v}_i \mid i \in \{1, \dots, n\}\}$ *and consists of the rules*

$$\begin{aligned}
\mathsf{q}(\mathsf{h}(x)) &\xrightarrow{1} \mathsf{h}(\mathsf{h}(\mathsf{q}(x))) & \mathsf{a}(x,x) &\xrightarrow{5} \mathsf{q}(x)\\
\mathsf{h}(x) &\xrightarrow{2} \mathsf{g}(x,x) & \mathsf{h}(x) &\xrightarrow{6} \mathsf{a}(\mathsf{z},x)\\
\mathsf{g}(\mathsf{q}(x),\mathsf{h}(\mathsf{h}(\mathsf{h}(x)))) &\xrightarrow{3} \mathsf{q}(\mathsf{g}(x,\mathsf{h}(\mathsf{z}))) & \mathsf{h}(x) &\xrightarrow{7} \mathsf{a}(x,\mathsf{z})\\
\mathsf{h}(\mathsf{q}(x)) &\xrightarrow{4} \mathsf{a}(x,x) & \mathsf{h}(\mathsf{a}(\ulcorner P \urcorner, x)) &\xrightarrow{8} \mathsf{a}(\ulcorner Q \urcorner, x)
\end{aligned}$$

The main idea behind this system is that rules (1) through (7) restrict the possible interpretations of $\mathsf{a}_\mathcal{D}$ and $\mathsf{h}_\mathcal{D}$ such that Lemma 10 is applicable, and that compatibility with rule (8) implies $P(v_1, \dots, v_n) \geqslant Q(v_1, \dots, v_n)$. The rules (1)–(7) are similar to ones already used in [5], where they restrict possible interpretations over $\mathbb{N}$, and in [4], where they are also applied to interpretations over $\mathbb{Q}$ and $\mathbb{R}$.

▶ **Theorem 12.** *For $\mathcal{D} \in \{\mathbb{Q}, \mathbb{R}\}$ and polynomials $P, Q \in \mathbb{Z}[x_1, \dots, x_n]$ with positive integer coefficients the* TRS $\mathcal{Q}$ *can be proved terminating using a polynomial interpretation over $\mathcal{D}$ if and only if $P(v_1, \dots, v_n) \geqslant Q(v_1, \dots, v_n)$ for some $v_1, \dots, v_n \in \mathbb{N}_+$.*

Proof. For the if direction assume $P(v_1, \dots, v_n) \geqslant Q(v_1, \dots, v_n)$ for some $v_1, \dots, v_n \in \mathbb{N}_+$. Take the interpretations

$$\begin{aligned}
\mathsf{z}_\mathcal{D} &= 0 & \mathsf{g}_\mathcal{D}(x,y) &= x + y & \mathsf{a}_\mathcal{D}(x,y) &= xy + x + y + 1\\
\mathsf{q}_\mathcal{D}(x) &= x^2 + 2x & \mathsf{h}_\mathcal{D}(x) &= hx + h & \mathsf{v}_{i\mathcal{D}}(x) &= x^{v_i} \quad \text{for all } i \in \{1, \dots, n\}
\end{aligned}$$

where $h > 2$. With $\delta = 1$ these orient the rules (1) – (7). Lemma 10 yields $\deg([\ulcorner P \urcorner]_\mathcal{D}) = P(\deg([\mathsf{v}_1]_\mathcal{D}), \dots, \deg([\mathsf{v}_n]_\mathcal{D})) = P(v_1, \dots, v_n)$ and similarly $\deg([\ulcorner Q \urcorner]_\mathcal{D}) = Q(v_1, \dots, v_n)$. From the assumption $P(v_1, \dots, v_n) \geqslant Q(v_1, \dots, v_n)$ we therefore obtain $\deg([\ulcorner P \urcorner]_\mathcal{D}) \geqslant \deg([\ulcorner Q \urcorner]_\mathcal{D})$. Consequently,

$$\deg([\mathsf{h}(\mathsf{a}(\ulcorner P \urcorner, x))]_\mathcal{D}) = \deg([\ulcorner P \urcorner]_\mathcal{D}) + 1 \geqslant \deg([\ulcorner Q \urcorner]_\mathcal{D}) + 1 = [\mathsf{a}(\ulcorner Q \urcorner, x)]_\mathcal{D}$$

It follows that by choosing the coefficient h large enough, the remaining rule (8) is oriented.

For the only-if direction assume the TRS $\mathcal{Q}$ is polynomially terminating over $\mathcal{D}$. From compatibility with rule (1) we infer

$$\deg([\mathsf{q}(\mathsf{h}(x))]_\mathcal{D}) = \deg(\mathsf{q}_\mathcal{D}) \cdot \deg(\mathsf{h}_\mathcal{D}) \geqslant \deg(\mathsf{q}_\mathcal{D}) \cdot \deg(\mathsf{h}_\mathcal{D})^2 = \deg[\mathsf{h}(\mathsf{h}(\mathsf{q}(x)))]_\mathcal{D}$$

Hence $\deg(\mathsf{h}_\mathcal{D}) = 1$ and thus $\mathsf{h}_\mathcal{D}(x) = h_1 x + h_0$ for some $h_1 \geqslant 1$ and $h_0 \geqslant 0$. From compatibility with rule (2) we infer $\deg(\mathsf{g}_\mathcal{D}) = 1$ and thus $\mathsf{g}_\mathcal{D}(x,y) = g_2 x + g_1 y + g_0$ with $g_2, g_1 \geqslant 1$. Moreover, $h_1 \geqslant g_1 + g_2 \geqslant 2$ and $h_0 >_\delta g_0 \geqslant 0$. Looking back at rule (1) we now can infer that $\mathsf{q}_\mathcal{D}$ is at least quadratic, since if it were linear we would obtain the inequality

$$q_1 h_1 \cdot x + q_1 h_0 + q_0 >_\delta q_1 h_1^2 \cdot x + h_1^2 q_0 + h_1 h_0 + h_0$$

for all $x \in \mathcal{D}_{\geqslant 0}$. This can only hold if $q_1 h_1 \geqslant q_1 h_1^2$, which in turn implies $h_1 \leqslant 1$, contradicting $h_1 \geqslant 2$. Next we show $\deg(\mathsf{q}_\mathcal{D}) = 2$. Compatibility with rule (3) induces the constraint $g_2 \cdot [\mathsf{q}(x)]_\mathcal{D} + g_1 \cdot [\mathsf{h}(\mathsf{h}(\mathsf{h}(x)))]_\mathcal{D} + g_0 >_\delta [\mathsf{q}(\mathsf{g}(x, \mathsf{h}(\mathsf{z})))]_\mathcal{D}$, which implies

$$1 = \deg([\mathsf{h}(\mathsf{h}(\mathsf{h}(x)))]_\mathcal{D} \geqslant \deg([\mathsf{q}(\mathsf{g}(x, \mathsf{h}(\mathsf{z})))]_\mathcal{D} - g_2 \cdot [\mathsf{q}(x)]_\mathbb{N})$$

Since $h_0 > 0$ and $[\mathsf{h}(\mathsf{z})]_{\mathcal{D}} > 0$ this can only be the case if $\deg(\mathsf{q}_{\mathcal{D}}) = 2$. Using this fact together with compatibility with the rules (4) and (5) we infer $\deg(\mathsf{a}_{\mathcal{D}}) = 2$ and hence $\mathsf{a}_{\mathcal{D}}(x, y) = a_5x^2 + a_4y^2 + a_3xy + a_2x + a_1y + a_0$. Compatibility with rules (6) and (7) implies $a_5 = a_4 = 0$ resulting in $\mathsf{a}_{\mathcal{D}}(x, y) = a_3xy+a_2x+a_1y+a_0$ with $a_3, a_2, a_1 \in \mathcal{D}_{>0}$. Compatibility with (8) implies $\deg([\ulcorner P \urcorner]_{\mathcal{D}}) + 1 \geqslant \deg([\ulcorner Q \urcorner]_{\mathcal{D}}) + 1$. With the help of Lemma 10 we obtain $P(\deg([\mathsf{v}_1]_{\mathcal{D}}), \ldots, \deg([\mathsf{v}_n]_{\mathcal{D}})) \geqslant Q(\deg([\mathsf{v}_1]_{\mathcal{D}}), \ldots, \deg([\mathsf{v}_n]_{\mathcal{D}}))$. ◀

▶ **Corollary 13.** *Polynomial termination over $\mathbb{Q}$ and $\mathbb{R}$ is undecidable.* ◀

References

1 David Hilbert. Mathematical problems. *Bulletin of the American Mathematical Society*, 8(10):437–479, 1902. doi:10.1090/S0002-9904-1902-00923-3.

2 Yuri Y. Matijasevic. Enumerable sets are diophantine (translated from Russian). In *Soviet Mathematics Doklady*, volume 11, pages 354–358, 1970.

3 Fabian Mitterwallner and Aart Middeldorp. Polynomial termination over $\mathbb{N}$ is undecidable. In *Proceedings of the 7th International Conference on Formal Structures for Computation and Deduction*, volume 228 of *Leibniz International Proceedings in Informatics*, pages 27:1–27:17, 2022. doi:10.4230/LIPIcs.FSCD.2022.27.

4 Friedrich Neurauter and Aart Middeldorp. Polynomial interpretations over the natural, rational and real numbers revisited. *Logical Methods in Computer Science*, 10(3:22):1–28, 2014. doi:10.2168/LMCS-10(3:22)2014.

5 Friedrich Neurauter, Aart Middeldorp, and Harald Zankl. Monotonicity criteria for polynomial interpretations over the naturals. In *Proceedings of the 5th International Joint Conference on Automated Reasoning*, volume 6173 of *Lecture Notes in Artificial Intelligence*, pages 502–517, 2010. doi:10.1007/978-3-642-14203-1_42.

Complexity Analysis for Call-by-Value Higher-Order Rewriting

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

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

Abstract

In this short paper, we consider a form of higher-order rewriting with a call-by-value evaluation strategy so as to model call-by-value programs. We briefly present a cost–size semantics to call-by-value rewriting: a class of algebraic interpretations that map terms to tuples that bound both the reductions' cost and the size of normal forms.

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

Keywords and phrases Call-by-Value Evaluation, Complexity Theory, Higher-Order Rewriting

Funding This work is supported by the NWO TOP project "Implicit Complexity through Higher-Order Rewriting", NWO 612.001.803/7571 and the NWO VIDI project "Constrained Higher-Order Rewriting and Program Equivalence", NWO VI.Vidi.193.075.

1 Introduction

This short paper is a brief exposition of the conference paper "Cost–Size Semantics for Call-by-Value Higher-Order Rewriting" recently published at FSCD 2023 [14]. We study *complexity* in this work, which in the context of term rewriting is typically understood as the number of steps needed to reach a normal from when starting in terms of a certain shape and size. A natural way to determine these bounds is by adapting techniques for proving termination to deduce the complexity. There is a myriad of works following this idea. To mention a few, see [2, 3, 5, 10, 11] for interpretation methods, [4, 9, 18] for lexicographic and path orders, and [8, 16] for dependency pairs. However, those ideas are focused on *first-order* term rewriting. The literature on the complexity of *higher-order* rewriting is scarce. While there is a lot of work studying the complexity of functional programs [1, 6, 12, 15], this work uses quite different ideas from the methods developed for term rewriting. It would be beneficial to combine these ideas.

In a previous work [13], we introduced an extension of the method of *weakly monotonic algebras* [7, 17] to *tuple interpretations*. This work deals with complexity analysis of higher-order rewriting in the context of full rewriting, i.e., no choice for the evaluation strategy. The idea of algebraic interpretations is to choose an interpretation domain A, and interpret terms s as elements $[\![s]\!]$ of A compositionally in such a way that whenever $s \to t$ we have $[\![s]\!] > [\![t]\!]$. Hence, a rewriting step on terms implies a strict decrease on A. The defining characteristic of tuple interpretations is to split the complexity measure into abstract notions of cost and size. This coincides with ideas often used in resource analysis of functional programs [1, 6]. This is a popular idea, as a very similar approach was introduced for first-order rewriting around the same time [19].

2 Preliminaries

The formalism we consider here is a style of simply typed lambda calculus extended with function symbols and rules. The matching mechanism is modulo alpha, and beta reduction

is included in the rewriting relation.

Let $\mathbb{B}$ be a nonempty set of *base types*. The set $\mathbb{T}(\mathbb{B})$ of *simple types* over $\mathbb{B}$ is generated by the grammar: $\mathbb{T}(\mathbb{B}) \coloneqq \mathbb{B} \mid \mathbb{T}(\mathbb{B}) \Rightarrow \mathbb{T}(\mathbb{B})$. As usual, we assume that the $\Rightarrow$ type constructor is right-associative. A *signature* $\mathbb{F}$ is a triple $(\mathbb{B}, \Sigma, \mathtt{ar})$ where $\mathbb{B}$ is a set of base types, Σ is a nonempty finite set of symbols, and $\mathtt{ar}$ is a function $\mathtt{ar} : \Sigma \longrightarrow \mathbb{T}(\mathbb{B})$. We postulate, for each type σ, the existence of a nonempty set $\mathbb{X}_\sigma$ of countably many variables. Furthermore, we impose that $\mathbb{X}_\sigma \cap \mathbb{X}_\tau = \emptyset$ whenever $\sigma \neq \tau$ and let $\mathbb{X}$ denote the family of sets $(\mathbb{X}_\sigma)_{\sigma \in \mathbb{T}(\mathbb{B})}$ indexed by $\mathbb{T}(\mathbb{B})$ and assume that $\Sigma \cap \mathbb{X} = \emptyset$.

The set $\mathsf{T}(\mathbb{F}, \mathbb{X})$ — of terms built from $\mathbb{F}$ and $\mathbb{X}$ — collects those expressions s for which the judgment $s : \sigma$ can be deduced using the following rules:

$$\frac{x \in \mathbb{X}_\sigma}{x : \sigma} \qquad \frac{\mathsf{f} \in \Sigma \qquad \mathtt{ar}(\mathsf{f}) = \sigma}{\mathsf{f} : \sigma} \qquad \frac{s : \sigma \Rightarrow \tau \qquad t : \sigma}{(s\,t) : \tau} \qquad \frac{x \in \mathbb{X}_\sigma \qquad s : \tau}{(\lambda x.\, s) : \sigma \Rightarrow \tau}$$

We assume the usual λ-calculus association and precedence scheme for application and abstraction. We shall remove unnecessary parentheses and write terms following those rules. Application of substitutions is defined as expected.

Call-by-Value Higher-order Rewriting A *rewrite rule* $\ell \to r$ is a pair of terms of the same type such that $\ell = \mathsf{f}\,\ell_1 \dots \ell_k$ and $\mathtt{fv}(r) \subseteq \mathtt{fv}(\ell)$, here $\mathtt{fv}(s)$ is the function mapping a term to the set of its free variables. A *term rewriting system* (TRS) $\mathbb{R}$ is a set of rules. In this paper, we are interested in a restricted evaluation strategy, which limits reduction to terms whose immediate subterms are *values*:

▶ **Definition 1.** A term s is a *value* whenever s is:
- of the form $\mathsf{f}\,v_1 \dots v_n$, with each v_i a value and there is no rule $\mathsf{f}\,\ell_1 \dots \ell_k \to r$ with $k \leq n$;
- an abstraction, i.e., $s = \lambda x.\, t$.

Every rewrite rule $\ell \to r$ *defines* a symbol f, namely, the head symbol of ℓ. For each $\mathsf{f} \in \Sigma$, let $\mathbb{R}_\mathsf{f}$ denote the set of rewrite rules that define f in $\mathbb{R}$. A symbol $\mathsf{f} \in \Sigma$ is a *defined symbol* if $\mathbb{R}_\mathsf{f} \neq \emptyset$. A *constructor symbol* is a symbol $\mathsf{c} \in \Sigma$ such that $\mathbb{R}_\mathsf{c} = \emptyset$. We let $\Sigma^{\mathtt{def}}$ be the set of defined symbols and $\Sigma^{\mathtt{con}}$ the set of constructor symbols. Hence, $\Sigma = \Sigma^{\mathtt{def}} \uplus \Sigma^{\mathtt{con}}$. A *ground constructor term* is a term $\mathsf{c}\,s_1 \dots s_n$ with $n \geq 0$, where each s_i is a ground constructor term.

Notice that by definition ground constructor terms are values since there is no rule $\mathsf{c}\ \ell_1\ \dots\ \ell_k \to r$ for any k if $\mathsf{c} \in \Sigma^{\mathtt{con}}$. More complex values include partially applied functions and lambda-terms; for example, $\mathsf{add}\ \mathsf{0}$ or a list of functions $[\mathsf{add}\ \mathsf{0}; \lambda x.x; \mathsf{mult}\ \mathsf{0}; \mathsf{dbl}]$.

▶ **Definition 2.** The **higher-order weak call-by-value rewrite relation** $\to_v$ induced by $\mathbb{R}$ is defined as follows:
- $\mathsf{f}\,(\ell_1\gamma) \dots (\ell_k\gamma) \to_v r\gamma$, if $\mathsf{f}\,\ell_1 \dots \ell_k \to r \in \mathbb{R}$ and each $\ell_i\gamma$ is a value;
- $(\lambda x.\, s)\, v \to_v s[x \coloneqq v]$, if v is a value;
- $s\,t \to_v s'\,t$ if $s \to_v s'$; and $s\,t \to_v s\,t'$ if $t \to_v t'$.

▶ **Example 3.** Let us consider two simple examples of functions encoded as rules. The first is map, which applies a function $F : \mathsf{nat} \Rightarrow \mathsf{nat}$.

$$\mathsf{map}\,F\,\mathsf{nil} \to \mathsf{nil} \qquad\qquad \mathsf{add}\,x\,\mathsf{0} \to \mathsf{0}$$
$$\mathsf{map}\,F\,(\mathsf{cons}\,x\,xs) \to \mathsf{cons}\,(F\,x)\,(\mathsf{map}\,F\,xs) \qquad\qquad \mathsf{add}\,x\,(\mathsf{s}\,y) \to \mathsf{s}\,(\mathsf{add}\,x\,y)$$

3 Cost–Size Semantics for Types and Terms

An interpretation of types is a function $(\!|\cdot|\!)$ that maps each type $\sigma \in \mathbb{T}(\mathbb{B})$ to a well-founded set $(\!|\sigma|\!)$, the cost–size interpretation of σ. In order to define such a function, we need an *interpretation key* function $K : \mathbb{B} \longrightarrow \mathbb{N}$ mapping base types ι to a number $K(\iota)$. This number sets the "length" of the tuples in the size interpretation of ι.

▶ **Definition 4** (Interpretation of Types)**.** We define for each type σ the **cost–size tuple interpretation** of σ as the set $(\!|\sigma|\!) = \mathcal{C}_\sigma \times \mathcal{S}_\sigma$ where $\mathcal{C}_\sigma$ and $\mathcal{S}_\sigma$ are defined as follows:

$$\mathcal{C}_\sigma = \mathbb{N} \times \mathcal{F}^{\mathsf{c}}_\sigma \qquad \mathcal{S}_\iota = \mathbb{N}^{K(\iota)}$$
$$\mathcal{F}^{\mathsf{c}}_\iota = \mathtt{unit} \qquad \mathcal{S}_{\sigma\Rightarrow\tau} = \mathcal{S}_\sigma \Longrightarrow \mathcal{S}_\tau$$
$$\mathcal{F}^{\mathsf{c}}_{\sigma\Rightarrow\tau} = (\mathcal{F}^{\mathsf{c}}_\sigma \times \mathcal{S}_\sigma) \Longrightarrow \mathcal{C}_\tau$$

The set $(\!|\sigma|\!)$ is ordered component-wise. With that this interpretation of types is well-founded, which was proved in the full version of this paper. Next, we need an *application operator* for applying cost–size tuples. More precisely, given a type $\sigma \Rightarrow \tau$ and cost–size tuples $\boldsymbol{f} \in (\!|\sigma \Rightarrow \tau|\!)$ and $\boldsymbol{x} \in (\!|\sigma|\!)$, we define the application of $\boldsymbol{f}$ to $\boldsymbol{x}$ as follows.

▶ **Definition 5.** Let $\sigma \Rightarrow \tau$ be an arrow type, $\boldsymbol{f} = \langle (n, f^{\mathsf{c}}), f^{\mathsf{s}} \rangle \in (\!|\sigma \Rightarrow \tau|\!)$, and $\boldsymbol{x} = \langle (m, x^{\mathsf{c}}), x^{\mathsf{s}} \rangle \in (\!|\sigma|\!)$. The **semantic application** of $\boldsymbol{f}$ to $\boldsymbol{x}$, denoted $\boldsymbol{f} \cdot \boldsymbol{x}$, is defined by:

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

An interpretation of a signature $\mathbb{F} = (\mathbb{B}, \Sigma, \mathtt{ar})$ interprets the base types in $\mathbb{B}$ and each $\mathsf{f} \in \Sigma$ of arity $\mathtt{ar}(\mathsf{f}) = \sigma$ as an element of $(\!|\sigma|\!)$ which is constructed by Definition 4.

▶ **Definition 6.** A **cost–size tuple interpretation** $\mathcal{F}$ for a signature $\mathbb{F} = (\mathbb{B}, \Sigma, \mathtt{ar})$ consists of a pair of functions $(K, \mathcal{J}_\Sigma)$ where

- K is a type interpretation key, which maps each base type ι to its tuple dimension $K(\iota)$
- $\mathcal{J}_\Sigma$ is an *interpretation of symbols* in Σ which maps each $\mathsf{f} \in \Sigma$ with $\mathtt{ar}(\mathsf{f}) = \sigma$ to a cost–size tuple in $(\!|\sigma|\!)$, where $(\!|\sigma|\!)$ is built using K in Definition 4.

In what follows we slightly abuse notation by writing $\mathcal{J}_\mathsf{f}$ for $\mathcal{J}_\Sigma(\mathsf{f})$ and just $\mathcal{J}$ for $\mathcal{J}_\Sigma$.

▶ **Example 7.** As a first example of interpretation, let us interpret the data constructors from Example 3. Recall that $\mathsf{0} : \mathsf{nat}, \mathsf{s} : \mathsf{nat} \Rightarrow \mathsf{nat}$ are the constructors for nat. We then set $K(\mathsf{nat}) = 1$.

$$\mathcal{J}_\mathsf{0} = \Big\langle \ (0, \mathtt{u}) \ , 1 \Big\rangle \qquad \mathcal{J}_\mathsf{s} = \Big\langle \ (0, \boldsymbol{\lambda} x.(0, \mathtt{u})) \ , \boldsymbol{\lambda} x.x + 1 \Big\rangle$$

The highlighted cost components for the constructors are filled with zeroes. That is because in the rewriting cost model data values do not fire rewriting sequences. Intuitively, the *cost number* for $\mathsf{0}$ is 0, (because it is a value), the *cost function* is the unit element $\mathtt{u} \in \mathtt{unit}$, (because it has base type), and *size component* is 1 (since we chose a notion of size for terms of type nat to mean "number of symbols"). The cost number for s is 0, the cost function is the constant function mapping to 0, and the size component is the function $\boldsymbol{\lambda} x.x + 1$ in $\mathcal{S}_{\mathsf{nat}\Rightarrow\mathsf{nat}}$. We interpret the constructors for list, i.e., nil and cons, following the same principle, with $K(\mathsf{list}) = 2$. We write a size tuple q in $\mathcal{S}_\mathsf{list}$ as $(q_\mathsf{l}, q_\mathsf{m})$ since the first component is to mean the length of the list and the second a bound on the size of its elements.

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

The highlighted cost components are filled with zeroes for lists as well. Size components are interpreted following the semantics we set for the two size components lenght and maximum element size, respectively.

The next step is to extend the interpretation of a signature $\mathbb{F}$ to the set of terms. But first, we define *valuation functions* to interpret the variables in $x : \sigma$ as elements of $(\!|\sigma|\!)$.

▶ **Definition 8.** A **cost–size valuation** α is a function that maps each $x : \sigma$ to a cost-size tuple in $(\!|\sigma|\!)$ such that:

- $\alpha(x) = \langle (0, \mathtt{u}), x^{\mathsf{s}} \rangle$, for all $x \in \mathbb{X}$ of base type; and $\alpha(F) = \langle (0, F^{\mathsf{c}}), F^{\mathsf{s}} \rangle$ when $F :: \sigma \Rightarrow \tau$.

▶ **Definition 9.** Assume given a signature $\mathbb{F} = (\mathbb{B}, \Sigma, \mathtt{ar})$ and its cost–size tuple interpretation $\mathcal{F} = (K, \mathcal{J})$ together with a valuation α. The **term interpretation** $[\![s]\!]^{\mathcal{J}}_{\alpha}$ of s under $\mathcal{J}$ and α is defined by induction on the structure of s as follows:

$$[\![x]\!]^{\mathcal{J}}_{\alpha} = \alpha(x) \qquad [\![\mathsf{f}]\!]^{\mathcal{J}}_{\alpha} = \mathcal{J}_{\mathsf{f}} \qquad [\![s\,t]\!]^{\mathcal{J}}_{\alpha} = [\![s]\!]^{\mathcal{J}}_{\alpha} \cdot [\![t]\!]^{\mathcal{J}}_{\alpha}$$
$$[\![\lambda x.\, s]\!]^{\mathcal{J}}_{\alpha} = \left\langle \left(0, \boldsymbol{\lambda} d.(1 + \pi_{11}([\![s]\!]^{\mathcal{J}}_{[x:=d]\alpha}), \pi_{12}([\![s]\!]^{\mathcal{J}}_{[x:=d]\alpha})\right), \boldsymbol{\lambda} d^{\mathsf{s}}.\pi_2([\![s]\!]^{\mathcal{J}}_{[x:=(\underline{0},d)]\alpha}) \right\rangle,$$

where π_i is the projection on the ith-component and π_{ij} is the composition $\pi_j \circ \pi_i$, and $\underline{0}$ is a cost function of the form $\boldsymbol{\lambda} x_1.(0, \boldsymbol{\lambda} x_2 \ldots (0, \mathtt{u}) \ldots)$. If $d = (d^c, d^s)$, the notation $[x := d]\alpha$ denotes the valuation that maps x to $\langle (0, d^c), d^s \rangle$ and every other variable y to $\alpha(y)$.

We write $[\![s]\!]$ for $[\![s]\!]^{\mathcal{J}}_{\alpha}$ whenever α and $\mathcal{J}$ are universally quantified or clear from the context.

The interpretation for abstractions may seem baroque, but can be understood as follows: an abstraction is a value, so its cost number is 0. The cost of applying that abstraction on a value v is 1 plus the cost number for $s[x := v]$ – which is obtained by evaluating $[\![s]\!]^{\mathcal{J}}_{[x:=d]\alpha}$ if d is the cost function/size pair for v. The cost *function* of this application is exactly the cost function of $s[x := v]$. The *size* of an abstraction $\lambda x.s$ is exactly the function that takes a size and maps it to the size interpretation of s where x is mapped to that size. Technically, to obtain the size component of $[\![s]\!]^{\mathcal{J}}_{[x:=d]\alpha}$ we also need a cost component, but by definition, this component does not play a role, so we can safely choose an arbitrary pair $\underline{0}$ in the right set.

▶ **Example 10.** We continue with Example 7 by interpreting ground constructor terms fully. A ground constructor term d of type nat is of the form $\mathsf{s}\,(\mathsf{s} \ldots (\mathsf{s}\,\mathsf{0}) \ldots)$ where the number $n \in \mathbb{N}$ is represented by n successive applications of s to $\mathsf{0}$. Let us write n as shorthand notation for such terms. Similarly, for ground constructor terms of type list, we write $[\mathsf{n}_1; \ldots; \mathsf{n}_k]$ for the term $\mathsf{cons}\,\mathsf{n}_1 \ldots (\mathsf{cons}\,\mathsf{n}_k\,\mathsf{nil})$. The empty list constructor nil is written as [] in this notation. Hence, the cost–size interpretation of $\mathsf{3} : \mathsf{nat}$ is given by:

$$[\![\mathsf{3}]\!] = [\![\mathsf{s}\,(\mathsf{s}\,(\mathsf{s}\,\mathsf{0}))]\!] = [\![\mathsf{s}]\!] \cdot ([\![\mathsf{s}]\!] \cdot ([\![\mathsf{s}]\!] \cdot [\![\mathsf{0}]\!])) = \left\langle \ (0, \mathtt{u}) \ , 4 \right\rangle.$$

Consider, for instance, the list $[\mathsf{1}; \mathsf{7}; \mathsf{9}]$. Its cost–size interpretation is given by:

$$[\![[\mathsf{1}; \mathsf{7}; \mathsf{9}]]\!] = [\![\mathsf{cons}\,\mathsf{1}\,(\mathsf{cons}\,\mathsf{7}\,(\mathsf{cons}\,\mathsf{9}\,\mathsf{nil}))]\!] = \left\langle \ (0, \mathtt{u}) \ , (3, 10) \right\rangle.$$

The important information we can extract from such interpretations is their size component. Indeed, $[\![\mathsf{3}]\!]^{\mathsf{s}} = 4$ counts the number of constructor symbols in the term representation $\mathsf{3}$ and $[\![[\mathsf{1}; \mathsf{7}; \mathsf{9}]]\!]^{\mathsf{s}} = (3, 10)$ gives us the length and an upper bound on the size of each element in $[\mathsf{1}; \mathsf{7}; \mathsf{9}]$. The size interpretation for the constructors of nat and list correctly capture our notion of "size" given earlier.

We give a concrete cost–size interpretation for map and add below:

$$\mathcal{J}_{\mathsf{add}} = \left\langle \ (0, \boldsymbol{\lambda} x.(0, \boldsymbol{\lambda} y.(y^{\mathsf{s}}, \mathtt{u}))) \ , \boldsymbol{\lambda} xy.x + y \right\rangle.$$

$$\mathcal{J}_{\mathsf{map}} = \left\langle \ (0, \boldsymbol{\lambda} F.(0, \boldsymbol{\lambda} q.(q_{\mathsf{l}} + F^{\mathsf{c}}(\mathtt{u}, q_{\mathsf{m}})q_{\mathsf{l}} + 1, \mathtt{u}))) \ , \boldsymbol{\lambda} F q.(q_{\mathsf{l}}, F(q_{\mathsf{m}})) \right\rangle,$$

4 Complexity Analysis of Call-by-Value Rewriting

Since our analysis is quantitative, our goal is not merely to find tuple interpretations that prove termination but also ones that provide "good" upper bounds on the complexity of reducing terms. To start, we will extend the notion of *derivation height* to our setting:

▶ **Definition 11.** The weak call-by-value **derivation height** of a term s, notation $\mathtt{dh}_{\mathbb{R}}(s)$, is the largest number n such that $s \to_v s_1 \to_v \ldots \to_v s_n$.

This notion is defined for all terms when the TRS is terminating. The methodology of weakly monotonic algebras offers a systematic way to derive bounds for the derivation height of a given term:

▶ **Lemma 12.** If $[\![s]\!] = \langle (n, F^{\mathsf{c}}), F^{\mathsf{s}} \rangle$, then $\mathtt{dh}_{\mathbb{R}}(s) \leq n$.

As an illustration of how this is used, let us complete the interpretation of Example 3. We start with the system $\mathbb{R}_{\mathsf{add}}$. We will use the type and constructor interpretations as given in Example 7. The rules in $\mathbb{R}_{\mathsf{add}}$ suggest the following cost–size interpretation:

$$\mathcal{J}_{\mathsf{add}} = \left\langle \; (0, \boldsymbol{\lambda} x.(0, \boldsymbol{\lambda} y.(y^{\mathsf{s}}, \mathtt{u}))) \; , \boldsymbol{\lambda} xy.x + y \right\rangle .$$

Notice that the (highlighted) cost component of $\mathcal{J}_{\mathsf{add}}$ suggest a linear cost measure for computing with add. We also set the intermediate numeric components in the cost tuple to zero. The reason for this choice is that in a cost tuple $\mathcal{C}_\sigma = \mathbb{N} \times \mathcal{F}^{\mathsf{c}}_\sigma$, the numeric component $\mathbb{N}$ captures the cost of partially applying terms, which is 0 in this case.

Now, consider the partially applied term $s = \mathsf{add}\,(\mathsf{add}\,2\,3)$ (of type $\mathsf{nat} \Rightarrow \mathsf{nat}$). Intuitively, the cost of reducing this term to normal form, is the cost of reducing the subterm $\mathsf{add}\,2\,3$ to 5, since the partially applied term $\mathsf{add}\,5$ cannot be reduced. Hence, $\mathtt{dh}_{\mathbb{R}}(s) = 4$. This is also the bound we find through interpretation:

$$\begin{aligned}
[\![s]\!] &= [\![\mathsf{add}]\!] \cdot ([\![\mathsf{add}]\!] \cdot [\![2]\!] \cdot [\![3]\!]) \\
&= [\![\mathsf{add}]\!] \cdot \langle (4, \mathtt{u}), 7 \rangle \\
&= \left\langle \; (4, \boldsymbol{\lambda} y.(y^{\mathsf{s}}, \mathtt{u})) \; , \boldsymbol{\lambda} y.7 + y \right\rangle .
\end{aligned}$$

While in this case the upper bound we find is tight, this is not always the case; for instance $[\![\mathsf{add}\ 0\ (\mathsf{add}\ 0\ 0)]\!] = \langle (3, \mathtt{u}), 3 \rangle$, even though $\mathtt{dh}_{\mathbb{R}}(\mathsf{add}\ 0\ (\mathsf{add}\ 0\ 0)) = 2$. We could obtain a tight upper bound by choosing a different interpretation, but this is also not always possible.

With this observation, we get a framework that provides us with a systematic approach to establish bounds to the complexity of weak call-by-value systems. The difficulty now lies in developing techniques to find suitable interpretation shapes. For instance, a first example of a higher-order function over lists is that of map. We give a concrete cost–size interpretation for map below:

$$\mathcal{J}_{\mathsf{map}} = \left\langle \; (0, \boldsymbol{\lambda} F.(0, \boldsymbol{\lambda} q.(q_{\mathsf{l}} + F^{\mathsf{c}}(\mathtt{u}, q_{\mathsf{m}})q_{\mathsf{l}} + 1, \mathtt{u}))) \; , \boldsymbol{\lambda} F q.(q_{\mathsf{l}}, F(q_{\mathsf{m}})) \right\rangle ,$$

The highlighted cost component accounts for q_{l} possible β steps, the cost of applying the higher-order argument F over the list q is bounded by $F^{\mathsf{c}}(\mathtt{u}, q_{\mathsf{m}})q_{\mathsf{l}}$ since F^{c} is assumed to be weakly monotonic, and the unitary component is for dealing with the empty list case.

5 Conclusions

In this short paper we briefly discussed an interpretation method for higher-order rewriting with weak call-by-value reduction. In this approach, we build on existing work defining tuple interpretations [13, 19], but restrict the evaluation strategy, and define a cost–size semantics for types and terms which generate a whole new class of cost–size semantic techniques that can be used to reason about the complexity of weak call-by-value systems.

References

1 M. Avanzini and U. Dal Lago. Automating sized-type inference for complexity analysis. In *Proc. ICFP*, 2017. doi:10.1145/3110287.

2 P. Baillot and U. Dal Lago. Higher-order interpretations and program complexity. *IC*, 2016. doi:10.1016/j.ic.2015.12.008.

3 G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet. Algorithms with polynomial interpretation termination proof. *Journal of Functional Programming*, 11(1):33–53, 2001. doi:10.1017/S0956796800003877.

4 G. Bonfante, J. Marion, and J. Moyen. On lexicographic termination ordering with space bound certifications. In *Proc. PSI*, 2001. doi:10.1007/3-540-45575-2_46.

5 A. Cichon and P. Lescanne. Polynomial interpretations and the complexity of algorithms. In *CADE*, pages 139–147, 1992. doi:10.1007/3-540-55602-8_161.

6 N. Danner, D.R. Licata, and R. Ramyaa. Denotational cost semantics for functional languages with inductive types. In *Proc. ICFP*, 2015. doi:10.1145/2784731.2784749.

7 C. Fuhs and C. Kop. Polynomial interpretations for higher-order rewriting. In *Proc. RTA*, 2012. doi:10.4230/LIPIcs.RTA.2012.176.

8 N. Hirokawa and G. Moser. Automated complexity analysis based on the dependency pair method. In *Proc. IJCAR*, 2008. doi:10.1007/978-3-540-71070-7_32.

9 D. Hofbauer. Termination proofs by multiset path orderings imply primitive recursive derivation lengths. *TCS*, 1992. doi:10.1007/3-540-53162-9_50.

10 D. Hofbauer. Termination proofs by context-dependent interpretations. In *Proc. RTA*, 2001. doi:10.1007/3-540-45127-7_10.

11 D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations. In *Proc. RTA*, 1989. doi:10.1007/3-540-51081-8_107.

12 D. M. Kahn and J. Hoffmann. Exponential automatic amortized resource analysis. In *Proc. FoSSaCS*, 2020. doi:10.1007/978-3-030-45231-5_19.

13 C. Kop and D. Vale. Tuple interpretations for higher-order complexity. In *Proc. FSCD*, 2021. doi:10.4230/LIPIcs.FSCD.2021.31.

14 Cynthia Kop and Deivid Vale. Cost-Size Semantics for Call-By-Value Higher-Order Rewriting. In *Proc. FSCD 2023*, pages 15:1–15:19, 2023. doi:10.4230/LIPIcs.FSCD.2023.15.

15 Y. Niu and J. Hoffmann. Automatic space bound analysis for functional programs with garbage collection. In *Proc. LPAR*, 2018. doi:10.29007/xkwx.

16 L. Noschinski, F. Emmes, and J. Giesl. A dependency pair framework for innermost complexity analysis of term rewrite systems. In *CADE-23*, pages 422–438, 2011. doi:10.1007/978-3-642-22438-6_32.

17 J.C. van de Pol. *Termination of Higher-order Rewrite Systems*. PhD thesis, University of Utrecht, 1996. URL: https://www.cs.au.dk/~jaco/papers/thesis.pdf.

18 A. Weiermann. Termination proofs for term rewriting systems by lexicographic path orderings imply multiply recursive derivation lengths. *TCS*, 1995. doi:10.1016/0304-3975(94)00135-6.

19 A. Yamada. Multi-dimensional interpretations for termination of term rewriting. In *In. Proc. CADE28*, volume 12699 of *Lecture Notes in Computer Science*, pages 273–290, 2021. doi:10.1007/978-3-030-79876-5_16.

Tool Descriptions

MultumNonMulta entering Term Rewriting

Dieter Hofbauer
ASW Saarland

Abstract

In this talk we report on recent attempts to generalize the tool MultumNonMulta (MnM) from string to term rewriting. Here, termination proofs are based on a reduction from term to string rewriting as described by Yamazaki [1], called *stringification*, combined with the concept of relative termination. Non-termination proofs rely on standard forward closures, enriched by initial substitutions, using MnM's data structures, which have already proven successful for string rewriting, see [2].

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

Keywords and phrases termination, relative termination, term rewriting, string rewriting

References

1 Kento Yamazaki. *Termination analysis of rewriting systems based on ligaturization and stringification*. Master thesis, School of Information Science, Japan Advanced Institute of Science and Technology, Japan, 2016. URL: `https://dspace.jaist.ac.jp/dspace/handle/10119/13610`.

2 Harald Zankl, Christian Sternagel, Dieter Hofbauer, and Aart Middeldorp. Finding and certifying loops. In *Proc. 36th Intl. Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM-10*, volume 5901 of *Lecture Notes in Computer Science*, pages 755–766. Springer, 2010. `doi:10.1007/978-3-642-11266-9_63`.

NTI+cTI: a Logic Programming Termination Analyzer

Fred Mesnard[1]
LIM, university of La Réunion, France

Étienne Payet[2]
LIM, university of La Réunion, France

Abstract

We describe NTI+cTI, our logic programming termination analyzer that takes part in the Termination Competition 2023. The tool is built from two separate components, NTI for *Non-Termination Inference* and cTI for *constraint-based Termination Inference*, plus an overall main process.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Termination, Non-termination, Logic Programming

1 NTI

NTI [9] is fully written in Java. It performs automated non-termination proofs of logic programs. It implements a technique that consists in unfolding [6] the program under analysis and in checking whether the produced unfolded clauses satisfy some non-termination criteria [11]. When a proof is successful, NTI provides an example of a non-terminating query. Two kinds of criteria are used.

- The first kind [11] relies on an extension of the "is more general than" relation. It is able to detect infinite derivations that consist of the repeated application of the same sequence ω of clauses, *i.e.*, of the form $Q_0 \Rightarrow_\omega Q_1 \Rightarrow_\omega \cdots$. If the body of an unfolded clause is more general than the head up to some predicate arguments in *neutral position*, then non-termination is detected; more precisely, every query obtained from replacing the neutral arguments of the head with ground terms is non-terminating. So, if such a non-terminating query belongs to the mode of interest then the proof is successful.
- The second kind [10] is able to detect infinite derivations that rely on two sequences ω_1 and ω_2 of clauses, *i.e.*, that have the form $Q_0(\Rightarrow^*_{\omega_1} \circ \Rightarrow_{\omega_2}) Q_1 (\Rightarrow^*_{\omega_1} \circ \Rightarrow_{\omega_2}) \cdots$. It consists in detecting pairs (c_1, c_2) of unfolded clauses of a particular form. Intuitively, c_1 and c_2 are mutually recursive and, in c_1, a context is removed from the head to the body while, in c_2, it is added again.

2 cTI

Termination analysis starts with applying termination inference as presented in [8] [3]. If the mode given in the moded query of interest of the analyzed program implies the inferred termination condition, termination is ensured. This first analysis relies on the *term-size* norm to abstract the logic program and on linear ranking functions, see e.g., [3] for a review. If necessary, termination inference is restarted using the same tool but by combining both the term-size norm and the *list-size* norm, as proposed in [5]. Combining these two norms

[1] Corresponding author
[2] Corresponding author
[3] Source code available here: `https://github.com/FredMesnard/cTI`

doubles the arity of each predicate, hence increases the analysis time so we use it in a second step.

In case these first attemps fail, we switch to BinTerm, the termination analyzer we've built for Java bytecode termination analysis [12]. BinTerm includes various termination tests: linear and eventual ranking functions [2], multi-dimensional linear ranking [1] and the size-change principle [7]. BinTerm analyzes binary Constrained Horn Clauses. Here how we map the original moded query and the original logic program to a binary Constrained Horn Clauses. The original logic program goes through a tabled left-to-right top-down mode analysis starting from the original moded query of interest. An abstract numeric constraint logic program is built using the term-size norm. A numerical model is computed [4]. From these three pieces, a binary Constrained Horn Clauses program is created by a tabled left-to-right top-down interpreter, which keeps only the input arguments of the predicates. Finally, this binary Constrained Horn Clauses program is analyzed by BinTerm.

Again, if necessary, a similar analysis is done by combining both the term-size and the list-size norms.

At last resort, a left-to-right top-down meta-interpreter with occurs-check computes a time-bounded SLD tree for the most general query. If the tree is finite and because we deal with logic programs, any query from the set of concrete queries abstracted by the original moded query terminates.

Otherwise, the termination analyzer cannot conclude.

3 NTI+cTI

The main process of our analyzer performs non-termination and termination analyses in parallel. It launches a thread that runs NTI and another thread that runs cTI. If one thread terminates successfully then the other one is stopped.

References

1 Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 117–133. Springer, 2010. URL: `https://doi.org/10.1007/978-3-642-15769-1_8`, `doi:10.1007/978-3-642-15769-1\_8`.

2 Roberto Bagnara and Fred Mesnard. Eventual linear ranking functions. In Ricardo Peña and Tom Schrijvers, editors, *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, pages 229–238. ACM, 2013. URL: `https://doi.org/10.1145/2505879.2505884`, `doi:10.1145/2505879.2505884`.

3 Roberto Bagnara, Fred Mesnard, Andrea Pescetti, and Enea Zaffanella. A new look at the automatic synthesis of linear ranking functions. *Inf. Comput.*, 215:47–67, 2012. URL: `https://doi.org/10.1016/j.ic.2012.03.003`, `doi:10.1016/j.ic.2012.03.003`.

4 Florence Benoy and Andy King. Inferring argument size relationships with CLP(R). In John P. Gallagher, editor, *Logic Programming Synthesis and Transformation, 6th International Workshop, LOPSTR'96, Stockholm, Sweden, August 28-30, 1996, Proceedings*, volume 1207 of *Lecture Notes in Computer Science*, pages 204–223. Springer, 1996. URL: `https://doi.org/10.1007/3-540-62718-9_12`, `doi:10.1007/3-540-62718-9\_12`.

5 Maurice Bruynooghe, Michael Codish, John P. Gallagher, Samir Genaim, and Wim Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM Trans.*

Program. Lang. Syst., 29(2):10, 2007. URL: https://doi.org/10.1145/1216374.1216378, doi:10.1145/1216374.1216378.

6 Michael Codish and Cohavit Taboch. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999. doi:10.1016/S0743-1066(99)00006-0.

7 Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In Chris Hankin and Dave Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 81–92. ACM, 2001. URL: https://doi.org/10.1145/360204.360210, doi:10.1145/360204.360210.

8 Fred Mesnard and Roberto Bagnara. cTI: A constraint-based termination inference tool for iso-prolog. *Theory Pract. Log. Program.*, 5(1-2):243–257, 2005. URL: https://doi.org/10.1017/S1471068404002017, doi:10.1017/S1471068404002017.

9 NTI (Non-Termination Inference). http://lim.univ-reunion.fr/staff/epayet/Research/NTI/NTI.html and https://github.com/etiennepayet/nti.

10 Étienne Payet. Binary non-termination in term rewriting and logic programming. In Akihisa Yamada, editor, *Submitted to the 19th International Workshop on Termination (WST'23)*, 2023.

11 Étienne Payet and Fred Mesnard. Non-termination inference of logic programs. *ACM Transactions on Programming Languages and Systems*, 28(2):256–289, 2006. doi:10.1145/1119479.1119481.

12 Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for java bytecode based on path-length. *ACM Trans. Program. Lang. Syst.*, 32(3):8:1–8:70, 2010. URL: https://doi.org/10.1145/1709093.1709095, doi:10.1145/1709093.1709095.