# 14th International Workshop on Termination (WST 2014)

Carsten Fuhs (Editor)

WST 2014, 17–18 July 2014, Vienna, Austria

# Vienna Summer of Logic Preface


In the summer of 2014, Vienna hosted the largest scientific conference in the history of logic. The Vienna Summer of Logic (VSL, http://vsl2014.at) consisted of twelve large conferences and 82 workshops, attracting more than 2000 researchers from all over the world. This unique event was organized by the Kurt Gödel Society at Vienna University of Technology from July 9 to 24, 2014, under the auspices of the Federal President of the Republic of Austria, Dr. Heinz Fischer.

The conferences and workshops dealt with the main theme, logic, from three important angles: logic in computer science, mathematical logic, and logic in artificial intelligence. They naturally gave rise to respective streams gathering the following meetings:

**Logic in Computer Science / Federated Logic Conference (FLoC)**

- 26th International Conference on Computer Aided Verification (CAV)
- 27th IEEE Computer Security Foundations Symposium (CSF)
- 30th International Conference on Logic Programming (ICLP)
- 7th International Joint Conference on Automated Reasoning (IJCAR)
- 5th Conference on Interactive Theorem Proving (ITP)
- Joint meeting of the 23rd EACSL Annual Conference on Computer Science Logic (CSL) and the 29th ACM/IEEE Symposium on Logic in Computer Science (LICS)
- 25th International Conference on Rewriting Techniques and Applications (RTA) joint with the 12th International Conference on Typed Lambda Calculi and Applications (TLCA)
- 17th International Conference on Theory and Applications of Satisfiability Testing (SAT)
- 76 FLoC Workshops
- FLoC Olympic Games (System Competitions)

**Mathematical Logic**

- Logic Colloquium 2014 (LC)
- Logic, Algebra and Truth Degrees 2014 (LATD)
- Compositional Meaning in Logic (GeTFun 2.0)
- The Infinity Workshop (INFINITY)
- Workshop on Logic and Games (LG)
- Kurt Gödel Fellowship Competition

**Logic in Artificial Intelligence**

- 14th International Conference on Principles of Knowledge Representation and Reasoning (KR)
- 27th International Workshop on Description Logics (DL)
- 15th International Workshop on Non-Monotonic Reasoning (NMR)
- 6th International Workshop on Knowledge Representation for Health Care 2014 (KR4HC)

The VSL keynote talks which were directed to all participants were given by Franz Baader (Technische Universität Dresden), Edmund Clarke (Carnegie Mellon University), Christos Papadimitriou (University of California, Berkeley) and Alex Wilkie (University of Manchester); Dana Scott (Carnegie Mellon University) spoke in the opening session. Since the Vienna Summer of Logic contained more than a hundred invited talks, it is infeasible to list them here.

The program of the Vienna Summer of Logic was very rich, including not only scientific talks, poster sessions and panels, but also two distinctive events. One was the award ceremony of the *Kurt Gödel Research Prize Fellowship Competition*, in which the Kurt Gödel Society awarded three research fellowship prizes endowed with 100.000 Euro each to the winners. This was the third edition of the competition, themed *Logical Mind: Connecting Foundations and Technology* this year.

The other distinctive event were the *1st FLoC Olympic Games* hosted by the Federated Logic Conference (FLoC) 2014. Intended as a new FLoC element, the Games brought together 12 established logic solver competitions by different research communities. In addition to the competitions, the Olympic Games facilitated the exchange of expertise between communities, and increased the visibility and impact of state-of-the-art solver technology. The winners in the competition categories were honored with Kurt Gödel medals at the FLoC Olympic Games award ceremonies.

Organizing an event like the Vienna Summer of Logic has been a challenge. We are indebted to numerous people whose enormous efforts were essential in making this vision become reality. With so many colleagues and friends working with us, we are unable to list them individually here. Nevertheless, as representatives of the three streams of VSL, we would like to particularly express our gratitude to all people who have helped to make this event a success: the sponsors and the honorary committee; the organization committee and the local organizers; the conference and workshop chairs and program committee members; the reviewers and authors; and of course all speakers and participants of the many conferences, workshops and competitions.

The Vienna Summer of Logic continues a great legacy of scientific thought that started in Ancient Greece and flourished in the city of Gödel, Wittgenstein and the Vienna Circle. The heroes of our intellectual past shaped the scientific world-view and changed our understanding of science. Owing to their achievements, logic has permeated a wide range of disciplines, including computer science, mathematics, artificial intelligence, philosophy, linguistics, and many more. Logic is everywhere – or in the language of Aristotle, πάντα πλήρη λογικῆς τέχνης.

Vienna, July 2014

Matthias Baaz, Thomas Eiter, Helmut Veith

## Editor's Preface

This volume contains the informal proceedings of the *14th International Workshop on Termination*, to be held on 17–18 July 2014 in Vienna, Austria.

The International Workshop on Termination (WST) 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-fertilisation of ideas from term rewriting and from the different programming language communities.

WST 2014 continues the tradition of 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), and Bertinoro (2013).

As in 2006 and 2010, also WST 2014 is part of the *Federated Logic Conference*, which in 2014 is part of an even larger event, the *Vienna Summer of Logic 2014*. In particular, WST is affiliated with the following FLoC conferences:

- 26th International Conference on Computer Aided Verification (CAV)
- 7th International Joint Conference on Automated Reasoning (IJCAR)
- 25th International Conference on Rewriting Techniques and Applications (RTA) joint with the 12th International Conference on Typed Lambda Calculi and Applications (TLCA)

The 14th Workshop on Termination features 19 regular extended abstracts, contained in this volume, and an invited talk by Jasmin Fisher on *Termination of Biological Programs*.

I would like to thank everyone who helped to prepare and run the workshop: the participants, the members of the programme committee, and the local organisers.

London, June 2014 Carsten Fuhs

## Programme Committee

| | |
|---|---|
| Elvira Albert | Complutense University of Madrid |
| Amir Ben-Amram | Tel-Aviv Academic College |
| Byron Cook | Microsoft Research and University College London |
| Carsten Fuhs (chair) | University College London |
| Jürgen Giesl | RWTH Aachen |
| Laure Gonnord | University of Lyon |
| Albert Rubio | Universitat Politècnica de Catalunya |
| Peter Schneider-Kamp | University of Southern Denmark |
| Christian Sternagel | University of Innsbruck |
| Thomas Stroeder | RWTH Aachen |
| Johannes Waldmann | HTWK Leipzig |
| Harald Zankl | University of Innsbruck |

# Table of Contents

# Type Introduction for Runtime Complexity Analysis*

**Martin Avanzini[1] and Bertram Felgenhauer[1]**

1 **Institute of Computer Science,**
**University of Innsbruck, Austria**
{martin.avanzini,bertram.felgenhauer}@uibk.ac.at

## 1 Introduction

*Runtime complexity analysis* is a natural refinement of *termination* analysis. Instead of asking whether all reductions yield a result eventually, we are interested in how long the reduction process takes. In order to measure the runtime complexity of a term rewrite system (TRS for short) it is natural to look at the maximal length of derivation sequences, a program first suggested by Hofbauer and Lautemann [5]. The resulting notion of complexity is called *derivational complexity*. Hirokawa and Moser [4] introduced a variation, called *runtime complexity*, that only takes *basic* or *constructor-based* terms as start terms into account. This notion of complexity accurately express the complexity of a program through the runtime complexity of a TRS, and constitutes an *invariant cost model* for rewrite systems [2].

Advanced techniques developed in the context of program complexity analysis essentially rely on sort information. For instance Hoffmann et al. [6] define an elegant and powerful calculus to infer various complexity properties of *resource aware ML programs*, essentially *sorted rewrite systems*, automatically. It is inherently difficult to transfer these techniques into an untyped setting.

In this note we show that the runtime complexity function of a sorted rewrite system $\mathcal{R}$ coincides with the runtime complexity function of the unsorted rewrite system $\Theta(\mathcal{R})$, obtained by forgetting sort information. Hence our result states that *sort-introduction*, a process that is easily carried out via unification, is sound for runtime complexity analysis. Our result thus provides the foundation for exploiting sort information in analysis of TRSs.

Our main research is tightly related to the research on *persistent* properties [9] of rewrite systems, e.g. [1, 7]. Here a property on rewrite systems is called persistent if it holds for the sorted TRS $\mathcal{R}$ if and only if it holds on the unsorted variant $\Theta(\mathcal{R})$. As trivial corollary to our main result we obtain that *innermost termination* is persistent, a result that has been previously established in [3].

## 2 Preliminaries

We assume familiarity with rewriting [8]. We denote by $\mathcal{V}$ a countable infinite set of *variables*, $\mathcal{F}$ denotes a signature and $\mathcal{T}(\mathcal{F},\mathcal{V})$ denotes the set of terms with symbols in $\mathcal{F}$ and variables in $\mathcal{V}$. We denote by $\mathsf{Var}(t)$ the set of variables occurring in $t$. Let $\mathcal{R}$ be a TRS. Roots of left-hand sides in $\mathcal{R}$ are called *defined*, symbols that are not defined are called constructors and are collected in $\mathcal{C}_\mathcal{R}$. Terms $t = f(t_1,\dots,t_k)$ with $t_i \in \mathcal{T}(\mathcal{C}_\mathcal{R},\mathcal{V})$ for all $i = 1,\dots,k$ are called *basic*. The *rewrite relation* of a term rewrite system $\mathcal{R}$ is denoted by $\to_\mathcal{R}$, by $\xrightarrow{\mathsf{i}}_\mathcal{R}$ we denote the innermost rewrite relation of $\mathcal{R}$. The *runtime complexity (function)* $\mathrm{rc}_\mathcal{R} : \mathbb{N} \to \mathbb{N}$

* This work is supported by FWF (Austrian Science Fund) projects J3563 and P22467.

of $\mathcal{R}$ is defined by

$$\mathrm{rc}_{\mathcal{R}}(n) := \max\{\ell \mid \exists t_0, \dots, t_\ell.\ t_0 \to_{\mathcal{R}} \cdots \to_{\mathcal{R}} t_\ell \text{ and } t_0 \text{ is a basic term of size up to } n\}\ .$$

Note that $\mathrm{rc}_{\mathcal{R}}$ is well-defined when $\mathcal{R}$ is terminating. The *innermost runtime complexity (function)* $\mathrm{rci}_{\mathcal{R}}$ of $\mathcal{R}$ is defined analogously, considering innermost reductions only.

To simplify notations, we employ the notion of *$\mathcal{S}$-sorted rewriting* of Aoto and Toyama [1]. Let $\mathcal{S}$ be a set of *sorts*. Sorts are denoted by $\alpha, \beta, \dots$, possibly followed by subscripts. A *sort-attachment* is a mapping $\tau$ from $\mathcal{V} \cup \mathcal{F}$ to $\mathcal{S}^*$ such that $\tau(x) \in \mathcal{S}$ for $x \in \mathcal{V}$ and $\tau(x) \in \mathcal{S}^{k+1}$ for every $k$-ary $f \in \mathcal{F}$. In the latter case we write $f : \alpha_1, \dots, \alpha_k \to \alpha$ instead of $\tau(f) = \alpha_1, \dots, \alpha_k, \alpha$. Without loss of generality, we assume that for each $\alpha \in \mathcal{S}$ the sets $\mathcal{V}_\alpha := \{x \mid \tau(x) = \alpha\}$ are countable infinite. The sort $\mathsf{sort}(t)$ of a term $t$ is defined by the root symbol only. We set $\mathsf{sort}(x) := \tau(x)$ for variables $x$ and $\mathsf{sort}(f(t_1, \dots, t_k)) = \alpha$ where $f : \alpha_1, \dots, \alpha_k \to \alpha$.

A term $t$ is *well-sorted* (under $\tau$) with sort $\alpha$ if $t : \alpha$ is derivable by the following rules: (i) $t = x$ and $\tau(x) = \alpha$, or (ii) $t = f(t_1, \dots, t_k)$, $f : \alpha_1, \dots, \alpha_k \to \alpha$ and $t_i : \alpha_i$ $(i = 1, \dots, k)$. We denote by $\mathcal{T}(\mathcal{F}, \mathcal{V})^\tau \subseteq \mathcal{T}(\mathcal{F}, \mathcal{V})$ the set of all term which are well-sorted under $\tau$.

An *$\mathcal{S}$-sorted TRS* $\mathcal{R}$ is given by an unsorted TRS $\Theta(\mathcal{R})$ and sort-attachment $\tau$ such that every rule $l \to r \in \Theta(\mathcal{R})$, $l : \alpha$ and $r : \alpha$ holds for some sort $\alpha \in \mathcal{S}$. In the following, $\mathcal{R}$ always denotes an $\mathcal{S}$-sorted TRS. The rewrite relation $\to_{\mathcal{R}}$ of an $\mathcal{S}$-sorted TRS is given by the restriction of $\to_{\Theta(\mathcal{R})}$ to well-sorted terms $\mathcal{T}(\mathcal{F}, \mathcal{V})^\tau$. We extend the notion of runtime complexity function in the obvious way to $\mathcal{S}$-sorted TRSs. A property $P$ of TRSs is called *persistent* if for each rewrite system $\mathcal{R}$, $\mathcal{R}$ has property $P$ if and only if $\Theta(\mathcal{R})$ has property $P$. Notice that our notion of persistency coincides with the standard notion formulated on many-sorted TRSs, see [1].

## 3 Bounded Runtime Complexity is a Persistent Property of TRSs

In the following we show that the *bounded runtime complexity problem*, which asks for a TRS $\mathcal{R}$ and function $f : \mathbb{N} \to \mathbb{N}$ whether $\mathrm{rc}_{\mathcal{R}}(n) \leqslant f(n)$ for all $n \in \mathbb{N}$ holds, is persistent. We even show a stronger property, viz, $\mathrm{rc}_{\mathcal{R}}(n) = \mathrm{rc}_{\Theta(\mathcal{R})}(n)$ for all $n \in \mathbb{N}$. It is clear that every $\mathcal{R}$-derivation is also an $\Theta(\mathcal{R})$-derivation, hence $\mathrm{rc}_{\mathcal{R}}(n) \leqslant \mathrm{rc}_{\Theta(\mathcal{R})}(n)$ holds trivially. The converse is however not true in general. Consider the sorted TRS $\mathcal{R}_1$ consisting of rules

$$\mathsf{f}(0, 1, x) \to \mathsf{f}(x, x, x) \qquad \mathsf{g}(y, z) \to y \qquad \mathsf{g}(y, z) \to z\ ,$$

and sort-attachment so that $0, 1 : \alpha$, $\mathsf{f} : \alpha, \alpha, \alpha \to \alpha$ and $\mathsf{g} : \beta, \beta \to \beta$. Notice that $\mathcal{R}_1$ is terminating, since sorting excludes the formation of terms involving both $\mathsf{f}$ and $\mathsf{g}$ symbols. On the other hand, the TRS $\Theta(\mathcal{R}_1)$ gives rise to a cycle

$$t := \underline{\mathsf{f}(0, 1, \mathsf{g}(0, 1))} \to_{\Theta(\mathcal{R}_1)} \mathsf{f}(\underline{\mathsf{g}(0, 1)}, \mathsf{g}(0, 1), \mathsf{g}(0, 1)) \to_{\Theta(\mathcal{R}_1)} \mathsf{f}(0, \underline{\mathsf{g}(0, 1)}, \mathsf{g}(0, 1)) \to_{\Theta(\mathcal{R}_1)} t\ ,$$

and is thus not terminating. The TRS $\mathcal{R}_1$ is the prototypical example that shows that termination is not persistent, it is however not a counterexample to our claim. The notion of runtime complexity considers only basic, i.e. *argument normalised* terms. Indeed, the runtime complexity function of the sorted TRS $\mathcal{R}_1$ and its unsorted version $\Theta(\mathcal{R}_1)$ coincide.

Our central observation is that in a $\Theta(\mathcal{R})$-derivation $D$ starting from argument normalised term $t$, subterms that lead to a sort conflict (called *aliens* of $t$ below) do not contribute to the derivation $D$ itself. Although the (normalised) aliens might get duplicated or erased, the sorting condition on $\mathcal{R}$ ensures that aliens never contribute to a pattern which triggers

the application of a rule. This suggests that such aliens in $t$ can be replaced by fresh variables so that the resulting term $s$ is well-sorted. Although some care has to be taken in the assignment of variables to aliens for non-left-linear systems, the derivation $D$ of $t$ can be simulated step-wise by a $\mathcal{R}$-derivation starting from the modified term $s$.

Fix a set of sorts $\mathcal{S}$ and an $\mathcal{S}$-sorted TRS $\mathcal{R}$. To define sorted contexts, we assume the presence of fresh constants $\Box_\alpha$, the *holes*, for each sort $\alpha \in \mathcal{S}$. We extend the type assignment $\tau$ underlying $\mathcal{R}$ so that $\tau(\Box_\alpha) = \alpha$. A *multi-holed context* $C[\Box_{\alpha_1}, \dots, \Box_{\alpha_n}]$ is a sorted term that contains each hole $\Box_{\alpha_i}$ $(i = 1, \dots, n)$ exactly once. With $C[t_1, \dots, t_n]$ we denote the term obtained by replacing holes $\alpha_i$ with $t_i$ in $C[\Box_{\alpha_1}, \dots, \Box_{\alpha_n}]$.

▶ **Definition 3.1.** We write $s = C[\![s_1, \dots, s_n]\!]$ for the *unique* decomposition $s = C[s_1, \dots, s_n]$ into a well-sorted context $C[\Box_{\alpha_1}, \dots, \Box_{\alpha_n}]$ and terms $s_i$ with and $\mathsf{sort}(s_i) \neq \alpha_i$ for every $i = 1, \dots, n$. The subterms $s_1, \dots, s_n$ of $s$ are called the *aliens* of $s$. The set of all aliens $\{s_1, \dots, s_n\}$ in $s$ is denoted by $\mathsf{alien}(s)$.

Note that when $s$ is well-sorted, the context $C$ degenerates to $s$.

▶ **Definition 3.2.** Let $s$ be a term. Consider a family $\gamma = (\gamma_\alpha : T_{\neq\alpha} \to V_\alpha)_{\alpha\in\mathcal{S}}$ of *bijective* mappings from terms $T_{\neq\alpha} \subseteq \{t \in \mathcal{T}(\mathcal{F}, \mathcal{V}) \mid \mathsf{sort}(t) \neq \alpha\}$ to variables $V_\alpha \subseteq \mathcal{V}_\alpha$. We define the *domain* and *range* of $\gamma$ by $\mathsf{dom}(\gamma) := \cup_{\alpha\in\mathcal{S}} T_{\neq\alpha}$ and $\mathsf{range}(\gamma) := \cup_{\alpha\in\mathcal{S}} V_\alpha$ respectively. Then $\gamma$ is called an *alien replacement* for $s$ if $\mathsf{alien}(s) \subseteq \mathsf{dom}(\gamma)$ and $\mathsf{range}(\gamma) \cap \mathsf{Var}(s) = \varnothing$. We denote by $\bar\gamma$ the *inverse* of $\gamma$: $\bar\gamma(x) := t$ where $\gamma_{\mathsf{sort}(x)}(t) = x$ for all $x \in \mathsf{range}(\gamma)$.

We define $s \gtrdot_\gamma t$ if $s = C[\![s_1, \dots, s_n]\!]$ for context $C[\Box_{\alpha_1}, \dots, \Box_{\alpha_n}]$, $\gamma$ is an alien replacement for $s$ and $t = C[\gamma_{\alpha_1}(s_1), \dots, \gamma_{\alpha_n}(s_n)]$ is well-sorted.

Notice that to each term $s = C[\![s_1, \dots, s_n]\!]$ we can associate an alien replacement $\gamma$ and term $t$ such that $s \gtrdot_\gamma t$ holds: Start from a well-sorted term $C[x_1, \dots, x_n]$ for pairwise disjoint and fresh variables of appropriate sort. Identify variables $x_i$ and $x_j$ $(i, j \in \{1, \dots, n\})$ when $\mathsf{sort}(x_i) = \mathsf{sort}(x_j)$ and $s_i = s_j$. The fixpoint of this construction yields the well-sorted term $t := C[y_1, \dots, y_n]$. The family $\gamma = (\gamma_\alpha)_{\alpha\in\mathcal{S}}$, defined by $\gamma_{\mathsf{sort}(x_i)}(y_i) := s_i$ for $i = 1, \dots, n$, is an alien replacement where by construction $s \gtrdot_\gamma t$ holds.

Consider $s \gtrdot_\gamma t$. By the conditions on $\mathsf{range}(\gamma)$ it follows that $t$ matches $s$ with substitution $\bar\gamma$, i.e. $s = t\bar\gamma$. Provided $\gamma$ is an alien replacement, we can also state the inverse correspondence.

▶ **Lemma 3.3.** *Let $\gamma$ denote an alien replacement for a term $s$. Then $s \gtrdot_\gamma t$ if and only if $s = t\bar\gamma$ and $t$ is well-sorted.*

The following lemma confirms that $t$ is a *maximal* well-sorted pattern that matches $s$.

▶ **Lemma 3.4.** *Suppose $s \gtrdot_\gamma t$ holds, and let $u$ be a well-sorted term with $\mathsf{Var}(u) \cap \mathsf{range}(\gamma) = \varnothing$. If $u$ matches $s$ then $u$ matches also $t$.*

**Proof.** Let $\sigma$ be a substitution with $s = u\sigma$. Without loss of generality, we suppose $\mathsf{dom}(\sigma) \subseteq \mathsf{Var}(u)$. Observe that since $u$ is well-sorted, aliens of $s$ occur only in the substitution part. We define the substitution $\sigma_\gamma$ as follows, for all $x \in \mathsf{dom}(\sigma)$: suppose $\mathsf{sort}(x) \neq \mathsf{sort}(\sigma(x))$, thus $\sigma(x) \in \mathsf{alien}(s)$ and $\gamma_{\mathsf{sort}(x)}(s)$ is well-defined. Then we set $\sigma_\gamma(x) := \gamma_{\mathsf{sort}(x)}(\sigma(x))$. Otherwise, suppose $\sigma(x) = C_x[\![s_1, \dots, s_m]\!]$ for some non-empty context $C_x[\Box_{\alpha_1}, \dots, \Box_{\alpha_m}]$ and aliens $\{s_1, \dots, s_m\} \subseteq \mathsf{alien}(s)$. Then we set $\sigma_\gamma(x) := C_x[\gamma_{\alpha_1}(s_1), \dots, \gamma_{\alpha_1}(s_m)]$.

By definition of $\sigma_\gamma$, $\sigma_\gamma(x)\bar\gamma = \sigma(x)$ for $x \in \mathsf{dom}(\sigma)$. By the variable condition on $u$, we have $(u\sigma_\gamma)\bar\gamma = s$. Note that $u\sigma_\gamma$ is by construction well-sorted, Lemma 3.3 thus gives $s \gtrdot_\gamma u\sigma_\gamma$. By definition of $\gtrdot_\gamma$ we see that $s \gtrdot_\gamma u\sigma_\gamma$ and $s \gtrdot_\gamma t$ implies $u\sigma_\gamma = t$. ◀

The following lemma provides our central simulation result. Since we consider derivations from argument normalised terms only, it suffices to consider only *outer steps* in the simulation.

▶ **Definition 3.5.** A rewrite step $s \to_{\Theta(\mathcal{R})} t$ is called *inner* if it takes place in one of the aliens of $s$. The step $s \to_{\Theta(\mathcal{R})} t$ is called *outer* if it is not an inner rewrite step.

▶ **Lemma 3.6.** *Suppose $s_1 \succcurlyeq_\gamma t_1$ holds for an alien replacement $\gamma$ with $\mathsf{range}(\gamma)$ disjoint from the set of variables occurring in $\mathcal{R}$.*

1) *If $s_1 \to_{\Theta(\mathcal{R})} s_2$ is an outer step then $t_1 \to_{\mathcal{R}} t_2$ for some term $t_2$ with either (i) $s_2 \succcurlyeq_\gamma t_2$ or (ii) $s_2 \in \mathsf{alien}(s_1)$ with $t_2 = \gamma_\alpha(s_2)$ for some $\alpha \in \mathcal{S}$.*
2) *If $s_1 \xrightarrow{\mathsf{i}}_{\Theta(\mathcal{R})} s_2$ is an outer step then $t_1 \xrightarrow{\mathsf{i}}_{\mathcal{R}} t_2$ for some term $t_2$ with either (i) $s_2 \succcurlyeq_\gamma t_2$ or (ii) $s_2 \in \mathsf{alien}(s_1)$ and $t_2 = \gamma_\alpha(s_2)$ for some $\alpha \in \mathcal{S}$.*

**Proof.** We consider Proposition 1 first. Suppose $s_1 \succcurlyeq_\gamma t_1$ for $\gamma$ as above. Consider an outer rewrite step $s_1 \to_{\Theta(\mathcal{R})} s_2$. The proof is by induction on the rewrite context.

In the base case, $s_1 = l\sigma$ and $s_2 = r\sigma$ for some substitution $\sigma$ and rewrite rule $l \to r \in \mathcal{R}$. By Lemma 3.4 we obtain a substitution $\sigma_\gamma$ such that $t_1 = l\sigma_\gamma \to_{\mathcal{R}} r\sigma_\gamma$. We verify that either condition (i) or (ii) holds for $t_2 = r\sigma_\gamma$.

Reconsider the substitution $\sigma_\gamma$ constructed in Lemma 3.4. Suppose first that the applied rewrite rule is collapsing, i.e. $r \in \mathsf{Var}(l)$. We distinguish the two cases in construction of $\sigma_\gamma$. In the first case, $r\sigma_\gamma = \gamma_{\mathsf{sort}(r)}(r\sigma)$ with $r\sigma \in \mathsf{alien}(s_1)$, i.e. (ii) holds. In the second case $r\sigma_\gamma = C_x[\![u_1, \dots, u_m]\!]$ for some non-empty context $C_x[\Box_{\alpha_1}, \dots, \Box_{\alpha_m}]$ and aliens $\{u_1, \dots, u_m\} \subseteq \mathsf{alien}(s_1)$. This yields $\mathsf{alien}(r\sigma) \subseteq \mathsf{alien}(l\sigma)$, as moreover $\mathsf{Var}(r\sigma) \subseteq \mathsf{Var}(l\sigma)$ we conclude that $\gamma$ is also an alien replacement for $r\sigma$. As the side conditions on $\mathsf{range}(\gamma)$ gives $(r\sigma_\gamma)\bar{\gamma} = r\sigma$, we conclude $r\sigma \succcurlyeq_\gamma r\sigma_\gamma$ by Lemma 3.3.

Now suppose that the applied rewrite rule is non-collapsing. Since $l \to r$ is well-sorted, any alien in $r\sigma$ occurs in the substitution, and hence is an alien of $l\sigma$. Hence again $\gamma$ is an alien replacement for $r\sigma$, since $(r\sigma_\gamma)\bar{\gamma} = r\sigma$ we obtain $r\sigma \succcurlyeq_\gamma r\sigma_\gamma$ using Lemma 3.3. This finishes the base case.

For the inductive step, consider an outer rewrite step

$$s_1 = f(u_1, \dots, u_i, \dots, u_k) \to_{\Theta(\mathcal{R})} f(u_1, \dots, v_i, \dots, u_k) = s_2 \ ,$$

with $u_i \to_{\Theta(\mathcal{R})} v_i$ outer. Using $s_1 \succcurlyeq_\gamma t_1$, Lemma 3.3 gives $t_1 = f(u'_1, \dots, u'_i, \dots, u'_k)$ with $u_i = u'_i\bar{\gamma}$ for all $i = 1, \dots, k$. Hence by induction hypothesis, $u'_i \to_{\mathcal{R}} v'_i$ for some well-sorted term $v'_i$ with either $v_i \succcurlyeq_\gamma v'_i$ or $v'_i = \gamma_\alpha(v_i)$ for $v_i \in \mathsf{alien}(u_i)$ and $\alpha \in \mathcal{S}$. Set $t_2 := f(u'_1, \dots, v'_i, \dots, u'_k)$ and thus $t_1 \to_{\mathcal{R}} t_2$. If $v_i \succcurlyeq_\gamma v'_i$ holds then $s_2 \succcurlyeq_\gamma t_2$ follows by applying Lemma 3.3 immediately. Hence suppose $v'_i = \gamma_\alpha(v_i)$. Since $t_2$ is well-sorted and $\mathsf{sort}(v_i) \neq \mathsf{sort}(\gamma_\alpha(v_i))$ by definition, it follows that $v_i$ is an alien in $s_2$. Again we conclude $s_2 = t_2\bar{\gamma}$ and thus $s_2 \succcurlyeq_\gamma t_2$ by Lemma 3.3. We conclude Proposition 1.

For Proposition 2, observe that if $l\sigma \to_{\Theta(\mathcal{R})} r\sigma$ is an innermost step, i.e $l\sigma$ is argument normalised, then so is $l\sigma_\gamma$ and hence $l\sigma_\gamma \to_{\mathcal{R}} r\sigma_\gamma$ is an innermost rewrite step. Proposition 2 follows then by reasoning identical to above. ◀

▶ **Lemma 3.7.** *If $s_1 \to_{\Theta(\mathcal{R})} s_2$ is outer, then either $\mathsf{alien}(s_2) \subseteq \mathsf{alien}(s_1)$ or $s_2 \in \mathsf{alien}(s_1)$.*

**Proof.** Consider an outer step $s_1 \to_{\Theta(\mathcal{R})} s_2$, and let $t_1$ be such that $s_1 \succcurlyeq_\gamma t_1$ holds. Then by Lemma 3.6, either $s_2 \succcurlyeq_\gamma t_2$ for some term $t_2$ or $s_2 \in \mathsf{alien}(s_1)$. In the former case, $s_2 \succcurlyeq_\gamma t_2$ witnesses that aliens of $s_2$ occur as aliens in $s_1$, in the latter case we conclude directly. ◀

▶ **Theorem 3.8.** *Let $s$ be a term such that all aliens in $s$ are in $\Theta(\mathcal{R})$ normal-form. Then any (innermost) $\Theta(\mathcal{R})$-derivation of $s$ is simulated step-wise by an (innermost) $\mathcal{R}$-derivation starting from some $t$ with $s \succcurlyeq_\gamma t$.*

**Proof.** Consider a derivation $D : s = s_0 \to_{\Theta(\mathcal{R})} s_1 \to_{\Theta(\mathcal{R})} s_2 \to_{\Theta(\mathcal{R})} \cdots$. Using Lemma 3.7, a standard induction shows that $\mathsf{alien}(s_i) \subseteq \mathsf{alien}(s)$ for all but possibly the last term in $D$, and that the steps $s_i \to_{\Theta(\mathcal{R})} s_{i+1}$ are outer. We conclude the by Lemma 3.6(1) (Lemma 3.6(2) respectively). ◀

Any basic term $s$ satisfies trivially that aliens of $s$ are in $\Theta(\mathcal{R})$ normal-form. The above theorem thus shows that the $\mathsf{dh}(s, \to_{\Theta(\mathcal{R})}) \leqslant \mathsf{dh}(s', \to_{\mathcal{R}})$, whenever $s$ is basic. Since $s \succcurlyeq_\gamma s'$ implies that $|s| \geqslant |s'|$ it follows that $\mathrm{rc}_{\Theta(\mathcal{R})}(n) \leqslant \mathrm{rc}_{\mathcal{R}}(n)$. By identical reasoning, $\mathrm{rci}_{\Theta(\mathcal{R})}(n) \leqslant \mathrm{rci}_{\mathcal{R}}(n)$. Thus we obtain the following corollary.

▶ **Corollary 3.9.** *The (innermost) runtime complexity functions of $\mathcal{R}$ and $\Theta(\mathcal{R})$ coincide. In particular, the bounded (innermost) runtime complexity problem is persistent.*

Observe that if a TRS $\mathcal{R}$ is innermost non-terminating, then there exists a minimal non-terminating term $s = f(s_1, \ldots, s_n)$ in the sense that all arguments are in normal-form. In particular, the aliens of $s$ are normalised. We thus re-obtain the following result from [3].

▶ **Corollary 3.10.** *Innermost termination is a persistent property.*

## 4 Conclusion

In this abstract we have shown that sort-introduction is sound for runtime complexity analysis. We considered the most simple form of sorted rewriting. It is expected that our result can be extended to more general forms, allowing for instance *polymorphism* or *ordered sorts*. Such extensions are subject to future research. To which extent sort information can be exploited in runtime complexity analysis is also subject to further research.

## References


1 T. Aoto and Y. Toyama. Persistency of Confluence. *JUCS*, 3(11):1134–1147, 1997.
2 M. Avanzini and G. Moser. Closing the Gap Between Runtime Complexity and Polytime Computability. In *Proc. of 21st RTA*, volume 6 of *LIPIcs*, pages 33–48. Dagstuhl, 2010.
3 C. Fuhs, J. Giesl, M. Parting, P. Schneider-Kamp, and S. Swiderski. Proving Termination by Dependency Pairs and Inductive Theorem Proving. *JAR*, 47(2):133–160, 2011.
4 N. Hirokawa and G. Moser. Automated Complexity Analysis Based on the Dependency Pair Method. In *Proc. of 4th IJCAR*, volume 5195 of *LNAI*, pages 364–380. Springer, 2008.
5 D. Hofbauer and C. Lautemann. Termination Proofs and the Length of Derivations. In *Proc. of 3rd RTA*, volume 355 of *LNCS*, pages 167–177. Springer, 1989.
6 J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *Proc. of 38th POPL*, pages 357–370. ACM, 2011.
7 A. Middeldorp and H. Ohsaki. Type Introduction for Equational Rewriting. *AI*, 36(12): 1007–1029, 2000.
8 TeReSe. *Term Rewriting Systems*, volume 55 of *CTTCS*. Cambridge University Press, 2003.
9 H. Zantema. Termination of Term Rewriting: Interpretation and Type Elimination. *JSC*, 17(1):23–50, 1994.

# Automated SAT Encoding for Termination Proofs with Semantic Labelling*

**Alexander Bau, René Thiemann, and Johannes Waldmann**

**HTWK Leipzig, Fakultät IMN, 04277 Leipzig, Germany**
{abau|waldmann}@imn.htwk-leipzig.de

**University of Innsbruck, Austria,** rene.thiemann@uibk.ac.at

**Abstract**

We discuss design choices for SAT-encoding constraints for termination orders based on semantic labelling and unlabelling, linear interpretations, recursive path orders with argument filters, within the dependency pairs framework.

We specify constraints in a high-level Haskell-like language, and translate to SAT fully automatically by the CO4 compiler. That way, constraints can be combined easily.

This allows to write a single constraint for *find a model, and a sequence of ordering constraints for the labelled system, such that at least one original rule can be removed completely.* Reliability is achieved via certification of generated proofs.

The size of the resulting propositional logic formulas can be reduced by strengthening the constraints. We discuss an encoding of finite maps via patterns.

## 1 Introduction

Termination of a (rewrite) relation $\to$ can be shown by embedding $\to$ in a well-founded order $>$. Such an order can be given syntactically, by comparing the shape of terms, and occurences of symbols, as it happens in recursive path orders. Another option is to define the order semantically, where each term is assigned an element of a well-founded algebra, by giving interpretations of symbols. For the domain of such an algebra, we can use, e.g., numbers, vectors, matrices. Then there are methods for constructing a well-founded order by combination of others, for example, using the lexicographic product. We also have methods that transform a termination problem into another. Semantic labelling uses a finite algebra that is a model for the rewrite relation, to assign labels to function symbols, thereby increasing the signature, allowing a more fine-grained analysis of the transformed system. The dependency pairs transformation also transforms a rewrite system, so that sequences of "function calls" can be analyzed.

When we write a program to prove termination automatically, we prescribe a certain termination proof method, and the task of the program (the "prover") is to fill in all the parameters. E.g., if we want to use interpretations by linear functions, then suitable coefficients of such functions must be determined. If we want to use a recursive path order, then a suitable precedence of symbols is needed. In each case, suitability can be described by a formula in predicate logic, which we call a "termination (ordering) constraint". Thus, the termination prover is actually a "solver" of constraints.

*The present paper is not about new methods of proving termination, but about the pragmatics of writing down termination constraints.* As with all source code, one goal is efficiency of execution, and another goal is efficiency of expression, leading to readability and maintainability. We advocate the use of our high-level declarative constraint programming language

* Supported by ESF grant 100088525 and FWF project P22767.

CO4, that comes with a compiler that targets propositional satisfiabiliy (SAT).

SAT encoding is a successful method of solving finite domain constraints. Some ordering constraints have finite domain by definition (a finite model, a precedence for a finite signature), others do not, e.g., because they involve numbers. SAT encoding can still be used, by restricting to some finite subset, e.g., numbers of certain bit width.

The current state of SAT encoding (for termination) can be described as: it is successful (several successful termination provers use it) but it is also laborious. This is mainly due to explicit manipulation of propositional variables, corresponding to manual assignments from identifiers to memory locations in low-level (assembly) programs.

This increases the work in encoding combined constraints. E.g., argument filtering and path order comparison are handled at the same time, while they are conceptually independent. A filter $\pi$ denotes a mapping $F^\pi$ from terms to terms (that removes some nodes and subterms). Then, mapped terms are compared w.r.t. a *p*ath *o*rder: $F^\pi(s) >_{PO} F^\pi(t)$. In the source code of our termination prover, this is literally expressed as the Haskell expression

```
case order of
  FilterAndPrec f p ->
    lpo p (filterArgumentsDPTerm f lhs) (filterArgumentsDPTerm f rhs)
```

In contrast, the encoding described in [5] combines these steps, so that the encoding of $F^\pi$ is "fused" into the encoding of the path order, realizing a relation $>^\pi_{PO}$ on terms.

We implement the following constraint for termination proofs that use semantic labelling [11], path orders and linear interpretations in the context of the dependency pairs framework [1]. The known input is a DP problem (set of (dependency) pairs and set of rewrite rules). The unknown is a pair of an interpretation into a finite domain, and a termination order (on the labelled signature), such that the interpretation is a model for the rewrite rules; all labelled pairs and rules are weakly compatible with the order, and for at least one pair, all its labelled versions are strictly compatible with the order.

The order is a lexicographic combination of basic orders, where a basic order is either a recursive path order with argument filter, or built from a linear interpretation.

That means our constraint can describe a proof step where we first apply semantic labelling, then remove a number of labelled pairs by using several orders in succession (e.g., first, a linear interpretation, then a path order, or path orders with different argument filters), where at each step, we recompute usable rules, and finally unlabel. Proofs are certified by CeTA [9].

The source code of our termination prover (and the SAT compiler CO4 [4]) is available at `https://github.com/apunktbau/co4`

The present paper builds on [3]. New contributions are: extension from string rewriting to term rewriting, applying the dependency pairs framework, and certifiable proofs.

## 2 Structured Finite Domain Constraints and their SAT Encoding

We briefly review the concepts of constraint programming with CO4. A parametric constraint is given as a function `c :: P -> U -> Bool`, written in a subset of Haskell, where `P` is a domain of parameters, and `U` is the domain of unknowns. The constraint `c` is compiled to a function `cc :: P -> CNF` such that `cc p` gives a propositional logic formula `f` in conjunctive normal form such that from a satisfying assignment of `f`, an object `u :: U` can be reconstructed with `c p u == True`.

CO4 handles algebraic data types (`data`), and case distinctions by pattern matching. A set of unknown objects of an algebraic data type is represented as a tree, where each node

contains propositional variables. Each assignment determines a (binary) number, which in turn determines the constructor in this node. Pattern match on the constructor is realized by compiling all branches, and adding selector functions. When merging results from different branches of a case distinction, the corresponding trees are overlapped. This allows to handle finite domain constraints in Haskell notation. For infinite types (lists, trees), we can specifiy finite subsets by restricted recursion. While the core language of CO4 is first-order, we allow higher order functions and remove them by specialization.

In our application, the main constraint is `c :: DPProblem -> Proof -> Bool`, where `c d p == True` if `p` proves that at least one pair can be removed from the DP problem `d`. We use these types:

```
data Proof = Proof (Model Symbol) [ UsableOrder (Symbol,Label) ]
type UsableOrder key = (UsableSymbol key, TerminationOrder key)
type UsableSymbol key = Map key Bool
```

In particular, a `Proof` object consists of a model $m$, and a list $[(u_1, o_1), \ldots, (u_k, o_k)]$. Here, $o_i$ is an order, and $u_i$ describes an over-approximation of the symbols that are usable w.r.t. the dependency pairs that remain after removing those that are decreasing w.r.t. the lexicographic product of $o_1$ to $o_{i-1}$. For orders, we use

```
data TerminationOrder key = FilterAndPrec (ArgFilter key) (Precedence key)
                          | LinearInt (LinearInterpretation key)
data Index = This | Next Index
data Filter = Selection [ Index ] | Projection Index
type ArgFilter key = Map key Filter
data Precedence key = EmptyPrecedence | Precedence (Map key Nat)
```

## 3 Encoding of Finite Maps with Patterns

We discuss in more detail the cost of SAT-encoding of maps (lookup tables). These are used for finite algebras (as models of rewrite systems). The basic functionality is

```
lookup :: Map k v -> k -> Maybe v
```

Let us estimate the cost of the encoding. Consider the (common) case that the key to be looked up is unknown (i.e., encoded). Then the naive algorithm is to compare the key with each key in the map. This requires a linear (in the size of the map) number of (encoded) key comparisons. This is exactly what CO4 generates from the following program.

```
type Map k v = [(k,v)]
lookup x m = case m of
    [] -> Nothing
    (k,v): m' -> if x == k then Just v else lookup x m'
```

We might think of a balanced tree instead of a list, as in

```
data Map k v = Leaf | Branch k v (Map k v) (Map k v)
lookup x m = case m of
    Leaf -> Nothing
    Branch k v l r -> if x == k then Just v else
        if x < k then lookup x l else lookup x r
```

Note that the result of `x < k` is unknown when generating the formula, since `x` is encoded. This means that both branches of `if` have to be encoded. Again, this is what CO4 does

when translating the program. This results, again, in linear formula size. We conclude that using balanced trees for lookups with encoded keys is not helpful.

We can still achieve smaller formulas by giving up on completeness: We encode only a subset of all maps. We restrict to encoding maps where keys are lists of domain elements (that is, `Map [d] v`) as it happens in semantic labelling.

We represent such a map by a list of patterns that are matched from left to right. If we are lucky, the model is representable in the subset.

```
data Match d = Any | Exactly d
type MapList d v = [ ( [Match d] , v) ] --  represents Map [d] v
```

For instance, the pattern `[(([Any,Exactly 0],0), ([Any,Any],1)]` represents the function `[([0,0],0), ([0,1],1), ([1,0],0), ([1,1],1)]`.

As discussed earlier, we do have linear cost for the lookup of an encoded key. This cost increases for pattern matching, but the plan is to reduce it by making the number of patterns (much) smaller than the domain of the function.

In the extreme case, we use just one pattern `[Any,Any,... Any]`. This will give a model where all symbols are interpreted by constant functions.

## 4 Certification

The approach of iteratively applying "1) labelling, 2) applying several orders with usable rules, 3) unlabelling" is not easy to certify, since if one would allow arbitrary sound termination techniques in 2), the whole approach would be unsound: the problem is that unlabelling on its own is unsound, cf. [9, Example 4.3]. To solve this problem, [9] utilizes a dedicated semantics for DP problems w.r.t. semantic labelling.

But since this semantics was hard to extend, CeTA is now based on a more general semantics of DP problems which borrows ideas from relative rewriting [10], where there are relative DP problems with strict and weak pairs and rules. Then all unlabelling steps (UL) can be eliminated as follows: CeTA automatically transforms every proof of the form $(\mathcal{P}_0, \mathcal{R}_0) \overset{\text{SL}}{\hookrightarrow} (\mathcal{P}_1, \mathcal{R}_1) \overset{\succ_1}{\hookrightarrow} \cdots \overset{\succ_{n-1}}{\hookrightarrow} (\mathcal{P}_n, \mathcal{R}_n) \overset{\text{UL}}{\hookrightarrow} (\mathcal{P}', \mathcal{R}')$ into the following proof: first, a *split* processor is applied on $(\mathcal{P}_0, \mathcal{R}_0)$ which returns two new DP problems: the resulting DP problem $(\mathcal{P}', \mathcal{R}')$ and a relative DP problem $(\mathcal{P}_0 - \mathcal{P}', \mathcal{P}', \mathcal{R}_0 - \mathcal{R}', \mathcal{R}')$ where the first and third components are strict rules which have to be deleted. Then semantic labelling (SL) is applied on this relative DP problem and afterwards all orders $\succ_1, \ldots, \succ_{n-1}$ are used to finally get the DP problem $(\varnothing, \mathcal{P}_n, \varnothing, \mathcal{R}_n)$. Termination of this relative DP problem is then trivially proven as it does neither contain strict pairs nor strict rules.

All generated proofs have been certified via this approach, though initially problems occurred: the termination tool had bugs in its CPF-export; and CeTA rejected some valid proofs due to a buggy implementation of the transformation to eliminate unlabelling steps.

## 5 Related Work and Discussion

Semantic labelling had been used in "early" (2006) termination competitions [8] in termination provers Torpa (Zantema), Jambox (Endrullis), in TPA (Koprowski), Teparla (van der Wulp). Here, Jambox used SAT encoding, TPA used predictive labelling [7], and Teparla used recursive labelling with Boolean models.

Except for TPA, we assume that these early implementations used some kind of generate-and-test approach, where a model is found in one step, and in an independent step, a

termination proof is attempted for the labelled system. This means that often, several models need to be tried, and trivial models are to be excluded somehow.

Currently, semantic labelling with finite models is implemented in AProVE (Giesl et. al) which still uses a generate-and-test approach, but benefits from using several termination techniques between labelling and unlabelling. Also the complexity tool TcT (Avanzini et. al) uses semantic labelling, and both TcT and TPA are similar to our approach: they perform a combined SAT search for suitable models and orderings [2, 7]. However, both TcT and TPA use a manual encoding to SAT and do not support certifiable output for labelling.

Our current implementation uses "cheap" termination methods first (SCC decomposition, arctic matrices with small bit width and small dimension), and semantic labelling, as described here, only as a "last resort". We remark that proof search is nicely controlled in the `LogicT IO` monad [6]. We are currently evaluating experimentally the influence of different choices and shortcuts in the formulation of the termination constraint. Results will be made available from `http://www.imn.htwk-leipzig.de/~abau/wst2014.html`. We believe that Matchbox is the first to produce a certified proof of termination of TRS/AProVE/JFP_Ex31. This termination problem had been solved by AProVE and Jambox in 2008, but not later.

Our approach allows to SAT-encode large constraints automatically. This is also the drawback: resulting formulas can get huge, and pose a challenge to SAT solvers. This motivates to work further on more efficient compilation in CO4 on the one hand, but also on tools (type systems) for statically analyzing the "circuit complexity" (the size of the generated formula) of constraint programs.

## References


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

2 Martin Avanzini. POP* and semantic labeling using SAT. In *ESSLLI Student Sessions*, volume 6211 of *LNCS*, pages 155–166. Springer, 2009.

3 Alexander Bau, Jörg Endrullis, and Johannes Waldmann. SAT compilation for termination proofs via semantic labelling. In *WST 2013*, 2013.

4 Alexander Bau and Johannes Waldmann. Propositional encoding of constraints over tree-shaped data. *CoRR*, abs/1305.4957, 2013.

5 Michael Codish, Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. SAT solving for termination proofs with recursive path orders and dependency pairs. *J. Autom. Reasoning*, 49(1):53–93, 2012.

6 Oleg Kiselyov, Chung chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *ICFP*, pages 192–203. ACM, 2005.

7 Adam Koprowski and Aart Middeldorp. Predictive labeling with dependency pairs using SAT. In *CADE*, volume 4603 of *LNAI*, pages 410–425. Springer-Verlag, 2007.

8 Claude Marché and Hans Zantema. The termination competition. In *RTA*, volume 4533 of *LNCS*, pages 303–313. Springer, 2007.

9 Christian Sternagel and René Thiemann. Modular and certified semantic labeling and unlabeling. In *RTA*, volume 10 of *LIPIcs*, pages 329–344. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

10 Christian Sternagel and René Thiemann. A relative dependency pair framework. In *Proc. WST'12*, pages 79–83, 2012.

11 Hans Zantema. Termination of term rewriting by semantic labelling. *Fundam. Inform.*, 24(1/2):89–105, 1995.

# Abstract: Fairness for Infinite-State Systems

**Byron Cook[1,2], Heidy Khlaaf[2], and Nir Piterman[3]**

1 **Microsoft Research**
2 **University College London**
3 **University of Leicester**

**Abstract**

In this extended abstract we introduce the first known tool for symbolically proving *fair*-CTL properties of (infinite-state) integer programs. Our solution is based on a reduction to existing techniques for fairness-free CTL model checking. The key idea is to use prophecy variables in the reduction for the purpose of symbolically partitioning fair from unfair executions.

**1998 ACM Subject Classification** D.2.4 Software/Program Verification, F.3.1 Specifying and Verifying and Reasoning about Programs, I.2.2 Automatic Programming

**Keywords and phrases** Model checking, Temporal logic, Fair CTL, Fair termination

## 1 Introduction

In model checking, fairness allows us to bridge between linear-time (*a.k.a.* trace-based) and branching-time (*a.k.a.* state-based) reasoning. Fairness is crucial, for example, to Vardi & Wolper's automata-theoretic technique for LTL verification [11]. Furthermore, when proving state-based CTL properties, we must often use fairness to model trace-based assumptions about the environment.

In this paper we introduce the first-known fair-CTL model checking technique for (infinite-state) integer programs. Our solution reduces fair CTL to fairness-free CTL using prophecy variables. We use the prophecy to encode a partition of fair from unfair paths. Prophecy variables introduce additional information into the state-space of the program under consideration, thus allowing fairness-free CTL proving techniques to reason only about fair executions.

Cognoscenti may at first find this result surprising. It is well known that fair termination of Turing machines cannot be reduced to termination of Turing machines. The former is $\Sigma^1_1$-complete and the latter is RE-complete [10].[1] For similar reasons fair-CTL model checking of Turing machines cannot be reduced to CTL model checking of Turing machines. The key to our reduction is the use of infinite non-deterministic branching: Recent approaches (*e.g.* [2, 6, 7]) facilitate model-checking fairness-free CTL over programs with infinite (discrete) non-deterministic branching. We use these results to provide a model checking procedure for fair CTL. As a consequence, in the context of infinite branching, fair and fairness-free CTL are equally difficult (and similarly for termination).

With our new technique we can build practical tools for automatically proving fair CTL of programs. We show the viability of our approach in practice using examples drawn from device drivers and algorithms utilizing shared resources.

[1] Sometimes termination refers to *universal termination*, which entails termination for *all* possible inputs. This is a harder problem and is co-$\text{RE}^{\text{RE}}$-complete.

$$\begin{array}{rcl} \textsc{Fair}((S, S_0, R, L), (p, q)) & \triangleq & (S_\Omega, S^0_\Omega, R_\Omega, L_\Omega) \\ \text{where} & & \end{array}$$

$$\begin{array}{rl} S_\Omega = & S \times \mathbb{N} \\ R_\Omega = & \{((s,n),(s',n')) \mid (s,s') \in R\} \wedge \\ S^0_\Omega = & S^0 \times \mathbb{N} \\ L_\Omega(s,n) = & L(s) \end{array} \quad \left( \begin{array}{c} (\neg p \wedge n' \leq n) \vee \\ (p \wedge n' < n) \vee \\ q \end{array} \right)$$

**Figure 1** Fair takes a system $(S, S_0, R, L)$ and a fairness constraint $(p, q)$ where $p, q \subseteq S$, and returns a new system $(S_\Omega, S^0_\Omega, R_\Omega, L_\Omega)$. Note that $n \geq 0$ is implicit, as $n \in \mathbb{N}$.

## 1.1 Intuition

The procedure builds on a transformation of infinite-state programs by adding a prophecy variable that truncates unfair paths. We start by presenting the transformation, followed by an illustrative example adapted for using said transformation, and subsequently our experimental results.

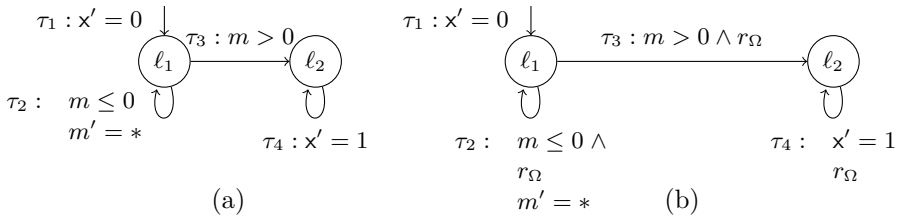
In Fig. 1, we propose a reduction $\textsc{Fair}(M, \Omega)$ that encodes an instantiation of the fairness constraint within a transition system. A transition system is $M = (S, S_0, R, L)$, where $S$ is a countable set of states, $S_0 \subseteq S$ a set of initial states, $R \subseteq S \times S$ a transition relation, and $L : S \to 2^{AP}$ a labeling function associating a set of propositions with every state $s \in S$. A *trace* or a *path* of a transition system is either a finite or infinite sequence of states. When given a transition system $(S, S_0, R, L)$ and a strong fairness constraint $\Omega = (p, q)$ where $p, q \subseteq S$, $\textsc{Fair}(M, \Omega)$ returns a new transition system that, through the use of a prophecy variable $n$, infers all possible paths that satisfy the fairness constraint, while avoiding all paths violating the fairness policy. Intuitively, $n$ is decreased whenever a transition imposing $p \wedge n' < n$ is taken. Since $n \in \mathbb{N}$, $n$ cannot decrease infinitely often, thus enforcing the eventual invalidation of the transition $p \wedge n' < n$. Therefore, $R_\Omega$ would only allow a transition to proceed if $q$ holds or $\neg p \wedge n' \leq n$ holds. That is, either $q$ occurs infinitely often or $p$ will occur finitely often. Note that a $q$-transition imposes no constraints on $n'$, which effectively resets $n'$ to an arbitrary value.

The conversion of $M$ with fairness constraint $\Omega$ to $\textsc{Fair}(M, \Omega)$ involves the truncation of paths due to the wrong estimation of the number of $p$-s until $q$. This means that $\textsc{Fair}(M, \Omega)$ can include (maximal) finite paths that are prefixes of unfair infinite paths. It follows that when model checking CTL we have to ensure that these paths do not interfere with the validity of our model checking procedure. Hence, we have to distinguish between maximal (finite) paths that occur in $M$ and those introduced by our reduction. This is done through adding a proposition $t$ to mark all original "valid" termination states prior to the reduction in Fig. 1, followed by adjusting the CTL specification through a transformation.

# 2 Illustrative Example

We first provide high-level understanding of our approach through an example.

Consider the example in Fig. 2 for the CTL property $\mathsf{AG}(\mathsf{x} = 0 \to \mathsf{AF}(\mathsf{x} = 1))$ and the fairness constraint $\mathsf{GF}\ \tau_2 \to \mathsf{GF}\ m > 0$ for the initial transition system introduced in (a). That is, we are attempting to prove that for all states, when $\mathsf{x} = 0$ then we must always eventually reach a state such that $\mathsf{x} = 1$ under the fairness constraint that if the transition $\tau_2$ occurs infinitely often, then $m$ must be greater than 0 infinitely often. We demonstrate the resulting transformation for this infinite-state program which allows us to reduce fair model checking to model checking. By applying $\textsc{Fair}(M, \Omega)$ from Fig. 1, we obtain (b) where each original transition, $\tau_2, \tau_3$, and $\tau_4$, are adjoined with restrictions such




$$r_\Omega : \{\ (\neg\tau_2 \wedge n' \leq n)\ \vee (\tau_2 \wedge n' < n)\ \vee m > 0\ \} \wedge n \geq 0$$

**Figure 2** Reducing a transition system with the CTL property $\mathsf{AG}(\mathsf{x} = 0 \rightarrow \mathsf{AF}(\mathsf{x} = 1))$ and the weak fairness constraint $\mathsf{GF}\ \tau_2 \rightarrow \mathsf{GF}\ m > 0$. The original transition system is represented in (a), followed by the application of our fairness reduction in (b).

that $\{(\neg\tau_2 \wedge n' \leq n) \vee (\tau_2 \wedge n' < n) \vee m > 0\} \wedge n \geq 0$ holds. That is, we wish to restrict our transition relations such that if $\tau_2$ is visited infinitely often, then the variable $m$ must be $> 0$ infinitely often. In $\tau_2$, the assignment $m' = *$ indicates that the variable $m$ is being assigned to a nondeterministic value, thus with every iteration of the loop, $m$ acquires a new value. In the original transition system, $\tau_2$ can be taken infinitely often given said nondeterminism, however in (b), such case is not possible. The transition $\tau_2$ in (b) now requires that $n$ be decreased on every iteration. Since $n \in \mathbb{N}$, $n$ cannot be decreased infinitely often, causing the eventual restriction to the transition $\tau_2$. Such an incidence is categorized as a finite path that is a prefix of some unfair infinite paths. As previously mentioned, such paths are disregarded. This leaves only paths where the prophecy variable "guessed" correctly. That is, it prophesized a value such that $\tau_3$ is reached, thus allowing our property to hold. The transformed figure in (b) can then be employed by an existing CTL model checking algorithm for infinite-state systems in order to verify the input CTL formula. We assume that the CTL model checking algorithm returns an assertion characterizing all the states in which a CTL formula holds. Tools such as Beyene *et al.* [2] and Cook *et al.* [4] support this functionality.

## 3 Experiments

We discuss the results of preliminary experiments with a prototype implementation. We applied our tool to several small programs: a classical mutual exclusion algorithm as well as code fragments drawn from device drivers. Our implementation is based on an extension to T2 [3, 8].[2] Despite theoretical contributions to the topic of fair CTL for infinite-state programs [1], there are no known tools supporting fair CTL for infinite-state programs. We are thus unable to make experimental comparisons.

Fig. 3 shows the results of our experiments. In our experiments we verified liveness properties, expressed in CTL. For each program we tested for both the success of the liveness property with a fairness constraint and its failure due to either a lack of fairness or a presence of a bug. A ✓ represents the existence of a validity proof, while $\chi$ represents the existence of a counterexample. We denote the lines of code in our program by LOC and the fairness constraint by FC.

Note that the Bakery algorithm is meant to be performed on a multi-threaded program. Due to our lack of support for concurrency, we have re-written the algorithm sequentially in

[2] New versions of T2 are not publicly available due to legal constraints. However, we are currently working through a release process with the Microsoft legal team.

| Program | LOC | Property | FC | Time(s) | Result |
|---|---|---|---|---|---|
| Windows Device Driver 1 | 20 | AG(PPBlockInits() ⇒ AFPPUnblockInits()) | Yes. | 14.43 | ✓ |
| Windows Device Driver 1 | 20 | AG(PPBlockInits() ⇒ AFPPUnblockInits()) | No. | 2.16 | χ |
| Windows Device Driver 1 + bug | 20 | AG(PPBlockInits() ⇒ AFPPUnblockInits()) | Yes. | 10.11 | χ |
| Bakery | 37 | AG(Noncritical ⇒ AFCritical) | No. | 16.43 | ✓ |
| Bakery | 37 | AG(Noncritical ⇒ AFCritical) | Yes. | 2.98 | χ |
| Bakery + bug | 37 | AG(Noncritical ⇒ AFCritical) | No. | 12.48 | χ |
| Windows Device Driver 2 | 374 | AG(KeAcquireSpinLock() ⇒ AFKeReleaseSpinLock()) | Yes. | 18.84 | ✓ |
| Windows Device Driver 2 | 374 | AG(KeAcquireSpinLock() ⇒ AFKeReleaseSpinLock) | No. | 14.12 | χ |
| Windows Device Driver 2 + bug | 374 | AG(KeAcquireSpinLock() ⇒ AFKeRelaseSpinLock()) | Yes. | 18.94 | χ |
| Windows Device Driver 3 | 58 | AF(KeEnCriticalRegion() ⇒ EGKeExCriticalRegion()) | Yes. | 12.58 | χ |
| Windows Device Driver 3 | 58 | AF(KeEnCriticalRegion() ⇒ EGKeExCriticalRegion) | No. | 9.62 | ✓ |

**Figure 3** Windows Device Driver 1 uses the fairness constraint GF(IoCreateDevice.exit{1}) ⇒ GF(status! = STATUS_OBJ_NAME_COLLISION). Windows Device Driver 2 and 3 utilize the same fairness constraint in relation to checking the acquisition and release of spin locks and the entrance and exit of critical regions, respectively. The Bakery algorithm utilizes a fairness constraint of the form GF(p) ⇒ GF(q) with p and q being relative to our sequential implementation.

a manner which simulates the behavior of a multi-threaded program. The variables p and q in the fairness constraint of Bakery denote specific program locations in our sequential algorithm.

For the existential fragment of CTL, fairness constraints restrict the transition relations required to prove an existential property, as demonstrated by Windows Device Driver 3. For universal CTL properties, fairness policies can assist in enforcing properties to hold that previously did not. Thus, our tool allows us to both prove and disprove the negation of each of the properties.

## 4 Discussion

We have shown the first-known fair-CTL model checking technique for integer based infinite-state programs through a reduction to existing techniques for fairness-free CTL model checking. The reduction relies on utilizing prophecy variables to introduce additional information into the state-space of the program under consideration. This allows fairness-free CTL proving techniques to reason only about fair executions. Our implementation seamlessly builds upon existing CTL proving techniques, resulting in experiments which demonstrate the practical viability of our approach.

Furthermore, our technique allows us to bridge between linear-time (LTL) and branching-time (CTL) reasoning. Cook *et al.* [5] have described an iterative symbolic determination procedure which efficiently uses techniques associated with CTL to verify a subset of LTL properties. However, these corresponding branching time techniques provide no support for verifying fair-CTL, thus excluding a large set of linear-time liveness properties necessitating fairness. Our contribution would thus allow for full support of LTL verification via CTL model checking techniques. Not only so, but a seamless integration between LTL and CTL

reasoning may make way for further extensions supporting CTL* verification of infinite-state programs [9]. We hope to further examine both the viability and practicality of such an extension.

## References


1 K. Apt and E. Olderog. Fairness in parallel programs: The transformational approach. *ACM Transactions on Programming Languages and Systems*, 10, 1988.

2 T. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified horn clauses. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, 2013.

3 M. Brockschmidt, B. Cook, and C. Fuhs. Better termination proving through cooperation. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, 2013.

4 B. Cook, H. Khlaaf, and N. Piterman. Faster temporal reasoning for infinite-state programs. Technical Report CS-14-001, University of Leicester, 2014.

5 B. Cook and E. Koskinen. Making prophecies with decision predicates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, 2011.

6 B. Cook and E. Koskinen. Reasoning about nondeterminism in programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–230. ACM, 2013.

7 B. Cook, E. Koskinen, and M. Vardi. Temporal property verification as a program analysis task. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, 2011.

8 B. Cook, A. See, and F. Zuleger. Ramsey vs. lexicographic termination proving. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, 2013.

9 E.A. Emerson and J.Y Halpern. "sometimes" and "not never" revisited: On branching versus linear time temporal logic. *J. ACM*, 33(1), January 1986.

10 D. Harel. Effective transformations on infinite trees, with applications to high undecidability, dominoes and fairness. *Journal of the ACM*, 33:224–248, 1986.

11 M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37, 1994.

# Reducing Deadlock and Livelock Freedom to Termination*

**Byron Cook[1,2,3], Stephen Magill[4], Matthew Parkinson[1], and Thomas Ströder[5]**

1 **Microsoft Research Cambridge, UK**
{bycook,mattpark}@microsoft.com
2 **Microsoft Research NYC, USA**
3 **University College London, UK**
4 **Unaffiliated**
stephen.magill@gmail.com
5 **RWTH Aachen University, Germany**
stroeder@informatik.rwth-aachen.de

**Abstract**

In this paper we introduce a general method for proving deadlock and livelock freedom of concurrent programs with shared memory. Our goal in this work is to support programs which use locks stored in mutable data structures. The key to our technique is the observation that dependencies between locks can be abstracted using recursion and non-determinism in a sequential logic program such that termination of the abstraction implies deadlock and livelock freedom of the original program.

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

**Keywords and phrases** deadlock, livelock, concurrency, termination, shared-memory

## 1 Introduction

Concurrent programs that use locks stored in mutable data structures are especially hard to show deadlock/livelock free, as a static lock acquisition order is difficult to impose given memory allocation, de-allocation and re-allocation. Currently the search for effective automatic methods for this problem remains open.

In this paper we present a procedure that addresses this problem via abstraction to the question of termination of a sequential logic program. We introduce an abstraction method that—using recursion and non-determinism—captures the dependencies between locks in the original concurrent program. Termination of our abstraction implies acyclicity of the lock dependency relation, and hence deadlock freedom. Moreover, termination of our abstraction also implies that threads cannot diverge while holding a lock. Together with the property that locks cannot be acquired infinitely often (which can be proved using existing techniques), this establishes livelock freedom.

To show the viability of our approach, we have successfully applied it to a series of challenging problems, among them a lock-coupling list [10], the optimistic list of Heller et al. [5], the lock-based queue of Michael and Scott [7], and the balanced binary trees of Kung and Lehman [6], Bronson et al. [1], and Schlatter Ellis [8]. The last example is particularly challenging, as it uses different lock types which are inter-dependent. To the best

* This work was supported by Microsoft Research Cambridge.

of our knowledge this represents the first known fully automatable proofs of deadlock/livelock freedom for these published algorithms.[1]

## 2 Approach

Our approach is based on showing two properties: (1) locks cannot be acquired infinitely often and (2) each lock acquired by the program is eventually released again. Together these imply deadlock and livelock freedom of the concurrent program. Note that we assume a fair scheduler, as property (2) is trivially false otherwise.

We use the previous separation logic based techniques underlying the tool CAVE [11] to prove memory safety, data structure consistency, and to learn the possible effects on the shared data structures, expressed as actions [2]. We then use another previous technique [4] to establish that those actions are not executed infinitely often, which already implies (1).

Next we employ our abstraction construction, which is designed to represent the dependencies between the locks as a sequential logic program. The abstraction is based on the fact that whenever a thread $T_1$ tries to acquire a lock held by another thread $T_2$, $T_2$ must first reach a corresponding unlock operation before $T_1$ can continue its execution.[2] If we interpret this dependency of the further execution of $T_1$ on the execution of $T_2$ as a control switch from $T_1$ to $T_2$ until the latter reaches the matching unlock operation, we can express this dependency as a function call (using lock operations as entry or call points and corresponding unlock operations as exit or return points). However, when switching control from a thread $T_1$ to another thread $T_2$, we do not know which lock operation in $T_2$'s code was executed to acquire the lock $T_1$ is waiting for. Hence, in our abstraction we need to allow a non-deterministic choice that could jump to any lock operation of $T_2$. Moreover, we need to pass a representation of the current state of the shared data structure where the lock has been acquired as an argument of each call to establish a relation on the heap locations where locks are acquired. Between the calls to lock operations, our abstraction mimics the behaviour of the program on the shared data structure. Because we allow non-deterministic jumps to arbitrary lock operations, we use the same predicate in the resulting logic program for all lock operations. Thus, we will obtain a recursive and non-deterministic, but sequential program simulating the control switches between the threads whenever they need to wait for a lock to be released. If this program is terminating, we know that no thread has to wait forever to release a lock. Together with an assumption of a fair scheduler, this implies (2): If a thread fails to terminate after acquiring a lock, so will the abstraction. Together with the already established property that lock actions cannot be executed infinitely often, this implies livelock freedom. Furthermore, the lock abstraction forms a disjunctive relation on heap locations representing all possible lock orderings. If the program is terminating, this relation is well-founded and cannot contain circular dependencies. This implies deadlock freedom.

**Example.** Consider the the Java-like program in Figure 1, which implements an add operation in a concurrent finite acyclic list that uses hand-over-hand locking to increase the lock-granularity. The syntax `<c>` is used to indicate the atomic execution of the command `c`.

[1] Caveat: Some of the steps in our method depend on previously published tools that are no longer available. Thus, these steps are currently performed manually, but could be automated with additional development work to replace these old tools.

[2] In principle, this is an over-approximation if locks can be released by threads which did not acquire them. However, if $T_2$ itself releases the lock, we can be sure that it is no longer held by $T_2$.

```
1  /* Create a node */
2  Node node(int e,
3            Node next) {
4    <h = new();>
5    <h.lock = 0;>
6    <h.value = e;>
7    <h.next = next;>
8    return h;
9  }
10
11 /* Add e to sorted
12    list l.        */
13 add(List l, int e) {
14   Node p,c,h;
15   int v;
16   lock(l);
17   <p = l.head;>
18   if (p == null) {
19     // empty list
20     h = node(e, null);
21     <l.head = h;>
22     unlock(l);
23     return;
24   }
25   lock(p);
26   <v = p.value;>
27   if (v >= e) {
28     // new first node
29     h = node(e, p);
30     <l.head = h;>
31     unlock(l);
32     unlock(p);
33     return;
34   }
35   unlock(l);
36   <c = p.next;>
37
38   /* hand-over-
39      hand locking  */
40   while (c != null) {
41     lock(c);
42     <v = c.value;>
43     if (v >= e) {
44       h = node(e, c);
45       <p.next = h;>
46       unlock(p);
47       unlock(c);
48       return;
49     }
50     unlock(p);
51     p = c;
52     <c = p.next;>
53   }
54   h = node(e, null);
55   <p.next = h;>
56   unlock(p);
57   return;
58 }
```

**Figure 1** Thread-safe insert into sorted list, managed using hand-over-hand locking.

The type `Node` describes a node in a singly-linked list with fields `lock`, `value`, and `next`. The type `List` is a record with fields `head` and `lock`, the former pointing to the head of the list.

Our aim is to prove deadlock/livelock freedom of a program with an arbitrary finite number of threads, each executing the code in Figure 1 with arbitrary values passed to the add operation. The difficulty in proving this concurrent program deadlock- and livelock-free is the hand-over-hand locking seen in the program: it is a common pattern in concurrent programs, and unfortunately it precludes the use of any previously known fully-automatic approach to prove deadlock or livelock freedom.

The actions which can occur in this program are adding (line 21, 30, 45, or 55), locking (line 16, 25, or 41) and unlocking (line 22, 31, 32, 35, 46, 47, 50, or 56) a node in the list. It is easy to see that adding a node cannot be executed infinitely often as the code (and hence each thread) directly terminates thereafter. Lock and unlock actions both occur in the loop starting at line 40, so they occur finitely often if this loop always terminates. The number of loop iterations is bounded by the finite length of the original list plus the finite number of add actions executed by the threads. Hence, no actions can be executed infinitely often, establishing property (1).

We show property (2) automatically by creating a *lock abstraction*, which characterizes the paths from each lock operation to the matching unlock operation. In our example, if we start at a lock operation, we can at most traverse one more lock operation before reaching a matching unlock operation (*e.g.* from line 41, we either directly reach a matching unlock operation at line 47 or 56 without traversing any further lock operations, or we enter the loop again and definitely reach a matching unlock operation at line 46 or 50 after traversing the lock operation at line 41 again – note that the position in the list where the lock has been acquired originally is referenced by `p` after executing line 51). Moreover, each such lock dependency acquires the second lock at the successor of the node where the first lock has been acquired. Having a finite list which only grows finitely often, this process cannot be repeated infinitely often. Also, as the direction in which further locks are acquired is always the same, it is not possible to form a circular dependency. This shows that eventually every attempt to acquire a lock is successful, which is equivalent to property (2).

This reasoning is captured by the logic program in Figure 2, which results from our construction. Here, we observe that the data structure is represented by terms using the constructors `h/1` and `lh/1` for the head structure, `n/2` and `ln/2` for nodes within the list, and `nil` for the empty list. The respective versions with the prefix `l` denote structure parts which are known to be locked by some thread while the other versions denote an unknown lock

```
lock(D,N).
lock(D1,N1) :-
  p1617182425(
    D1,E1,P1,C1,H1,V1,
    L2,E2,P2,C2,H2,V2),
  interf(L2,P2,C2,H2,N1,
         L3,P3,C3,H3,N2),
  lock(P3,N2).
lock(D1,N1) :-
  p2526273435364041(
    L1,E1,D1,C1,H1,V1,
    L2,E2,P2,C2,H2,V2),
  interf(L2,P2,C2,H2,N1,
         L3,P3,C3,H3,N2),
  lock(C3,N2).
lock(D1,N1) :-
  p414243495051524041(
    L1,E1,P1,D1,H1,V1,
    L2,E2,P2,C2,H2,V2),
  interf(L2,P2,C2,H2,N1,
         L3,P3,C3,H3,N2),
  lock(C3,N2).
```

```
p1617182425(
  h(n(I,R)),E,P,C,H,V,
  lh(n(I,R)),E,n(I,R),
  C,H,V).
p2526273435364041(
  lh(n(I1,n(I2,R1))),E,
  n(I1,n(I2,R1)),C,H,V,
  h(R2),E,ln(I1,n(I2,R1)),
  n(I2,R1),H,I1).
p414243495051524041(
  h(R1),E,
  ln(I1,n(I2,n(I3,R2))),
  n(I2,n(I3,R2)),H,I1,
  h(R1),E,ln(I2,n(I3,R2)),
  n(I3,R2),H,I2).
interf(L,P,C,H,N,
       L,P,C,H,N).
interf(L1,P1,C1,H1,N1,
       L2,P2,C2,H2,N2) :-
  envstep(L1,N1,L3,N3),
  interf(L3,P1,C1,H1,N3,
         L2,P2,C2,H2,N2).
```

```
interf(L1,P1,C1,H1,N1,
       L2,P2,C2,H2,N2) :-
  envstep(P1,N1,P3,N3),
  interf(L1,P3,C1,H1,N3,
         L2,P2,C2,H2,N2).
interf(L1,P1,C1,H1,N1,
       L2,P2,C2,H2,N2) :-
  envstep(C1,N1,C3,N3),
  interf(L1,P1,C3,H1,N3,
         L2,P2,C2,H2,N2).
interf(L1,P1,C1,H1,N1,
       L2,P2,C2,H2,N2) :-
  envstep(H1,N1,H3,N3),
  interf(L1,P1,C1,H3,N3,
         L2,P2,C2,H2,N2).
envstep(n(I1,R1),s(N1),
        n(I1,R2),N2) :-
  envstep(R1,N1,R2,N2).
envstep(ln(I1,R1),s(N1),
        ln(I1,R2),N2) :-
  envstep(R1,N1,R2,N2).
```

```
envstep(h(R1),s(N1),
        h(R2),N2) :-
  envstep(R1,N1,R2,N2).
envstep(lh(R1),s(N1),
        lh(R2),N2) :-
  envstep(R1,N1,R2,N2).
envstep(R1,s(N),R2,N) :-
  addH(R1,R2).
envstep(R1,s(N),R2,N) :-
  addHN(R1,R2).
envstep(R1,s(N),R2,N) :-
  addNN(R1,R2).
envstep(R1,s(N),R2,N) :-
  addN(R1,R2).
addH(h(nil),
     h(n(I2,nil))).
addHN(h(n(I1,R)),
      h(n(I2,n(I1,R)))).
addNN(n(I1,n(I2,R)),
      n(I1,n(I3,n(I2,R)))).
addN(n(I1,nil),
     n(I1,n(I3,nil))).
```

**Figure 2** Lock abstraction whose termination implies deadlock/livelock freedom.

status. Using this term representation, the `lock` predicate encodes the lock dependencies. Its arguments are the state of the data structure where a lock is acquired and a prophecy counter stating how many operations can still be performed by the interfering environment (*i.e.* other threads). The use of the latter is motivated by the fact that all actions are known to be executed only finitely often. Next, a postcondition transformer (starting with `p` and displaying the two digits line numbers of the path it follows in its name) is used to mimic the program behaviour along a path on the program variables. Afterwards, the `interf` predicate is used to apply the four add actions (corresponding to the four locations where nodes are added) non-deterministically to the data structure representations pointed to by the program variables (variables which provably never point to the shared data structure are excluded) while the prophecy counter is reduced accordingly. Finally, we have the recursive call to the `lock` predicate. Its first argument is the respective program variable passed to the lock operation which might block the current thread's execution. Its second argument is the number of remaining interference operations. The dependency described above is encoded by the last clause for the `lock` predicate.

The lock abstraction captures all execution paths that start at a lock operation and traverse further lock operations before reaching the matching unlock operation for the initial lock operation. Termination of the lock abstraction program thus implies that each acquired lock must eventually be released. This implies deadlock freedom and, together with the fact that locks cannot be acquired infinitely often, also implies livelock freedom.

To prove termination automatically, we apply the termination prover AProVE [9, 3], which proves the abstraction in our example terminating in 15 seconds. Our method scales to bigger programs, as the complexity of the abstraction is dependent on the number of actions and the size of the critical sections protected by the locks (*i.e.* those parts of the program between lock and corresponding unlock operations). Both are typically small in practice to allow as much concurrency as possible.

## 3 Conclusion

The problem of proving deadlock and livelock freedom is made difficult by the level of cross thread lock dependencies. In this paper we have presented an abstraction technique that encodes the lock dependencies from the concurrent program into a sequential logic program

such that existing termination tools prove livelock and deadlock freedom. Our approach has been successfully applied to a series of challenging problems, including a lock-coupling list [10], the optimistic list of Heller et al. [5], the lock-based queue of Michael and Scott [7], and the balanced binary trees of Kung and Lehman [6], Bronson et al. [1], and Schlatter Ellis [8].

**Acknowledgements** We would like to thank the following people for making valuable comments on a draft of an extended version of this paper: Aws Albarghouthi, Marc Brockschmidt, James Brotherston, Isil Dillig, Thomas Dillig, Carsten Fuhs, Jürgen Giesl, Kareem Khazem, Heidy Khlaaf, Zachary Kincaid, Eric Koskinen, Rustan Leino, Peter O'Hearn, and Viktor Vafeiadis. We would also like to thank Viktor Vafeiadis for his support on using CAVE.

## References

1 Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 257–268, New York, NY, USA, 2010. ACM.

2 Cristiano Calcagno, Matthew Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis*, volume 4634 of *Lecture Notes in Computer Science*, pages 233–248. Springer Berlin Heidelberg, 2007.

3 Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs. Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, PPDP '12, pages 1–12, New York, NY, USA, 2012. ACM.

4 Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 16–28, New York, NY, USA, 2009. ACM.

5 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. In James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *Principles of Distributed Systems*, volume 3974 of *Lecture Notes in Computer Science*, pages 3–16. Springer Berlin Heidelberg, 2006.

6 H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, September 1980.

7 Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

8 Carla Schlatter Ellis. Concurrent search and insertion in AVL trees. *IEEE Transactions on Computers*, C-29(9):811–817, 1980.

9 Peter Schneider-Kamp, Jürgen Giesl, Alexander Serebrenik, and René Thiemann. Automated termination proofs for logic programs by term rewriting. *ACM Transactions on Computational Logic*, 11(1):2:1–2:52, November 2009.

10 Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge Computer Laboratory, 2008.

11 Viktor Vafeiadis. RGSep action inference. In Gilles Barthe and Manuel Hermenegildo, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 5944 of *Lecture Notes in Computer Science*, pages 345–361. Springer Berlin Heidelberg, 2010.

# Another Proof for the Recursive Path Ordering

**Nachum Dershowitz**

**School of Computer Science, Tel Aviv University**
**Ramat Aviv, Israel**
nachum.dershowitz@cs.tau.ac.il

**Abstract**

Yet another proof of well-foundedness of the (multiset) recursive path ordering is provided.

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

**Keywords and phrases** Path orderings, recursive path ordering, well-foundedness, termination, order types

## 1 Introduction

The *recursive path ordering* [3] is a popular family of well-founded orderings on terms (trees), used for proving termination of functional programs (e.g. [8]) and rewrite systems (e.g. [11, 9, 2]) and for guiding completion procedures and theorem provers (e.g. [13]). See [4].

We give a new proof of its well-foundedness in what follows. Some previous proof approaches may be found in [3, 15, 10, 12, 1, 5]. Though some of the orderings in these references differ, when function symbols are totally ordered, they all coincide [16]. So a proof of one is a proof of all.

## 2 Basics

Let $F$ be a set of symbols, let $A, G \subseteq F$, and let $T_G(A)$ denote the (finite ordered) trees constructed with leaves (atoms) taken from $A$ and (internal) nodes from $G$. We are also given a well-founded partial ordering $\succ$ on $F$ (a *precedence*); *we will assume throughout that each leaf in $A$ is greater in this ordering than every node in $G$.*

Let $T_G^n(A)$ denote those trees in $T_G(A)$ in which the (maximum) nesting of *maximal* nodes in $G$ is at most $n$. So

$$T_F(A) \;=\; \bigcup_{n=0}^{\infty} T_F^n(A)$$

We refer to $n$ as the *altitude* of those trees that are in $T_G^n(A) \setminus T_G^{n-1}(A)$.

It is convenient to let

$$\max B \;=\; \{f \in F : \nexists\, g \in B.\ g \succ f\}$$

be all the *maximal* elements of partially ordered set $B$ and

$$^{<}B \;=\; B \setminus \max B \;=\; \{f \in F : \exists g \in B.\ g \succ f\}$$

be all the rest (the non-maximal elements in $B$). It can be that $B = {}^{<}B$ when $B$ has no maximal elements ($\max B = \varnothing$), as for the natural numbers, for example.

Bags (finite multisets)

$$\mathcal{M}(A) \;=\; \{\{\!|a_1, \ldots, a_\ell|\!\} : a_1, \ldots, a_\ell \in A,\ \ell \in \mathbb{N}\}$$

of elements of well-founded $A$ are known to be well-founded under the *bag* (multiset) ordering [7]. The bag ordering $\gg$ on $A \cup \mathcal{M}(A)$ may be defined as the transitive closure of the following rules:

$$\frac{}{\{\!|a|\!\} \gg a} \qquad \frac{a > b_1, \ldots, b_\ell}{\{\!|a|\!\} \gg \{\!|b_1, \ldots, b_\ell|\!\}} \qquad \frac{\{\!|a_1, \ldots, a_k|\!\} \gg \{\!|b_1, \ldots, b_\ell|\!\}}{\{\!|c, a_1, \ldots, a_k|\!\} \gg \{\!|c, b_1, \ldots, b_\ell|\!\}}$$

$k, \ell \geq 0$, $a, a_i, b_j, c \in A$. This makes a bag bigger than each of its elements.

We also allow colored bags, like red bags $\{\!|1,1,3|\!\} \in \mathcal{M}_r(\mathbb{N})$ and blue bags $\{\!|a,c,c|\!\} \in \mathcal{M}_b(a..z)$, which are incomparable. The ordering rules are color specific.

Given a partial ordering $\succcurlyeq$ on $F = A \cup G$, the original (multiset) path ordering $>$ on $T_G(A)$ is the transitive closure of the following recursive rules:

$$\frac{}{f(\ldots, a_i, \ldots) > a_i} \qquad \frac{f \succcurlyeq g,\ f(a_1, \ldots, a_k) > b_1, \ldots, b_\ell}{f(a_1, \ldots, a_k) > g(b_1, \ldots, b_\ell)} \qquad \frac{\{\!|a_1, \ldots, a_k|\!\} \gg \{\!|b_1, \ldots, b_\ell|\!\}}{f(a_1, \ldots, a_k) > f(b_1, \ldots, b_\ell)}$$

$k, \ell \in \mathbb{N}$, $f, g \in F$, $a_i, b_j \in T_G(A)$. The bags in the last rule are compared recursively in the presently defined ordering.

In the next section, we propose an alternative definition for this path ordering. The idea is to transform trees before comparing by turning each subtree rooted in a node labeled by maximal symbol into a leaf containing the bag of that node's children.

It pays to recall some properties of the original definition: The following facts hold for the path ordering $>$:

- If a tree $s$ contains a symbol (node or leaf) that is larger than every symbol in another tree $t$, then $s > t$.
- A leaf $s$ is bigger than a non-leaf tree $t$ iff $s$ is bigger than all leaves (and nodes) of $t$. (As stated earlier, we are presuming that all leaves are bigger than all nodes.)
- A leaf $s$ is smaller than a non-leaf tree $t$ iff $s$ is smaller than or the same as some leaf (or node) of $t$.

## 3 Repackaging

We deal here with the case where $F$ is totally (well-) ordered. To compare trees over $F$, take each maximal subtree rooted in the maximal element in $F$ and turn it into a bag of lower trees. Let this operation on a tree $t$ be denoted $\widehat{t}$. If $g$ is the largest node in $t = u[v_1, \ldots, v_k]$, which can be decomposed into a "cap" $u$ not containing $g$ and subtrees $v_1, \ldots, v_k$ each headed by $g$, then $\widehat{t} = u[a_1, \ldots, a_k]$ where each $a_i$ is a leaf labeled by the bag of immediate subtrees of $v_i$. The nodes of $\widehat{t}$ are all smaller than $g$. The new leaves of $\widehat{t}$ contain only strictly lower trees than the original $t$.

To compare trees $s$ and $t$, one first sees which has the largest leaf, then which has the largest node. Those being equal, this is followed by comparing the decomposed trees $\widehat{s}$ and $\widehat{t}$, with the new leaves made larger than the remaining node labels.

Let $B_n$ be short for $T^n_G(A)$ and let $\mathcal{M}^+(B_n) = \mathcal{M}(B_n) \cup A$ be bags of these trees plus leaves $A$, with leaves ordered above than these bags. Trees $T_G(A) = \bigcup_{n=0}^{\infty} B_n$ are viewed and compared inductively as follows:

$$B_0 = T_{\lessdot G}(A) \qquad \text{— provided } {\lessdot G} \subsetneq G \tag{1}$$

$$B_{n+1} = \mathcal{M}^+_g(B_n) \times G \times T_{\lessdot G}\big(\mathcal{M}^+_g(B_n)\big) \qquad \text{— where } g = \max G \tag{2}$$

$$B_0 = \bigcup_{H \lessdot G} T_H(A) \qquad \text{— if } \max G = \varnothing \tag{3}$$

where the $H \lessdot G$ are proper initial segments of ordered $G$. (The initial segments of $\mathbb{N}$ are $[0..i]$ for all $i \in \mathbb{N}$, for example.) The new leaves $\mathcal{M}_g(B_n)$ are placed below $A$ and above $\lessdot G$ in the leaf ordering.

1. The first case is just two ways of saying that all nodes are non-maximal in $G$.
2. The second means that a tree of altitude $n+1$ can be viewed instead as a tree built from smaller nodes and from leaves that are bags of lower trees $B_n$, first comparing the trees' maximal leaves and nodes and then the trees themselves.
3. What if $G$ has no maximal element and so $G = \lessdot G$, in which case the first case doesn't apply? Still each tree has a maximal element, and so any two trees may be compared according to the ordering of trees with nodes up to the largest node in either. (Technically, the first case is subsumed by this one.)

▸ **Example 1.** Consider binary trees built from leaves $A = a \gtrdot b \gtrdot c$ and internal nodes $G : \uparrow \gtrdot \times \gtrdot +$. The trees $T_G^1(A)$ of altitude one include $(b \uparrow b) + (a \times (b + c))$ and $(b \uparrow b) + ((a \times b) + (a \times c))$—think distributivity. They both have the same maximal node $\uparrow$ and the same maximal leaf $a$. Applying the above decomposition yields

$$\boxed{\lbag b,\, b \rbag} + (a \times (b + c))$$

and

$$\boxed{\lbag b,\, b \rbag} + ((a \times b) + (a \times c))$$

respectively, where each box is a leaf. Now they both have the same maximal node $\times$ and the same maximal leaf $a$ as before. The next decompositions are

$$\boxed{\lbag b,\, b \rbag} + \boxed{\lbag a,\, b + c \rbag}$$

and

$$\boxed{\lbag b,\, b \rbag} + \left( \boxed{\lbag a,\, b \rbag} + \boxed{\lbag a,\, c \rbag} \right)$$

with maximal leaf $\wr b, b \int$. One more step gives

$$\left\langle \boxed{\lbag b,\, b \rbag},\, \boxed{\lbag a,\, b + c \rbag} \right\rangle$$

and

$$\left\langle \boxed{\lbag b,\, b \rbag},\, \boxed{\lbag a,\, b \rbag} + \boxed{\lbag a,\, c \rbag} \right\rangle$$

To compare these two leaves, we compare the bags

$$\left\langle \boxed{\lbag b,\, b \rbag},\, \boxed{\lbag a,\, b + c \rbag} \right\rangle$$

and

$$\left\langle \boxed{\lbag b,\, b \rbag},\, \boxed{\lbag a,\, b \rbag} + \boxed{\lbag a,\, c \rbag} \right\rangle$$

The first is larger since its leaf element $\wr a, b + c \int$ is larger than the leaves $\wr a, b \int$ and $\wr a, c \int$ of the tree $\wr a, b \int + \wr a, c \int$. That is because $b + c$ is bigger than both $b$ and $c$.

To see that the new version is well-founded, consider an infinite descending sequence and reason by induction on maximal node and altitude. Since the leaves and nodes are well-ordered, the maximal leaf and maximal node stabilize from some point on. Look at the decompositions of those trees, all of whose nodes are strictly smaller than the maximal node in the original sequence. So, by induction, the old and new leaves are well-ordered, and hence that sequence of decompositions must in fact be finite.

The new tree ordering on $T_F(\{\top\})$ (starting with maximal leaves) is identical to the original path ordering. Any tree can be put in this form by sprouting a $\top$-leaf from each original leaf.

## 4 Discussion

We are hopeful that our redefinition of the recursive path ordering may facilitate extensions beyond $\Gamma_0$, which are of value for demonstrating termination of "non-simplifying" rewriting systems. The reason is that the definition given here bears a measure of similarity to the ordering of ordinal diagrams; see [14, 6].

When the ordering on nodes is partial, there may be more than one maximal node. Each should get its own incomparable bag of shallower trees. The details remain to be worked out.

## References


1 Frédéric Blanqui, Jean-Pierre Jouannaud, and Albert Rubio. The computability path ordering: The end of a quest. In Michael Kaminski and Simone Martini, editors, *Proceedings of the 17th Annual Conference on Computer Science Logic (CSL, Bertinoro, Italy)*, volume 5213 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2008. Available at `http://www.lsi.upc.edu/~albert/papers/csl08.pdf` (viewed May 20, 2014).

2 Evelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Automated certified proofs with CiME 3. In Manfred Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications (RTA, Novi Sad, Serbia)*, volume 10 of *LIPIcs*, pages 21–30. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2011. Available at `http://cedric.cnam.fr/fichiers/art_2044.pdf` (viewed May 20, 2014).

3 Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982. Available at `http://nachum.org/papers/Orderings4TRS.pdf` (viewed May 20, 2014).

4 Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1&2):69–115, February/April 1987. Available at `http://nachum.org/papers/termination.pdf` (viewed May 20, 2014).

5 Nachum Dershowitz. Jumping and escaping: Modular termination and the abstract path ordering. *Theoretical Computer Science*, 464:35–47, 2012. Available at `http://nachum.org/papers/Toyama.pdf` (viewed May 20, 2014).

6 Nachum Dershowitz. Ordinal path orderings. In *Informal Proceedings of the 13th International Workshop on Termination (WST 2013, Bertinoro, Italy)*, pages 31–35, August 2013. Available at `http://nachum.org/papers/OPO.pdf` (viewed May 20, 2014).

7 Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM (CACM)*, 22(8):465–476, August 1979.

8 Jürgen Giesl. Termination analysis for functional programs using term orderings. In Alan Mycroft, editor, *Proceedings Second International Symposium on Static Analysis (SAS '95, Glasgow, UK)*, volume 983 of *Lecture Notes in Computer Science*, pages

154–171. Springer, 1995. Available at `http://verify.rwth-aachen.de/giesl/papers/SAS-report.ps` (viewed May 20, 2014).

**9** Nao Hirokawa and Aart Middeldorp. Tsukuba Termination Tool. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA)*, 2003. Available at `http://colo6-c703.uibk.ac.at/ttt/rta03.pdf` (viewed May 20, 2014).

**10** Jean-Pierre Jouannaud, Pierre Lescanne, and Fernand Reinig. Recursive decomposition ordering. In Dines Bjørner, editor, *Proceedings of the Second IFIP Workshop on Formal Description of Programming Concepts (Garmisch-Partenkirchen, West Germany)*, pages 331–348. North-Holland, Amsterdam, June 1982.

**11** Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Automated termination proofs with AProVE. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA)*, 2004. Available at `http://verify.rwth-aachen.de/giesl/papers/RTA04-distribute.pdf` (viewed May 20, 2014).

**12** Jan Willem Klop, Vincent van Oostrom, and Roel C. de Vrijer. Iterative lexicographic path orders. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, volume 4060 of *Lecture Notes in Computer Science*, pages 541–554. Springer, 2006. Available at `http://www.cs.vu.nl/~tcs/trs/ilpogoguen.pdf` (viewed May 19, 2014).

**13** William McCune. Prover9 manual. `http://www.cs.unm.edu/~mccune/prover9/manual/2009-11A` (viewed May 19, 2014).

**14** Mitsuhiro Okada. A simple relationship between Buchholz's new system of ordinal notations and Takeuti's system of ordinal diagrams. *The Journal of Symbolic Logic*, 52(3):577–581, 1987.

**15** David A. Plaisted. A recursively defined ordering for proving termination of term rewriting systems. Report R-78-943, Department of Computer Science, University of Illinois, Urbana, IL, September 1978. Available at `https://ia601701.us.archive.org/9/items/recursivelydefin943plai/recursivelydefin943plai.pdf` (viewed May 19, 2014).

**16** Michael Rusinowitch. Path of subterms ordering and recursive decomposition ordering revisited. In Jean-Pierre Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science*, pages 225–240. Springer, Berlin, 1985.

# Non-termination using Regular Languages

**Jörg Endrullis[1] and Hans Zantema[2]**

1 **VU University Amsterdam, The Netherlands**
2 **Eindhoven University of Technology, The Netherlands**

**Abstract**

We describe a method for proving non-termination of term rewriting systems that do not admit looping reductions. As certificates of non-termination, we employ regular (tree) automata.

**1998 ACM Subject Classification** D.1.1, D.3.1, F.4.1, F.4.2, I.1.1, I.1.3

**Keywords and phrases** non-termination, finite automata, regular languages

## 1 Introduction

We describe a method for proving non-termination of term rewriting systems that do not admit looping reductions, that is, reductions from a term $t$ to a term $C[t\sigma]$ containing a substitution instance of $t$. For this purpose, we employ tree automata as certificates of non-termination. For proving non-termination of a term rewriting system $R$, we search a tree automaton $A$ whose language $\mathcal{L}(A)$ is not empty, weakly closed under rewriting and every term of the language contains a redex occurrence. We have fully automated the search for these certificates employing SAT-solvers.

All automata that we use as example in this paper have been found automatically; this concerns in particular fully automated proofs of non-termination for the following two rewrite systems.

▶ **Example 1.** We consider the following string rewriting system:

$$zL \to Lz \qquad Rz \to zR \qquad bL \to bR \qquad Rb \to Lzb$$

This rewrite system admits no reductions of the form $s \to^* \ell sr$.

▶ **Example 2.** We consider the $S$-rule from combinatory logic:

$$ap(ap(ap(S,x),y),z) \to ap(ap(x,z),ap(y,z))$$

For the $S$-rule it is known that there are no reductions $t \to^* C[t]$ for ground terms $t$, see [15]. For open terms $t$ the existence of reductions $t \to^* C[t\sigma]$ is open.

It turns out that the method can be fruitfully applied to obtain non-termination proofs of several string rewriting systems that have remained unsolved in the last full run of the termination competition.

### Related Work

The paper [11] investigates necessary conditions for the existence of loops. The work [17] employs SAT solvers to find loops, [18] uses forward closures to find loops efficiently, and the wook [16] introduces 'compressed loops' to find certain forms of (possibly very long) loops.

Non-termination beyond loops has been investigated in [14] and [2]; we note that Example 2 cannot be handled by these techniques.

Here we prove non-looping non-termination on regular languages. The converse, local termination on regular languages, has been investigated in [3]. Regular (tree) automata have been fruitfully applied to a wide rage of properties of term rewriting systems: for proving termination [10, 8, 12], for infinitary normalization [4], for proving liveness [13], and for analysing reachability and deciding the existance of common reducts [9, 5].

## 2 Non-termination and Weakly Closed Languages

▶ **Definition 3.** Let $L \subseteq T(\Sigma, \varnothing)$ a language and $R$ a TRS over $\Sigma$. Then $L$ is called:

- *closed* under rewriting if for every $t \in L$ and $s$ such that $t \to s$, one has $s \in L$, and
- *weakly closed* under rewriting if for every $t \in L$ that is not in normal form, there exists $s \in L$ such that $t \to_R s$.

The following theorem describes the basic idea that we employ for proving non-termination.

▶ **Theorem 4.** *A term rewriting system $R$ over $\Sigma$ is non-terminating if and only if there exists a non-empty language $L \subseteq T(\Sigma, \mathcal{X})$ such that*

*(i) every $t \in L$ contains a redex (that is, $t \to s$ for some term $s$), and*
*(ii) $L$ is weakly closed under rewriting.* ◀

A language fulfilling the properties of Theorem 4 is also called a *recurrence set*, see [1].

To automate this method, we need to restrict to a certain family of languages. In this paper, we consider regular tree languages. To guarantee that the language of a tree automaton is weakly closed under rewriting, we check that the language is not empty and that the automaton is a quasi-model (see Definition 13) for the rewrite system. The latter condition is actually too strict; it implies that the languages is not only weakly closed, but also closed under rewriting. In future, we plan to relieve this restriction.

## 3 Tree Automata

▶ **Definition 5.** A *(nondeterministic finite) tree automaton $A$ over a signature* $\Sigma$ is a tuple $A = \langle Q, \Sigma, F, \delta \rangle$ where

(i) $Q$ is a finite set of *states*,
(ii) $F \subseteq Q$ is a set of *accepting states*, and
(iii) $\{\delta_f\}_{f \in \Sigma}$ is a family of *transition relations* such that for every $f \in \Sigma$:

$$\delta_f \subseteq Q^n \times Q$$

where $n$ is the arity of $f$.

In examples, we often write the transition relation $\delta_f$ as $\to_f$.

▶ **Example 6.** The following is a tree automaton for the signature in Example 1. We consider string rewriting systems as term rewriting systems by interpreting all symbols as unary and adding a special constant $\varepsilon$ to denote the end of the word. Let $A_{LR} = \langle Q, \Sigma, F, \to \rangle$ where $Q = \{0, 1, 2, 3\}$, $\Sigma = \{b, L, R, 0, \varepsilon\}$, $F = \{3\}$ and

$$\to_\varepsilon 0 \qquad 1 \to_z 1 \qquad 0 \to_b 1 \qquad 1 \to_R 2 \qquad 1 \to_L 2$$
$$2 \to_z 2 \qquad 2 \to_b 3$$

The transition relation for $\varepsilon$ can be thought of as defining the initial states (here 0) of a word automaton.

▶ **Example 7.** The following is a tree automaton for Example 2. Let $A_S = \langle Q, \Sigma, F, \to \rangle$ where $Q = \{0,1,2,3,4\}$, $\Sigma = \{ap, S\}$, $F = \{4\}$ and

$$\begin{array}{llll} \to_S 0 & (0,0) \to_{ap} 1 & (1,0) \to_{ap} 2 & (2,2) \to_{ap} 3 \quad (3,3) \to_{ap} 3 \\ & (0,2) \to_{ap} 2 & & (2,3) \to_{ap} 3 \quad (3,3) \to_{ap} 4 \\ & (0,3) \to_{ap} 2 & & \end{array}$$

In Example 12 we show that this automaton accepts the term $SSS(SSS)(SSS(SSS))$.

▶ **Definition 8.** Let $A = \langle Q, \Sigma, F, \delta \rangle$ be a tree automaton over $\Sigma$. For terms $t \in T(\Sigma, \mathcal{X})$ and assignments $\alpha : \mathcal{X} \to \mathcal{P}(Q)$ we define the *interpretation* $[t, \alpha]_A$ by:

$$\begin{aligned} [x, \alpha]_A &= \alpha(x) \\ [f(t_1, \ldots, t_n), \alpha]_A &= \{q \mid (q_1, \ldots, q_n) \in [t_1, \alpha]_A \times \ldots \times [t_n, \alpha]_A,\ \langle (q_1, \ldots, q_n), q \rangle \in \delta_f\} \end{aligned}$$

Whenever $A$ is clear from the context, we write $[t, \alpha]$ as shorthand for $[t, \alpha]_A$. For ground terms $t$, the interpretation is independent of $\alpha$, allowing is to write $[t]_A$ or $[t]$ for short.

▶ **Example 9.** We use the automaton $A_S$ from Example 7. Let $\alpha(x) = \{2\}$, then we have:

$$\begin{array}{ll} [S, \alpha] = \{0\} & [ap(S,S), \alpha] = \{1\} \quad [ap(ap(S,S),S), \alpha] = \{2\} \\ [ap(x,x), \alpha] = \{3\} & [ap(ap(x,x), ap(x,x)), \alpha] = \{3,4\} \end{array}$$

▶ **Definition 10.** Let $A = \langle Q, \Sigma, F, \delta \rangle$ be a tree automaton over $\Sigma$. The *language* $\mathcal{L}(A)$ *accepted by* $A$ is the set $\mathcal{L}(A) = \{t \mid t \in T(\Sigma, \varnothing),\ [t]_A \cap F \neq \varnothing\}$ of ground terms.

▶ **Example 11.** The automaton in Example 6 accepts all words of the form $b\ z^*\ (L|R)\ z^*\ b$, that is, all words that start with $b$, end with $b$, contain one $L$ or $R$ and otherwise only $z$.

▶ **Example 12.** We continue Example 9:

$$\begin{array}{ll} [ap(ap(S,S),S)] = \{2\} & [ap(\ ap(ap(S,S),S)\ ,\ ap(ap(S,S),S)\ )] = \{3\} \\ [ap(\ ap(ap(ap(S,S),S), ap(ap(S,S),S))\ ,\ ap(ap(ap(S,S),S), ap(ap(S,S),S))\ )] = \{3,4\} & \end{array}$$

Thus $F \cap [SSS(SSS)(SSS(SSS))] = \{4\} \neq \varnothing$ and hence the term is accepted by the automaton.

## 4 Closure under Rewriting

▶ **Definition 13.** A tree automaton $A = \langle Q, \Sigma, F, \delta \rangle$ is a *quasi-model* for a term rewriting system $R$ over $\Sigma$ if $[\ell, \alpha]_A \subseteq [r, \alpha]_A$ for every $\ell \to r \in R$ and $\alpha : \mathcal{X} \to \mathcal{P}(Q)$.

Actually, it suffices to check the property $[\ell, \alpha]_A \subseteq [r, \alpha]_A$ for assignments $\alpha : \mathcal{X} \to \mathcal{P}(Q)$ that map variables to singleton sets.

▶ **Lemma 14.** *A tree automaton $A = \langle Q, \Sigma, F, \delta \rangle$ is a* quasi-model *for a term rewriting system $R$ over $\Sigma$ iff $[\ell, \alpha]_A \subseteq [r, \alpha]_A$ for every $\ell \to r \in R$ and $\alpha : \mathcal{X} \to \{\{q\} \mid q \in Q\}$.*

▶ **Example 15.** It is not difficult to check that the automaton $A_{LR}$ from Example 6 is a quasi-model for rewrite system in Example 1.

▶ **Example 16.** We consider the automaton $A_S$ from Example 7. We write $(a,b,c) \to d$ if $d \in [\ell, \alpha]$ when $\alpha(x) = \{a\}$, $\alpha(y) = \{b\}$, $\alpha(z) = \{c\}$. Then for $[\ell, \alpha]$ we have:

$$\begin{array}{lllll} (0,0,2) \to 1 & (2,2,3) \to 3 & (2,3,3) \to 3 & (3,2,3) \to 3 & (3,3,3) \to 3 \\ (0,0,3) \to 1 & (2,2,3) \to 4 & (2,3,3) \to 4 & (3,2,3) \to 4 & (3,3,3) \to 4 \end{array}$$

The interpretation $[r, \alpha]$ has all the above and additionally:

$(0,2,2) \to 3$ $\quad$ $(1,1,0) \to 3$ $\quad$ $(2,2,2) \to 3$
$(0,2,3) \to 3$ $\quad$ $(2,2,2) \to 4$
$(0,3,3) \to 3$

As a consequence $A_S$ is a quasi-model for the $S$-rule.

The following theorem is immediate:

▶ **Theorem 17.** *Let $A = \langle Q, \Sigma, F, \delta \rangle$ be a tree automaton and $R$ a term rewriting system over $\Sigma$. If $A$ is a quasi-model for $R$ then the language of $A$ is closed under rewriting.* ◀

## 5 Ensuring Redex Occurrences

Next, we want to guarantee that every term in the language $\mathcal{L}(A)$ of an automaton $A$ contains a redex with respect to the term rewriting system $R$. For left-linear systems $R$, this problem can be reduced to deciding the inclusion of regular languages.

Let $R$ be a left-linear term rewriting system. Then the set of ground terms containing a redex is a regular tree language. A deterministic automaton $B$ for this language can be constructed using the overlap-closure of subterms of left-hand sides, see further [6, 7].

▶ **Example 18.** The following tree automaton $C = \langle Q, \Sigma, F, \to \rangle$ accepts the language of ground terms that contain a redex occurrence with respect to the $S$-rule. Here $Q = \{0,1,2,3\}$, $\Sigma = \{ap, S\}$, $F = \{3\}$ and

$$\to_S 0 \qquad (0,q) \to_{ap} 1 \qquad (1,q) \to_{ap} 2 \qquad (2,q) \to_{ap} 3 \qquad (3,q) \to_{ap} 3 \qquad (q,3) \to_{ap} 3$$

for all $q \in \{0,1,2\}$.

As a consequence the problem of checking whether every term in $\mathcal{L}(A)$ contains a redex boils down to checking that $\mathcal{L}(A) \subseteq \mathcal{L}(B)$. For non-deterministic $A$ and deterministic $B$, this property can be decided by constructing the product automaton and considering the reachable states.

▶ **Definition 19.** The *product* $A \cdot B$ of tree automata $A = \langle Q, \Sigma, F, \delta \rangle$ and $B = \langle Q', \Sigma, F', \delta' \rangle$ is the automaton $C = \langle Q \times Q', \Sigma, \varnothing, \gamma \rangle$ where for every $f \in \Sigma$ of arity $n$, we define the transition relation $\gamma_f \subseteq (Q \times Q')^n \times (Q \times Q')$ by

$$\langle (q_1,p_1), \ldots, (q_n,p_n) \rangle \; \gamma \; (q',p') \iff \langle q_1, \ldots, q_n \rangle \; \delta_f \; q' \; \wedge \; \langle p_1, \ldots, p_n \rangle \; \delta'_f \; p'$$

▶ **Definition 20.** The set of *reachable states* of a tree automaton $A = \langle Q, \Sigma, F, \delta \rangle$ is the smallest set $S \subseteq Q$ such that $q \in S$ whenever $\langle q_1, \ldots, q_n \rangle \; \delta_f \; q$ for some $q_1, \ldots, q_n \in S$ and $f \in \Sigma$ with arity $n$.

The following theorem gives a method for checking $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ without the need for determinising $A$ (only $B$ needs to be deterministic).

▶ **Theorem 21.** *Let $A = \langle Q, \Sigma, F, \delta \rangle$ and $B = \langle Q', \Sigma, F', \delta' \rangle$ be tree automata such that $B$ is deterministic. Let $S$ be the set of reachable states of the product automaton $A \cdot B$. Then $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ if and only if for all $(q,p) \in S$ it holds that $q \in F \implies p \in F'$.*

▶ **Example 22.** The reachable states of product automaton $A_S \cdot C$ of the automata $A_S$ from Example 7 and $C$ from Example 18 are $(0,0), (1,1), (2,2), (2,1), (3,3), (3,2), (2,3), (4,3)$. The only state $(q,q')$ such that $q$ is accepting in $A_S$ is $(4,3)$ and 3 is an accepting state of $C$. Thus the conditions of Theorem 21 are fulfilled and hence $\mathcal{L}(A_S) \subseteq \mathcal{L}(C)$. Thus every term accepted by $A_S$ contains a redex.

## 6 Future Work

We plan to investigate whether the method described in this paper can be fruitfully extended from regular automata to pushdown automata, that is, context-free languages. For this purpose, it is important that it is decidable whether a context-free language is a subset of a regular language (the language of terms containing left-linear redex occurrences). However, it remains to be investigated whether context-free certificates can be found efficiently.

## References


1 B. Cook. Priciples of program termination. `http://research.microsoft.com/en-us/um/cambridge/projects/terminator/principles.pdf`.

2 F. Emmes, T. Enger, and J. Giesl. Proving Non-looping Non-termination Automatically. In *International Joint Conference on Automated Reasoning (IJCAR 2012)*, volume 7364 of *Lecture Notes in Computer Science*, pages 225–240. Springer, 2012.

3 J. Endrullis, R.C. de Vrijer, and J. Waldmann. Local Termination: Theory and Practice. *Logical Methods in Computer Science*, 6(3), 2010.

4 J. Endrullis, C. Grabmayer, D. Hendriks, J.W. Klop, and R.C. de Vrijer. Proving Infinitary Normalization. In *Postproc. Int. Workshop on Types for Proofs and Programs (TYPES 2008)*, volume 5497 of *Lecture Notes in Computer Science*, pages 64–82. Springer, 2009.

5 J. Endrullis, C. Grabmayer, J.W. Klop, and V. van Oostrom. On Equal $\mu$-Terms. *Theoretical Computer Science*, 412(28):3175–3202, 2011.

6 J. Endrullis and D. Hendriks. Transforming Outermost into Context-Sensitive Rewriting. *Logical Methods in Computer Science*, 6(2), 2010.

7 J. Endrullis and D. Hendriks. Lazy Productivity via Termination. *Theoretical Computer Science*, 412(28):3203–3225, 2011.

8 J. Endrullis, D. Hofbauer, and J. Waldmann. Decomposing Terminating Rewrite Relations. In *Proc. Workshop on Termination (WST '06)*, pages 39–43, 2006.

9 B. Felgenhauer and R. Thiemann. Reachability Analysis with State-Compatible Automata. In *Proc. Conf. on Language and Automata Theory and Applications (LATA 2014)*, volume 8370 of *Lecture Notes in Computer Science*, pages 347–359. Springer, 2014.

10 A. Geser, D. Hofbauer, J. Waldmann, and H. Zantema. On tree automata that certify termination of left-linear term rewriting systems. *Inf. Comput.*, 205(4):512–534, 2007.

11 A. Geser and H. Zantema. Non-looping string rewriting. *ITA*, 33(3):279–302, 1999.

12 M. Korp and A. Middeldorp. Match-bounds revisited. *Inf. Comput.*, 207(11):1259–1283, 2009.

13 M. Mousazadeh, B. T. Ladani, and H. Zantema. Liveness verification in trss using tree automata and termination analysis. *Computing and Informatics*, 29(3):407–426, 2010.

14 M. Oppelt. Automatische Erkennung von Ableitungsmustern in nichtterminierenden Wortersetzungssystemen. Technical report, HTWK Leipzig, Germany, 2008. Diploma Thesis.

15 J. Waldmann. The Combinator S. *Information Computation*, 159(1–2):2–21, 2000.

16 Johannes Waldmann. Compressed loops (draft).

17 H. Zankl and A. Middeldorp. Nontermination of String Rewriting using SAT. 2007.

18 Harald Zankl, Christian Sternagel, Dieter Hofbauer, and Aart Middeldorp. Finding and certifying loops. In *Proc. Conf. on Theory and Practice of Computer Science (SOFSEM 2010)*, volume 5901 of *Lecture Notes in Computer Science*, pages 755–766. Springer, 2010.

# A Solution to Endrullis-08 and Similar Problems*

Alfons Geser

**Fakultät Elektrotechnik und Informationstechnik**
**HTWK Leipzig, Germany**
alfons.geser@htwk-leipzig.de

**Abstract**

We prove the termination of the string rewriting system Endrullis-08, the termination of infinitely many related systems, and even the termination of their union, an infinite string rewriting system.

**1998 ACM Subject Classification** F.4.2 Grammars and Other Rewriting Systems

**Keywords and phrases** string rewriting, semi-Thue system, uniform termination, termination, loop

**Digital Object Identifier** 10.4230/LIPIcs.xxx.yyy.p

## 1 Endrullis's Problem

Jörg Endrullis raised the question whether the two-rule string rewriting system (SRS) $\{aabb \to bbba, ba \to aaaa\}$ terminates. This SRS was included in the Termination Competition 2006 as `SRS/Endrullis/08.srs` and was later renamed to `tpdb-8.0/TRS/Mixed_SRS/08.xml`. Chris Sternagel and René Thiemann have in 2011 succeeded in proving the termination of Endrullis-08.[1] They use the dependency pair framework with arctic matrices of dimension 3 and semantic labelling. A similar proof was done by AProVE in December of 2013.[2] This proof uses the dependeny pair framework with semantic labelling. In this contribution, we give a different proof. In order to simplify the presentation, let us switch to the reflected image $\{bbaa \to abbb, ab \to aaaa\}$ of Endrullis's SRS.

## 2 The Termination Proof

Consider systems $\{bbaa \to abbb, ab \to a^i\}$ for various $i \in \mathbb{N}_0$. If $i \leq 2$ then we have termination, as shown by a length-then-lexicographic argument where $b > a$. If $i \in \{3, 5\}$ then there is a loop of length 9 starting from $b^6a^2$. If $i \geq 7$ is odd then there is a loop of length $\frac{11+i}{2}$ starting from $b^6a^2$. This leaves the case where $i \geq 4$ is even. It turns out that the proof can be done uniformly for all even $i \geq 4$. Not only can we prove that each $\{bbaa \to abbb, ab \to a^i\}$ terminates, but also that their union terminates:

▶ **Theorem 1.** *The infinite SRS* $\{bbaa \to abbb\} \cup \{ab \to a^i \mid i \geq 2 \text{ is even}\}$ *terminates.*

▶ **Corollary 2.** *The SRS* $\{bbaa \to abbb, ab \to a^i\}$ *terminates if, and only if,* $i = 1$ *or* $i$ *is even.*

In the remainder of this contribution, we prove Theorem 1 by reducing the uniform termination problem of $E := \{bbaa \to abbb\} \cup \{ab \to a^i \mid i \geq 2 \text{ is even}\}$ in six steps (Lemmas 3, ..., 8).

* Partially supported by Fakultät Informatik, Mathematik und Naturwissenschaften, HTWK Leipzig, Germany.

[1] `http://cl-informatik.uibk.ac.at/software/ceta/experiments/semlab/with_semlab/___Users___rene___tpdb-8.0___TRS___Mixed_SRS___08xml/proof.xml.xml`

[2] `http://termcomp.uibk.ac.at/termcomp/competition/resultDetail.seam?resultId=480566&cid=68`

### First Step: Interreduction

Interreduction means rewriting the right hand sides of a rewrite system. The set of descendants of the string $ab^3$ by the SRS $R := \{ab \to a^i \mid i \geq 2 \text{ is even}\}$ is $\{ab^3, a^ib^2, a^{i+1}b, a^i \mid i \geq 0 \text{ even}\}$. Define the SRS $F$ by

$$F := \{b^2a^2 \to r \mid r \in \{ab^3, a^ib^2, a^{i+1}b, a^i\}, i \geq 0 \text{ even}\} \ .$$

▶ **Lemma 3.** *If $F$ terminates then $E$ terminates.*

**Proof.** We have $\to_F\to_R \subseteq \to_F \cup \to_R\to_F$, where the non-overlapping case commutes, and where the overlapping case either degenerates to a rewrite step on the contractum, e.g. for even $i \geq 0$,

$$b^2a^2 \to_F ab^3 \to_R a^ib^2 \text{ for which } b^2a^2 \to_F a^ib^2,$$

or it commutes: for even $i \geq 2$ and even $j \geq 0$,

$$b^2a^2b \to_F a^ib \to_R a^{i+j-1} \text{ for which } b^2a^2b \to_R b^2a^{j+1} \to_F a^{i+j-1} \ .$$

By a result of Doornbos and von Karger [2], if both $R$ and $F$ terminate and $\to_F\to_R \subseteq \to_R(\to_F \cup \to_R)^* \cup \to_F$, then $R \cup F$ terminates. By construction, $\to_E \subseteq \to_R \cup \to_F$, so if $F$ terminates then $E$ terminates. ◀

### Second Step: Encompassment

We encompass each rule of $F$ in a left context of the form $ba^m, m \in \mathbb{N}_0$ and a right context of the form $b^na, n \in \mathbb{N}_0$. This yields the SRS

$$G := \{ba^mb^2a^2b^na \to r \mid r \in \{ba^{m+1}b^{n+3}a, ba^{m+i}b^{n+2}a, ba^{m+i+1}b^{n+1}a, ba^{m+i}b^na\}, \\ i \geq 2 \text{ is even}, m, n \in \mathbb{N}_0\} \ .$$

▶ **Lemma 4.** *If $G$ terminates then $F$ terminates.*

**Proof.** For all $s \in \{a, b\}^*$, we have: $G$ admits a non-terminating derivation starting from $bsa$ if, and only if, $F$ admits a non-terminating derivation starting from $s$. ◀

### Third Step: Semantic Labelling

Let the interpretation $[s] : \mathcal{D} \to \mathcal{D}$ of a string $s$ on the domain $\mathcal{D} = \{0, 1\} \times \{0, 1, 2\}$ be defined by

$$[a](x, y) = \begin{cases} (x, 2), & \text{if } y = 1, \\ (x, 1), & \text{else;} \end{cases} \qquad [b](x) = \begin{cases} (0, 0), & \text{if } x = 1 \text{ and } y = 0, \\ (1, 0), & \text{else.} \end{cases}$$

For all $n > 0$, we get

$$[b^na](x, y) = \begin{cases} (0, 0), & \text{if } n \text{ is even,} \\ (1, 0), & \text{else.} \end{cases}$$

For all $m > 0, n > 0$ we get $[a^mb^na](x, y) = (x', y')$ where

$$x' = \begin{cases} 0, & \text{if } n \text{ is even,} \\ 1, & \text{else} \end{cases} \quad \text{and} \quad y' = \begin{cases} 2, & \text{if } m \text{ is even,} \\ 1, & \text{else.} \end{cases}$$

Case 1: $m$ odd, $n$ odd: $\ell_1 = b_{01}a^m b b_{12}a^2b^{n-1}b_{xy'}a$, $r_1 = b_{02}a^{m+1}b^{n+2}b_{xy'}a$,
$r'_1 = b_{11}a^{m+i}b^{n+1}b_{xy'}a$, $r''_1 = b_{02}a^{m+i+1}b^n b_{xy'}a$, $r'''_1 = b_{11}a^{m+i}b^{n-1}b_{xy'}a$ .
Case 2: $m = 0$, $n$ odd: $\ell_2 = b^2b_{12}a^2b^{n-1}b_{xy'}a$, $r_2 = b_{01}ab^{n+2}b_{xy'}a$,
$r'_2 = b_{12}a^i b^{n+1}b_{xy'}a$, $r''_2 = b_{01}a^{i+1}b^n b_{xy'}a$, $r'''_2 = b_{12}a^i b^{n-1}b_{xy'}a$ .
Case 3: $m > 0$ even, $n$ odd: $\ell_3 = b_{02}a^m b b_{12}a^2b^{n-1}b_{xy'}a$, $r_3 = b_{01}a^{m+1}b^{n+2}b_{xy'}a$,
$r'_3 = b_{12}a^{m+i}b^{n+1}b_{xy'}a$, $r''_3 = b_{01}a^{m+i+1}b^n b_{xy'}a$, $r'''_3 = b_{12}a^{m+i}b^{n-1}b_{xy'}a$ .
Case 4: $m$ odd, $n = 0$: $\ell_4 = b_{01}a^m b b_{xy'}a^3$, $r_4 = b_{12}a^{m+1}b^2b_{xy'}a$,
$r'_4 = b_{01}a^{m+i}b b_{xy'}a$, $r''_4 = b_{12}a^{m+i+1}b_{xy'}a$, $r'''_4 = b_{xy''}a^{m+i+1}$ .
Case 5: $m$ odd, $n > 0$ even: $\ell_5 = b_{01}a^m b b_{02}a^2b^{n-1}b_{xy'}a$, $r_5 = b_{12}a^{m+1}b^{n+2}b_{xy'}a$,
$r'_5 = b_{01}a^{m+i}b^{n+1}b_{xy'}a$, $r''_5 = b_{12}a^{m+i+1}b^n b_{xy'}a$, $r'''_5 = b_{01}a^{m+i}b^{n-1}b_{xy'}a$ .
Case 6: $m = 0$, $n = 0$: $\ell_6 = b^2b_{xy'}a^3$, $r_6 = b_{11}ab^2b_{xy'}a$,
$r'_6 = b_{02}a^i b b_{xy'}a$, $r''_6 = b_{11}a^{i+1}b_{xy'}a$, $r'''_6 = b_{xy'}a^{i+1}$ .
Case 7: $m = 0$, $n > 0$ even: $\ell_7 = b^2b_{02}a^2b^{n-1}b_{xy'}a$, $r_7 = b_{11}ab^{n+2}b_{xy'}a$,
$r'_7 = b_{02}a^i b^{n+1}b_{xy'}a$, $r''_7 = b_{11}a^{i+1}b^n b_{xy'}a$, $r'''_7 = b_{02}a^i b^{n-1}b_{xy'}a$ .
Case 8: $m > 0$ even, $n = 0$: $\ell_8 = b_{02}a^m b b_{xy'}a^3$, $r_8 = b_{11}a^{m+1}b^2b_{xy'}a$,
$r'_8 = b_{02}a^{m+i}b b_{xy'}a$, $r''_8 = b_{11}a^{m+i+1}b_{xy'}a$, $r'''_8 = b_{xy'}a^{m+i+1}$ .
Case 9: $m > 0$ even, $n > 0$ even: $\ell_9 = b_{02}a^m b b_{02}a^2b^{n-1}b_{xy'}a$, $r_9 = b_{11}a^{m+1}b^{n+2}b_{xy'}a$,
$r'_9 = b_{02}a^{m+i}b^{n+1}b_{xy'}a$, $r''_9 = b_{11}a^{m+i+1}b^n b_{xy'}a$, $r'''_9 = b_{02}a^{m+i}b^{n-1}b_{xy'}a$ .

**Table 1** The constituents of the SRS $H$

The interpretation is a model: Every left hand side and every right hand side have a prefix of the form $b^n a$ where $n$ is odd. Hence $[\ell](x,y) = (1,0) = [r](x,y)$ holds for every $(x,y) \in \mathcal{D}$ and rule $\ell \to r$.

We apply a Semantic Labelling [3] to $G$. We label only $b$ symbols, and we label $b$ by $(x,y)$ only if $y \neq 0$. In effect this means that in each sequence of successive $b$ symbols, only the rightmost $b$ symbol receives a label. The symbol $b$ labelled by $(x,y)$ is denoted $b_{xy}$. Let $H$ denote the resulting labelled SRS.

In order to compute the labels, we perform a case analysis on whether $m = 0$, $m > 0$ is even, or $m$ is odd, and whether $n = 0$, $n > 0$ is even, or $n$ is odd. For each case $c$, we get a family $H_c := \{\ell_c \to r_c, \ell_c \to r'_c, \ell_c \to r''_c, \ell_c \to r'''_c\}$ of labelled rules, indexed by even $i \geq 2$, $m, n \in \mathbb{N}_0$, $x \in \{0,1\}$, $y \in \{0,1,2\}$. Table 1 shows the labelled strings $\ell_c, r_c, r'_c, r''_c, r'''_c$ in the nine cases $c$. We use the abbreviations $(x,y') = [a](x,y)$, $(x,y'') = [a](x,y')$. Note that $y', y'' \in \{1,2\}$ and so that $[a](x,y'') = (x,y')$ and $[b](x,y') = [b](x,y'') = (1,0)$. Note also that $y = 0$ and $y = 2$ yield the same labelled rule because the rightmost symbol in each left and right hand side is $a$.

Zantema's Theorem yields:

▶ **Lemma 5.** *If $H$ terminates then $G$ terminates.*

### Fourth Step: Interpretation

In the remaining steps, we prove termination of the labelled SRS, $H$. We do so incrementally, using the concept of Relative Termination [1]. $R$ is called *terminating relative to* $S$ if there is no infinite $R \cup S$ derivation that contains infinitely many $R$ steps. If $S$ terminates and $R$ terminates relative to $S$ then $R \cup S$ terminates. Termination of $R$ relative to $S$ can be proven by a quasiorder $\gtrsim$ such that $\to_R \subseteq >$ and $\to_S \subseteq \sim$.

For the fourth step let $H' := H_1 \cup \{\ell_4 \to r_4''' \mid (x, y') \in \{(1,2),(0,1)\}\}$.

▶ **Lemma 6.** *$H'$ terminates relative to $H \setminus H'$.*

**Proof.** We use an interpretation that counts the number of $b_{12}$ and $b_{01}$ symbols. All rules in $H'$ decrease this number by 2, and all rules in $H \setminus H'$ keep it constant. ◀

### Fifth Step: Tuple Representation

Termination of $H \setminus H'$ remains to be proved. The rules in $H \setminus H'$ keep the number of $b_{12}$ and $b_{01}$ symbols constant. This suggests one to apply some lexicographic order on a tuple representation of the rules, where the symbols $b_{12}$ and $b_{01}$ act as separators.

Instead of the tuple representation, we prefer a modified tuple representation: The Modified Tuple Representation $T_f(s)$ of $s \in \Sigma^*$ through the interpretation $f$ w.r.t. a non-empty set $\Delta \subseteq \Sigma$ of separator symbols is defined by

$$T_f(s) = (f(s_0t_0), f(s_1t_1), \ldots, f(s_{k-1}t_{k-1}), f(s_k)),$$

provided that $k \geq 0$, $s = s_0t_0s_1t_1 \ldots s_{k-1}t_{k-1}s_k$, $s_i \in (\Sigma \setminus \Delta)^*$ for all $i \in \{0, \ldots, k\}$, and $t_i \in \Delta$ for all $i \in \{0, \ldots, k-1\}$. The point of this modification is to keep the separators $t_i$ accessible to the interpretation $f$.

For the fifth step let

$$\begin{aligned} H'' := H_2 \cup H_3 \cup \{\ell_4 \to r_4''' \mid (x, y') \in \{(1,1),(0,2)\}\} \cup \\ \{\ell_4 \to r_4, \ell_4 \to r_4'', \ell_5 \to r_5 \mid (x, y') \in \mathcal{D}\} \cup \\ \{\ell_c \to r_c'', \ell_c \to r_c''' \mid 5 \leq c \leq 9, (x, y') \in \mathcal{D}\} \ . \end{aligned}$$

▶ **Lemma 7.** *$H''$ terminates relative to $H \setminus (H' \cup H'')$.*

**Proof.** For our purposes, we define the interpretation $f$ by $f(s) = |s|_{b_*} - |s|_{b_{11}} + |s|_{b_{01}}$. Here $|s|_{b_*}$ means the number of $b$ symbols, without a label or with whatever label, in the string $s$. Note that $|s|_{b_*} - |s|_{b_{11}} \geq 0$ holds, whence $f(s) \in \mathbb{N}_0$, for all strings $s$.

We compare the Modified Tuple Representations through $f$ lexicographically from left to right, denoted by $>_{\text{lex}}$. For instance, rule $\ell_4 \to r_4'''$ in the case $x = 1, y' = 1$ satisfies

$$T_f(\ell_4) = (f(b_{01}), f(a^m b b_{11} a^3)) = (2,1) >_{\text{lex}} (1,0) = (f(b_{12}), f(a^{m+i+1})) = T_f(r_4''') \ .$$

For rule $\ell_4 \to r_4'''$ in the case $x = 0, y' = 2$ we get

$$T_f(\ell_4) = (f(b_{01}), f(a^m b b_{02} a^3)) = (2,2) >_{\text{lex}} (2,0) = (f(b_{01}), f(a^{m+i+1})) = T_f(r_4''') \ .$$

Thus we have solved the two remaining instances of rule $\ell_4 \to r_4'''$.

The remaining rules have a suffix in $b_{xy'}a^+$ both at the left hand side and at the right hand side. For the remainder of this proof, we can simplify our argument by dropping this suffix since our order is closed under right contexts and is insensitive to the number of $a$ symbols.

| Case $c$ | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| $T_f(\ell_c)$ | $(3, n-1)$ | $(3, n-1)$ | $(2, 1)$ | $(2, n+1)$ | $(2)$ | $(n+2)$ | $(2)$ | $(n+2)$ |
| $T_f(r_c)$ | $(2, n+2)$ | $(2, n+2)$ | $(1, 2)$ | $(1, n+2)$ | $\boxed{(2)}$ | $\boxed{(n+2)}$ | $\boxed{(2)}$ | $\boxed{(n+2)}$ |
| $T_f(r'_c)$ | $(1, n+1)$ | $(1, n+1)$ | $\boxed{(2, 1)}$ | $\boxed{(2, n+1)}$ | $\boxed{(2)}$ | $\boxed{(n+2)}$ | $\boxed{(2)}$ | $\boxed{(n+2)}$ |
| $T_f(r''_c)$ | $(2, n)$ | $(2, n)$ | $(1, 0)$ | $(1, n)$ | $(0)$ | $(n)$ | $(0)$ | $(n)$ |
| $T_f(r'''_c)$ | $(1, n-1)$ | $(1, n-1)$ | - | $(2, n-1)$ | $(0)$ | $(n)$ | $(0)$ | $(n)$ |

**Table 2** The Modified Tuple Representations through $f$

Table 2 summarizes the $T_f(s)$ for the left hand sides and the various corresponding right hand sides of the rewrite rules. Each right hand side tuple that equals its left hand side tuple is boxed. Check that a right hand side tuple that is not boxed is less w.r.t. $>_{\text{lex}}$ than its left hand side tuple. ◀

### Sixth Step: Recursive Path Order

▶ **Lemma 8.** *$H \setminus (H' \cup H'')$ terminates.*

**Proof.** The SRS $H \setminus (H' \cup H'')$ is ordered by a Recursive Path Order, where all symbols in $\{b, b_{01}, b_{02}, b_{12}\}$ are equivalent in precedence, and each symbol from this set is greater in precedence than any of $b_{11}$ and $a$. ◀

**Acknowledgements** I wish to thank Johannes Waldmann for presenting this challenge to me and for proof-reading the manuscript. Carsten Fuhs kindly pointed to Sternagel and Thiemann's proof.

## References

1 Leo Bachmair and Nachum Dershowitz. Commutation, transformation, and termination. In Jörg Siekmann, editor, *Proc. 8th Int. Conf. Automated Deduction*, LNCS 230, pages 5–20. Springer, July 1986.

2 Henk Doornbos and Burghard von Karger. On the union of well-founded relations. *Logic Journal of the IGPL*, 6(2):195–201, 1998.

3 Hans Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995.

# Kurth's Criterion H Revisited*


## Alfons Geser

**Fakultät Elektrotechnik und Informationstechnik**
**HTWK Leipzig, Germany**
alfons.geser@htwk-leipzig.de



**Abstract**

Criterion H is one of the criteria for termination of string rewriting that Winfried Kurth implemented in his thesis. Criterion H can be slightly improved. The improved version can be translated into a termination proof by Semantic Labelling.



**1998 ACM Subject Classification** F.4.2 Grammars and Other Rewriting Systems

**Keywords and phrases** semi-Thue system, uniform termination, termination, Criterion H, semantic labelling

**Digital Object Identifier** 10.4230/LIPIcs.xxx.yyy.p


## 1 Criterion H

In his dissertation [4], Winfried Kurth implemented a suite of criteria for the termination of single-rule string rewriting systems (SRSs). In the meantime, each of his Criteria A to G is covered by new termination criteria [5, 6, 7, 1]. Senizergues [9], Kobayashi et al. [3, 10], Moczydlowski and Geser [8], and Hofbauer, Waldmann, and Zantema [2, 12] have introduced further criteria. Criterion H has not yet been covered. We give a criterion that is slightly more powerful than Kurth's Criterion H and show that it can be translated into a proof by Semantic Labelling.

Let $p(k) := \max\{0, k-1\}$ denote the total predecessor on $\mathbb{N}_0$.

▶ **Definition 1** ([4, Definition 4.39]). Functions $\mu : \mathbb{N}_0^* \to \mathbb{Z}^*$ and $\hat{\mu} : \mathbb{N}_0^* \to \mathbb{N}_0^*$ are defined by

$$\mu((k_1, \ldots, k_n)) = (k_1 - 1, k_1 + k_2 - 2, k_1 + k_2 + k_3 - 3, \ldots, k_1 + \cdots + k_n - n)$$

and by $\hat{\mu}((k_1, \ldots, k_n)) = (m_1, m_2, \ldots, m_n)$ where the $m_i$ are defined recursively by $m_1 = p(k_1)$ and $m_i = p(m_{i-1} + k_i)$ for all $2 \leq i \leq n$.

If $\mu(\tau) \in \mathbb{N}_0^*$ then $\hat{\mu}(\tau) = \mu(\tau)$.

Let the alphabet $\{a, b\}$ be given. Let $|s|_c$ denote the number of occurrences of the letter $c \in \{a, b\}$ in the string $s$. The tuple representation $T(w)$ of a string $w$ is defined by $T(w) = (k_0, k_1, k_2, \ldots, k_n)$ if $w = a^{k_0} b a^{k_1} b a^{k_2} \ldots b a^{k_n}$, $k_0, \ldots, k_n \in \mathbb{N}_0$.

For $\tau, \tau' \in \mathbb{N}_0^*$, let $\tau >_{\text{mult}} \tau'$ denote the multiset extension of the order on $\mathbb{N}_0$, applied to the multiset of $\tau$ and the multiset of $\tau'$. Let $\geq_{\text{mult}}$ denote $>_{\text{mult}} \cup =$.

▶ **Theorem 2** (Criterion H [4, Satz 4.40]). *The SRS $(R, \{a, b\})$ terminates if the following conditions hold for every rule $(\ell \to r) \in R$:*

1. $|r|_a > 0$ *and* $|r|_b > 0$;
2. $|\ell|_a - |\ell|_b \geq |r|_a - |r|_b$;


* Partially supported by Fakultät Informatik, Mathematik und Naturwissenschaften, HTWK Leipzig, Germany.

3. $\mu(T(\ell)) \in \mathbb{N}_0^*$ *and* $\mu(T(r)) \in \mathbb{N}_0^*$;
4. $\mu(T(\ell)) >_{\text{mult}} \mu(T(r))$.

For instance, the SRS $aaba \to abaab$ satisfies Criterion H and hence terminates [4, Satz 4.44]. Kurth states Theorem 2 only for the single-rule case, but his proof works in the general case: he shows $\hat{\mu}(T(s\ell t)) >_{\text{mult}} \hat{\mu}(T(srt))$ for all $s, t \in \{a, b\}^*$.

For $s = a_1 a_2 \dots a_n$ let $\bar{s} = a_n \dots a_2 a_1$. For an SRS $R$, let $\bar{R} = \{\bar{\ell} \to \bar{r} \mid (\ell \to r) \in R\}$. We refer to the map $s \mapsto \bar{s}$ as string reflection. Given an SRS $(R, \Sigma)$ where $\Sigma$ may differ from $\{a, b\}$, and two non-empty sets $X, Y \subseteq \Sigma$ such that $X \cap Y = \varnothing$, let the projection $h : \Sigma^* \to \{a, b\}^*$ be defined by $h(c) = a$ if $c \in X$, $h(c) = b$ if $c \in Y$, and $h(c) = \varepsilon$ if $c \in \Sigma \setminus (X \cup Y)$. Let $h(R) = \{h(\ell) \to h(r) \mid (\ell \to r) \in R\}$. Kurth uses string reflection and projection as preprocessing steps in combination with Criterion H. For the purpose of a simple presentation, we drop these preprocessing steps for the time being.

## 2 Technical Stuff

We give a few definitions and basic results in order to state a stronger version of Theorem 2.

▶ **Definition 3.** For each $x \in \mathbb{N}_0$, let $\hat{\mu}_x : \mathbb{N}_0^* \to \mathbb{N}_0^*$ be defined by $\hat{\mu}_x((k_1, \dots, k_n)) = (m_1, m_2, \dots, m_n)$ where the $m_i$ are defined recursively by $m_1 = p(x + k_1)$ and $m_i = p(m_{i-1} + k_i)$ for all $2 \le i \le n$.

By definition, $\hat{\mu}_0 = \hat{\mu}$.

Let $\mathsf{last}((k_1, \dots, k_n)) = k_n$. For $n \in \mathbb{N}_0$, $p_1, \dots, p_n, x \in \mathbb{Z}$ let $(p_1, \dots, p_n) + x$ denote $(p_1 + x, \dots, p_n + x)$. Let $\mathsf{xcrit}(\tau) := \max\{0, -m_1, -m_2, \dots, -m_n\}$ where $(m_1, m_2, \dots, m_n) = \mu(\tau)$. By definition, $\mathsf{xcrit}(\tau)$ is the least $x \in \mathbb{N}_0$ such that $\mu(\tau) + x \in \mathbb{N}_0^*$. Let $\geq^*$ denote the pointwise order on $\mathbb{Z}^*$, i.e. $(p_1, \dots, p_m) \geq^* (q_1, \dots, q_n)$ iff $m = n$ and $p_i \ge q_i$ for all $1 \le i \le n$.

▶ **Lemma 4.** *For all* $\sigma, \tau \in \mathbb{N}_0^*$ *and* $x, y \in \mathbb{N}_0$*:*

1. *if* $y \ge x$ *then* $\hat{\mu}_y(\tau) \geq^* \hat{\mu}_x(\tau)$;
2. *if* $y \ge x$ *then* $\hat{\mu}_x(\tau) + (y - x) \geq^* \hat{\mu}_y(\tau)$;
3. *if* $x \ge \mathsf{xcrit}(\tau)$ *then* $\hat{\mu}_x(\tau) = \mu(\tau) + x$;
4. $\hat{\mu}_x(\tau) \geq^* \mu(\tau) + x$;
5. *if* $\hat{\mu}_{\mathsf{xcrit}(\sigma)}(\sigma) >_{\text{mult}} \hat{\mu}_{\mathsf{xcrit}(\sigma)}(\tau)$ *then* $\hat{\mu}_x(\sigma) >_{\text{mult}} \hat{\mu}_x(\tau)$ *for all* $x \ge \mathsf{xcrit}(\sigma)$;
6. *if* $\mathsf{last}(\hat{\mu}_{\mathsf{xcrit}(\sigma)}(\sigma)) \ge \mathsf{last}(\hat{\mu}_{\mathsf{xcrit}(\sigma)}(\tau))$ *then* $\mathsf{last}(\hat{\mu}_x(\sigma)) \ge \mathsf{last}(\hat{\mu}_x(\tau))$ *for all* $x \ge \mathsf{xcrit}(\sigma)$.

**Proof.** Claim 1: Let $y \ge x$, $\tau = (k_1, \dots, k_n)$, $\hat{\mu}_y(\tau) = (q_1, \dots, q_n)$, $\hat{\mu}_x(\tau) = (p_1, \dots, p_n)$. Then $q_1 = p(y + k_1)$, $q_i = p(q_{i-1} + k_i)$ for all $2 \le i \le n$, $p_1 = p(x + k_1)$, $p_i = p(p_{i-1} + k_i)$ for all $2 \le i \le n$. We show that $q_i \ge p_i$ for all $1 \le i \le n$ by induction on $i$. Case $i = 1$: Then $q_1 = p(y + k_1) \ge p(x + k_1) = p_1$. Case $i > 1$: By inductive hypothesis, $q_{i-1} \ge p_{i-1}$. Then $q_i = p(q_{i-1} + k_i) \ge p(p_{i-1} + k_i) = q_i$.

Claim 2 and Claim 3 are proven in a similar way.

Claim 4: $\hat{\mu}_x(\tau) \geq^* \hat{\mu}_y(\tau) + (x - y) = (\mu(\tau) + y) + (x - y) = \mu(\tau) + x$ where $y := \mathsf{xcrit}(\tau)$.

Claim 5: Let $y = \mathsf{xcrit}(\sigma)$. By Claim 3, the premise, and Claim 2 we get $\hat{\mu}_x(\sigma) = \mu(\sigma) + x = \hat{\mu}_y(\sigma) + (x - y) >_{\text{mult}} \hat{\mu}_y(\tau) + (x - y) \geq^* \hat{\mu}_x(\tau)$. Note here that $\sigma \geq^* \tau$ implies $\sigma \geq_{\text{mult}} \tau$ for all $\sigma, \tau \in \mathbb{N}_0^*$.

Claim 6: By Claim 3, the premise, and Claim 2 we get $\mathsf{last}(\hat{\mu}_x(\sigma)) = \mathsf{last}(\mu(\sigma)) + x = \mathsf{last}(\hat{\mu}_y(\sigma)) + (x - y) \ge \mathsf{last}(\hat{\mu}_y(\tau)) + (x - y) \ge \mathsf{last}(\hat{\mu}_x(\tau))$. ◀

Let & denote concatenation on $\mathbb{N}_0^*$.

▶ **Lemma 5.** *1.* $\hat{\mu}_x(T(as)) = \hat{\mu}_{x+1}(T(s))$ *for all* $s \in \{a,b\}^*$*;*
*2.* $\hat{\mu}_x(T(bs)) = (p(x))\&\hat{\mu}_{p(x)}(T(s))$ *for all* $s \in \{a,b\}^*$*;*
*3. For all* $s \in \{a,b\}^*$*, if* $\hat{\mu}_x(T(s)) = (p_1,\ldots,p_{n-1},p_n)$ *then* $\hat{\mu}_x(T(sa)) = (p_1,\ldots,p_{n-1},p_{n-1}+\mathsf{last}(T(s)))$*;*
*4. For all* $s \in \{a,b\}^*$*, if* $\hat{\mu}_x(T(s)) = (p_1,\ldots,p_n)$ *then* $\hat{\mu}_x(T(sb)) = (p_1,\ldots,p_n,p(p_n))$*.*

**Proof.** Claim 1: Let $T(s) = (k_1,k_2,\ldots,k_n)$, $\hat{\mu}_x(T(as)) = (q_1,\ldots,q_n)$, and $\hat{\mu}_{x+1}(T(s)) = (p_1,\ldots,p_n)$. We have $T(as) = (k_1+1,k_2,\ldots,k_n)$, $q_1 = p(x+k_1+1)$, $q_i = p(q_{i-1}+k_i)$ for $2 \le i \le n$, $p_1 = p(x+k_1+1)$, $p_i = p(p_{i-1}+k_i)$ for $2 \le i \le n$. By induction on $i$, one proves $q_i = p_i$ for all $1 \le i \le n$.

Claims 2, 3, and 4 are proven in a similar way. ◀

## 3 An Improvement

For $s,t \in \{a,b\}^*$ let $s >_G t$ if

$$\mathsf{last}(\hat{\mu}_x(T(sa))) \ge \mathsf{last}(\hat{\mu}_x(T(ta))) \text{ and } \hat{\mu}_x(T(s)) >_{\text{mult}} \hat{\mu}_x(T(t)) \text{ for all } x \in \mathbb{N}_0.$$

Let a reduction order be a well-founded and transitive binary relation on strings that is closed under left and right contexts.

▶ **Theorem 6.** $>_G$ *is a reduction order on* $\{a,b\}^*$*.*

**Proof.** Transitivity of $>_G$ follows from transitivity of both $\ge$ and $>_{\text{mult}}$. Well-foundedness of $>_G$ follows from well-foundedness of $>_{\text{mult}}$. Closure under left and right contexts remains to be proved. To this end, let $s >_G t$, which is $\mathsf{last}(\hat{\mu}_x(T(sa))) \ge \mathsf{last}(\hat{\mu}_x(T(ta)))$ and $\hat{\mu}_x(T(s)) >_{\text{mult}} \hat{\mu}_x(T(t))$ for all $x \in \mathbb{N}_0$. We claim $as >_G at$. We have $\mathsf{last}(\hat{\mu}_x(T(asa))) = \mathsf{last}(\hat{\mu}_{x+1}(T(sa))) \ge \mathsf{last}(\hat{\mu}_{x+1}(T(ta))) = \mathsf{last}(\hat{\mu}_x(T(ata)))$ and $\hat{\mu}_x(T(as)) = \hat{\mu}_{x+1}(T(s)) >_{\text{mult}} \hat{\mu}_{x+1}(T(t)) = \hat{\mu}_x(T(at))$ by Lemma 5(1) and the premise. Next we claim $bs >_G bt$. Then $\mathsf{last}(\hat{\mu}_x(T(bsa))) = \mathsf{last}(\hat{\mu}_{p(x)}(T(sa))) \ge \mathsf{last}(\hat{\mu}_{p(x)}(T(ta))) = \mathsf{last}(\hat{\mu}_x(T(bta)))$. Next, $\hat{\mu}_x(T(bs)) = (p(x))\&\hat{\mu}_{p(x)}(T(s)) >_{\text{mult}} (p(x))\&\hat{\mu}_{p(x)}(T(t)) = \hat{\mu}_x(T(bt))$ by Lemma 5(2) and the premise. For the proof of claim $sa >_G ta$, let $\hat{\mu}_x(T(sa)) = (p_1,\ldots,p_m)$, let $\hat{\mu}_x(T(ta)) = (q_1,\ldots,q_n)$, and let $k = \mathsf{last}(T(sa)) \ge 1$ and $k' = \mathsf{last}(T(ta)) \ge 1$. We have $\mathsf{last}(\hat{\mu}_x(T(sa))) = p_m + 1 \ge q_n + 1 = \mathsf{last}(\hat{\mu}_x(T(ta)))$, by Lemma 5(3) and the premise. By premise, $\hat{\mu}_x(T(s)) = (p_1,\ldots,p_{m-1},p_{m-1}+k-1) >_{\text{mult}} (q_1,\ldots,q_{n-1},q_{n-1}+k'-1) = \hat{\mu}_x(T(t))$. With that, $\hat{\mu}_x(T(sa)) = (p_1,\ldots,p_{m-1},p_{m-1}+k) >_{\text{mult}} (q_1,\ldots,q_{n-1},q_{n-1}+k') = \hat{\mu}_x(T(ta))$. Finally, we claim $sb >_G tb$. Let $\hat{\mu}_{x+1}(T(sa)) = (p_1,\ldots,p_m)$ and $\hat{\mu}_x(T(ta)) = (q_1,\ldots,q_n)$. Then $\mathsf{last}(\hat{\mu}_x(T(sba))) = p(p(p_m)) + 1 \ge p(p(q_n)) + 1 = \mathsf{last}(\hat{\mu}_x(T(tba)))$ and $\hat{\mu}_x(T(sb)) = (p_1,\ldots,p_m,p(p(p_m))) >_{\text{mult}} (q_1,\ldots,q_n,p(p(p_n))) \ge_{\text{mult}} (q_1,\ldots,q_n,p(p(q_n))) = \hat{\mu}_x(T(tb))$. ◀

Lemmas 4(4), 4(5) and Theorem 6 prove:

▶ **Theorem 7.** *The SRS* $(R,\{a,b\})$ *terminates if the following conditions hold for every rule* $(\ell \to r) \in R$*:*
*1.* $\mathsf{last}(\hat{\mu}_x(T(\ell a))) \ge \mathsf{last}(\hat{\mu}_x(T(ra)))$ *for all* $x \le \mathsf{xcrit}(T(\ell))$*;*
*2.* $\hat{\mu}_x(T(\ell)) >_{\text{mult}} \hat{\mu}_x(T(r))$ *for all* $x \le \mathsf{xcrit}(T(\ell))$*.*

▶ **Theorem 8.** *The premises of Theorem 7 hold whenever the premises of Theorem 2 hold.*

**Proof.** Let $(\ell \to r) \in R$ where $\ell, r \in \{a,b\}^*$, and let (1) $|r|_a > 0$ and $|r|_b > 0$; (2) $|\ell|_X - |\ell|_Y \geq |r|_X - |r|_Y$; (3) $\mu(T(\ell)) \in \mathbb{N}_0^*$ and $\mu(T(r)) \in \mathbb{N}_0^*$; (4) $\mu(T(\ell)) >_{\text{mult}} \mu(T(r))$. By (3) we have $\mathsf{xcrit}(T(\ell a)) = 0$. By (2) then, $\mathsf{last}(\hat{\mu}_0(T(\ell a))) = \mathsf{last}(\mu(T(\ell a))) = |\ell|_a - |\ell|_b \geq |r|_a - |r|_b = \mathsf{last}(\mu(T(ra))) = \mathsf{last}(\hat{\mu}_0(T(ra)))$ which proves Premise 1. By (4), $\hat{\mu}_0(T(\ell)) = \mu(T(\ell)) >_{\text{mult}} \mu(T(r)) = \hat{\mu}_0(T(r))$, which proves Premise 2. ◀

The following examples show that the improvement is strict, even if we allow preprocessing steps reflection and projection.

▶ **Example 9.** The SRS $baababaab \to abaabbaabab$, after preprocessing, satisfies the premises of Theorem 7 but not those of Theorem 2. Check that $T(\bar{\ell}) = (0,2,1,2,0)$, $T(\bar{r}) = (0,1,2,0,2,1)$. We get

$$\hat{\mu}_0(T(\bar{\ell})) = (0,1,1,2,1) \neq (-1,0,0,1,0) = \mu(T(\bar{\ell})) \text{ and}$$
$$\hat{\mu}_0(T(\bar{r})) = (0,0,1,0,1,1) \neq (-1,-1,0,-1,0,0) = \mu(T(\bar{r}));$$
$$\hat{\mu}_1(T(\bar{\ell})) = (0,1,1,2,1) = \mu(T(\bar{\ell})) + 1 \text{ and}$$
$$\hat{\mu}_1(T(\bar{r})) = (0,0,1,0,1,1) = \mu(T(\bar{r})) + 1 \ .$$

▶ **Example 10.** The SRS $abaaabba \to abbababaaab$ satisfies, after preprocessing, the premises of Theorem 7 but not those of Theorem 2. Check that $T(\ell) = (1,3,0,1)$ and $T(r) = (1,0,1,1,3,0)$. We get

$$\hat{\mu}_0(T(\ell)) = (0,2,1,1) = \mu(T(\ell)),$$
$$\hat{\mu}_0(T(r)) = (0,0,0,0,2,1) \neq (0,-1,-1,-1,1,0) = \mu(T(r)).$$

These two examples and the SRS $abccabca \to abcacabccab$ are the only SRSs $\ell \to r$ with $|r| \leq 11$ that satisfy the premises of Theorem 7 after preprocessing, but satisfy neither Theorem 2 after preprocessing, nor any of the above-mentioned improvements of Kurth's Criteria A to G. In contrast, 16 SRSs with $|r| = 11$ satisfy Theorem 2, too.

## 4 Semantic Labelling

There is an interesting connection between Theorem 7 and Semantic Labelling [11]. For $s \in \{a,b\}^*$, let the interpretation $[s] : \mathbb{N}_0^* \to \mathbb{N}_0^*$ be defined by $[a](x) = x+1$ and $[b](x) = p(x)$. The reflected strings are semantically labelled. Symbol $a$ receives no label; symbol $b$ receives $p(x)$ as its label where $x$ is the interpretation value. For convenience, let $\overline{\mathsf{lab}}_x(s)$ denote $\overline{\mathsf{lab}_x(\bar{s})}$, which is $s$ reflected, labelled, and reflected again. The tuple of labels of $b$ symbols is extracted by a string homomorphism $h : (\{a,b\} \times \mathbb{N}_0)^* \to \mathbb{N}_0^*$ that is defined by $h(a_x) = \varepsilon$ and $h(b_x) = (x)$.

▶ **Lemma 11.** *For all $s \in \{a,b\}^*$, $x \in \mathbb{N}_0$:*
1. $[\bar{s}](x) = \mathsf{last}(\hat{\mu}_x(T(sa)))$;
2. $h(\overline{\mathsf{lab}}_x(sb)) = \hat{\mu}_x(T(s))$.

**Proof.** The proof of Claim 1 is a straightforward induction on $|s|$. Claim 2 is shown by induction on $|s|$. Case $s = \varepsilon$. Then $h(\overline{\mathsf{lab}}_x(b)) = h(b_{p(x)}) = (p(x)) = \hat{\mu}_x((0)) = \hat{\mu}_x(T(\varepsilon))$. Case $s = as'$. Then $h(\overline{\mathsf{lab}}_x(as'b)) = h(a\,\overline{\mathsf{lab}}_{[a](x)}(s'b)) = h(\overline{\mathsf{lab}}_{x+1}(s'b)) = \hat{\mu}_{x+1}(T(s'b)) = \hat{\mu}_x(T(as'b))$ by Lemma 5(1). Case $s = as'$. Then $h(\overline{\mathsf{lab}}_x(bs'b)) = h(b_{p(x)}\,\overline{\mathsf{lab}}_{[b](x)}(s'b)) = (p(x))\&h(\overline{\mathsf{lab}}_{p(x)}(s'b)) = (p(x))\&\hat{\mu}_x(T(bs'b)) = \hat{\mu}_x(T(bs'b))$ by the inductive hypothesis and Lemma 5(2). ◀

▶ **Example 12.** $h(\overline{\mathsf{lab}}_x(aabab)) = h(a_x a_{x+1} b_{x+2} a_{x+1} b_{x+2}) = (x{+}2, x{+}2)$ and $h(\overline{\mathsf{lab}}_x(abaabb)) = h(a_x b_{x+1} a_x a_{x+1} b_{x+2} b_{x+1}) = (x+1, x+2, x+1)$.

A proof of termination of $R$ by Theorem 7 can be turned into a proof by Semantic Labelling in the following way:

▶ **Theorem 13.** *For a SRS* $(R, \{a, b\})$ *let* $R' = \{\ell a^i b \to r a^i b \mid (\ell \to r) \in R, i \in \mathbb{N}_0\}$.

1. *If Condition 1 of Theorem 7 holds then* $[]$ *is a quasi-model of* $\overline{R'}$.
2. *If Condition 2 of Theorem 7 holds then* $\overline{\mathsf{lab}}_x(R')$ *is ordered by the reduction order* $>_{\mathrm{mult}}$ *on* $\mathbb{N}_0^*$.

**Acknowledgements** The two anonymous referees provided valuable feedback.

## References


1 Alfons Geser. *Is termination decidable for string rewriting with only one rule?* Habilitation thesis, Wilhelm-Schickard-Institut, Universität Tübingen, Germany, January 2002. 201 pages.

2 Dieter Hofbauer and Johannes Waldmann. Termination of string rewriting with matrix interpretations. In Frank Pfenning, editor, *RTA*, volume 4098 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2006.

3 Yuji Kobayashi, Masashi Katsura, and Kayoko Shikishima-Tsuji. Termination and derivational complexity of confluent one-rule string rewriting systems. *Theoret. Comput. Sci.*, 262(1/2):583–632, 2001.

4 Winfried Kurth. *Termination und Konfluenz von Semi-Thue-Systemen mit nur einer Regel.* Dissertation, Technische Universität Clausthal, Germany, 1990.

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

6 Robert McNaughton. Well-behaved derivations in one-rule Semi-Thue Systems. Technical Report 95-15, Dept. of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, November 1995. See also "Correction by the author to 'Well-behaved derivations in one-rule Semi-Thue Systems'", unpublished paper, July, 1996.

7 Robert McNaughton. Semi-Thue Systems with an Inhibitor. *J. Automated Reasoning*, 26:409–431, 1997.

8 Wojciech Moczydlowski and Alfons Geser. Termination of single-threaded one-rule semi-thue systems. In Jürgen Giesl, editor, *RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 338–352. Springer, 2005.

9 Geraud Sénizergues. On the termination problem for one-rule Semi-Thue Systems. In Harald Ganzinger, editor, *Proc. 7th Int. Conf. Rewriting Techniques and Applications*, LNCS 1103, pages 302–316. Springer, 1996.

10 Kayoko Shikishima-Tsuji, Masashi Katsura, and Yuji Kobayashi. On termination of confluent one-rule string rewriting systems. *Inform. Process. Lett.*, 61(2):91–96, 1997.

11 Hans Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995.

12 Hans Zantema and Johannes Waldmann. Termination by quasi-periodic interpretations. In Franz Baader, editor, *RTA*, volume 4533 of *Lecture Notes in Computer Science*, pages 404–418. Springer, 2007.

# Ordering Networks

**Lars Hellström**[1]

1 **Division of Applied Mathematics, The School of Education, Culture and Communication, Mälardalen University, Box 883, 721 23 Västerås, Sweden;** `lars.hellstrom@residenset.net`

**Abstract**

This extended abstract discusses the problem of defining quasi-orders that are suitable for use with network rewriting. The author's primary interest is in using network rewriting as a tool for equational reasoning in algebraic theories with both operations and co-operations.

**Keywords and phrases** rewriting, network, PROP, PROP order

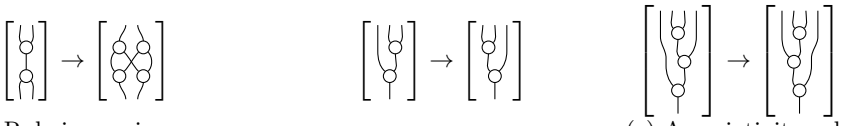
## 1 Introduction

Network rewriting [1] is a kind of graph rewriting; networks are directed acyclic graph with some extra structure—roughly the same extra structure as makes terms out of trees, but completely symmetric with respect to input and output. This allows networks to be viewed as expressions, so that on one hand one can use networks as an alternative notation where ordinary terms do not quite suffice, and on the other one can take a network and *evaluate* it with rather arbitrary interpretations of the symbols. This latter approach turns out to be convenient for defining orders on networks.

Formally, a *network* is a directed acyclic graph (DAG) with the following extra data. (i) There are two distinguished vertices 0 and 1 that represent the output and input respectively sides of the network; edges from 1 are input legs of the network, and edges to 0 are output legs of the network. (ii) Each inner vertex (those other than 0 or 1) is decorated with a symbol from a doubly ranked alphabet. If the symbol $D(v)$ of vertex $v$ has rank $(m, n)$, then the in-degree of $v$ must be $n$ (the *arity*) and the out-degree of $v$ must be $m$ (the *coarity*). (iii) There is at each vertex a total ordering of the incoming edges, and a separate total ordering of the outgoing edges. The arity of the network as a whole is the degree (all outgoing) of the input vertex 1, and the coarity of the network as a whole is the degree (all incoming) of the output vertex 0. By convention here, networks are drawn with all edges oriented downwards, so no arrowheads need to be drawn in them.

The use of networks as expressions when ordinary terms do not suffice may be observed in several specialities—physicists working with tensor fields (e.g. in General Relativity) may use the Penrose [6] graphical notation to visualise the structure of a complex expression, algebraicists studying Hopf algebras may use 'diagram shorthand' (see e.g. [4]) to do their calculations, and quantum computer programming is very much a matter of building 'arrays of quantum gates'—all of which may be formalised as networks or minor variations thereof. The common factor in these applications are operations that produce multiple results (in the sense of a subroutine having several out-parameters, not in the sense of a multivalued function). Much of what specialists in these fields do with their diagrams can be described as informal network rewriting.

The abstract setting within which one may evaluate a network is that of an algebraic structure known as a PROP [3, Ch. V]. This consists of a set of doubly ranked elements (sometimes formalised as the set of all morphisms in a category whose objects are the nonnegative integers; the domain of a morphism is then its arity and the codomain is its



(a) Rule increasing the number of vertices

(b) Associativity rule

(c) Associativity rule in a context

**Figure 1** Network rewrite rules (that can be troublesome to order)

coarity) together with two composition operations $\circ$ and $\otimes$, and a mapping $\phi$ of permutations to PROP elements. One PROP that elegantly illustrates the syntactic constraints of the PROP concept is that which takes as underlying set the set $\mathcal{R}^{\bullet\times\bullet}$ of all matrices over some (semi)ring $\mathcal{R}$: the arity is then the number of columns, the coarity is the number of rows, the $\circ$ operation is matrix multiplication (not defined unless the arity of the left factor equal the coarity of the right factor, just like composition of morphisms in a category), the image of a permutation is the corresponding permutation matrix, and the $\otimes$ operation constructs a larger matrix with the two operands as blocks, like $A \otimes B = \begin{bmatrix} A & 0 \\ 0 & B \end{bmatrix}$. PROPs also need to satisfy a number of axioms, but it would take too long time to state those here; suffices it to say that they are equivalent to the claim that networks can serve as expressions for PROPs [1, Th. 5.17].

This last point may also be stated as the claim that the set of all networks (or rather isomorphism classes of networks) on a given alphabet constitutes the free PROP with respect to that alphabet. The $\circ$ operation then amounts to joining the outputs of the right operand to the inputs of the left operand, whereas $\otimes$ simply places the operands side-by-side, exposing each input and output of either operand as an input or output of the combined network. The network corresponding to a permutation $\sigma$ consists only of edges from 1 to 0, the $j$'th outgoing edge at 1 also being the $\sigma(j)$'th incoming edge at 0. Formalised this way, the reason that an arity- and coarity-preserving function $f$ from a doubly ranked set $X$ to a PROP $\mathcal{P}$ gives rise to an evaluation map $\mathrm{eval}_f$ from the set of all networks on $X$ to $\mathcal{P}$ is that $\mathrm{eval}_f$ is the unique morphism from the free PROP to $\mathcal{P}$ whose existence is guaranteed by the universal property.

## 2 The biaffine PROP

One slightly nontrivial PROP is the *biaffine PROP* $\mathrm{Baff}(\mathcal{R})$, which can be defined over any associative (semi)ring $\mathcal{R}$ with unit. The name can be understood as hinting at the fact that the matrix PROP $\mathcal{R}^{\bullet\times\bullet}$ defined above can be described as a PROP of linear transformations. If each PROP element in addition to the matrix part also gets a translation part, then one could make a PROP of *affine* transformations (an element of arity $n$ and coarity $m$ maps an $n$-dimensional space into an $m$-dimensional space). The biaffine PROP does that too, but goes further to preserve the symmetry of input and output. A rank $(m, n)$ element of the biaffine PROP $\mathrm{Baff}(\mathcal{R})$ consists of four parts: an $m \times n$ matrix $A$, an $m \times 1$ vector $\mathbf{b}$, a $1 \times n$ vector $\mathbf{c}$, and a scalar $d$, wherein all elements are from the (semi)ring $\mathcal{R}$. It is often convenient to place these parts as blocks into an $(m+2) \times (n+2)$ matrix as follows

$$\begin{bmatrix} 1 & d & \mathbf{c} \\ 0 & 1 & 0 \\ 0 & \mathbf{b} & A \end{bmatrix} \tag{1}$$

since the composition $\circ$ is then ordinary matrix multiplication. The image under $\phi$ of a permutation has the matrix part $A$ equal to the permutation matrix but the other three parts zero. The $\otimes$ operation is given by

$$\begin{bmatrix} 1 & d_1 & \mathbf{c}_1 \\ 0 & 1 & 0 \\ 0 & \mathbf{b}_1 & A_1 \end{bmatrix} \otimes \begin{bmatrix} 1 & d_2 & \mathbf{c}_2 \\ 0 & 1 & 0 \\ 0 & \mathbf{b}_2 & A_2 \end{bmatrix} := \begin{bmatrix} 1 & d_1 + d_2 & \mathbf{c}_1 & \mathbf{c}_2 \\ 0 & 1 & 0 & 0 \\ 0 & \mathbf{b}_1 & A_1 & 0 \\ 0 & \mathbf{b}_2 & 0 & A_2 \end{bmatrix}.$$

It is always technically possible to decompose a network into simpler networks using $\circ$ and $\otimes$, and through such a decomposition calculate its value in a particular PROP, but it is often inconvenient to do so. In the biaffine PROP, it is fairly straightforward to evaluate a network without that detour over $\circ$ and $\otimes$. To do this, each inner vertex $v$ of the network should first have been assigned a corresponding biaffine PROP element (usually the value of the symbol at the vertex) with parts $A(v)$, $\mathbf{b}(v)$, $\mathbf{c}(v)$, and $d(v)$. Proceeding from input side to output side (the converse is equally possible), one calculates for each edge (i) a row of an intermediate $A$ matrix, and (ii) an element of an intermediate $\mathbf{b}$ vector. Denote by $A^-(v)$ the matrix obtained by stacking the rows assigned to the incoming edges at vertex $v$, in the order of those edges at that vertex, and similarly denote by $A^+(v)$ the matrix obtained by stacking the rows assigned to outgoing edges at that vertex. Then at the input vertex 1 initialise $A^+(1) = I$ and at each inner vertex $v$ let $A^+(v) = A(v)A^-(v)$; the matrix part of the value of the network as a whole is then the $A^-(0)$ matrix of the output vertex 0. If one similarly denotes by $\mathbf{b}^-(v)$ and $\mathbf{b}^+(v)$ respectively the vectors obtained by combining the vector elements assigned to the incoming and outgoing edges at vertex $v$, then $\mathbf{b}^+(1) = 0$ and $\mathbf{b}^+(v) = \mathbf{b}(v) + A(v)\mathbf{b}^-(v)$ at each inner vertex $v$, with $\mathbf{b}^-(0)$ being the $\mathbf{b}$ part of the value of the network. The $\mathbf{c}$ and $d$ parts of the value of the network as a whole are then

$$\mathbf{c} = \sum_{\text{inner vertex } v} \mathbf{c}(v)A^-(v), \qquad d = \sum_{\text{inner vertex } v} \big(d(v) + \mathbf{c}(v)\mathbf{b}^-(v)\big).$$

Yet another way to understand at least the biaffine PROP $\mathrm{Baff}(\mathbb{N})$ is as a generalised path-counting device. Consider the element of $\mathrm{Baff}(\mathbb{N})$ to which a particular network evaluates. In the matrix part $A$, element $A_{i,j}$ then keeps track of the number of paths from input leg $j$ to output leg $i$. In the vector parts $\mathbf{b}$ and $\mathbf{c}$, element $b_i$ keeps track of the number of paths which begin somewhere inside the network and leave through output leg $i$ whereas element $c_j$ keeps track of the number of paths which enter through input leg $j$ and end somewhere inside the network, and the scalar part $d$ keeps track of the number of paths which both begin an end inside the network. It is easily checked that that the definitions of $\circ$, $\otimes$, and permutations in $\mathrm{Baff}(\mathbb{N})$ are consistent with this interpretation. What makes it a *generalised* path-counting device is that the PROP elements assigned to the individual vertices need not reflect the number of paths in the actual DAG underlying the network.

## 3 Network rewriting and PROP orders

When formalising, and in particular *automating*,[1] network rewriting, it becomes necessary to somehow order the networks so that no rewrite cycles arise. Defining orders that take the graph-theoretic structure of a network into account has however turned out to be surprisingly difficult, so the point of this text is to summarise the solutions that the author has found, and to point out some of the difficulties that one encounters.

[1] See `http://www.mdh.se/ukk/personal/maa/lhm03/sw/rewriting` for one utility that does this.

What is easy to do is to count vertices carrying a particular symbol, and order by that. This corresponds quite directly to ordering by (weighted) degree of polynomial, but that rarely gets one all the way, and there are even cases in which the intuitive rewrite direction may cause the number of vertices to increase (Figure 1a).

Similarly counting edges is not at all straightforward, as illustrated in Figure 1b: one may think that the purpose of this rewrite rule is to eliminate instances of a  vertex as the right child of another, and in a way it is, but one cannot state this goal simply as decreasing the number of edges from the output of a  vertex to the right input of another  vertex. Applying the rule of Figure 1b clearly consumes such an edge, but the catch is that it can also create another such an edge, as shown in Figure 1c; the rule is being applied to the bottom two vertices. The problem with 'count two-vertex subgraphs of the  form' is that this quantity does not change deterministically when a rule is placed in a context. It is possible to get somewhere with this ordering idea, but it requires keeping track of more than just the number of edges where the rule might apply, and in the end it turns out that the construction can be expressed more succintly in terms of the biaffine PROP Baff($\mathbb{N}$).

A *PROP quasi-order* is a transitive and reflexive relation $\leqslant$ on a PROP $\mathcal{P}$ such that $a_1 \leqslant a_2$ and $b_1 \leqslant b_2$ implies $a_1 \circ b_1 \leqslant a_2 \circ b_2$ (whenever those compositions are defined) and $a_1 \otimes b_1 \leqslant a_2 \otimes b_2$. The order is said to be *strict* if $\circ$ and $\otimes$ preserve strictness of inequalities. Given any PROP with a strict PROP quasi-order, one can pull that order back to the free PROP of networks via an evaluation map $\mathrm{eval}_f$, and thereby define a strict PROP quasi-order on the networks. Because the direction of inequalities with respect to such an order is preserved when using $\circ$ and $\otimes$ to embed a network as a subexpression of a larger network, a network rewrite rule $l \to r$ where $r < l$ with respect to such an order will remain consistently oriented no matter what context $C$ it gets placed into, as it will follow that $C(r) < C(l)$. (Technically, in the case of the rewriting machinery of [1], it is also necessary that the order has the *strict uncut property* [1, pp. 152–157], but that has so far never emerged as an obstacle.)

What makes the biaffine PROP Baff($\mathbb{N}$) useful here is that the element-wise partial order on it (standard matrix order, if one considers the block matrix form (1) for an element) is a well-founded PROP quasi-order, and if one restricts to matrices with at least one positive element in each row and each column (again as with respect to the block matrix form) then this quasi-order will be strict. An assignment $f$ that is useful for the rule in Figure 1b is

$$f\left(\text{  }\right) = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

(network has coarity 1 and arity 2, so the element of Baff($\mathbb{N}$) it is mapped to must have that as well, and with the padding of the block matrix form that comes out as $3 \times 4$)

making

$$\mathrm{eval}_f\left(\left[\text{}\right]\right) = \begin{pmatrix} 1 & 0 & 0 & 1 & 2 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix} > \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix} = \mathrm{eval}_f\left(\left[\text{  }\right]\right).$$

An assignment that is useful for the rule in Figure 1a is

$$g\left(\text{  }\right) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix}, \qquad g\left(\text{  }\right) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix}^{\mathrm{T}}$$

as that will have the $d$ part of $\mathrm{eval}_g$ of the left hand side of Figure 1a come out as 1, but the $d$ part of $\mathrm{eval}_g$ of the right hand side come out as 0, which suffices for a strict inequality; all other parts come out equal.

(a) Given rule (orientation of the equation itself) (b) First derived rule

**Figure 2** Rewrite system based on the Yang–Baxter equation

The nice thing about using the biaffine PROP for ordering networks is that it turns out to be very versatile. What is perhaps a bit worrying is that there seems to be few known examples of going beyond the biaffine PROP. Lafont [2, p. 300] sets out with a seemingly more general construction of an order, but upon closer examination it turns out that the functions used must satisfy some additional condition in order for everything to fit together, and if that condition is to be polynomials of degree at most one then we are back to a special case of the biaffine PROP. The only example of a PROP order genuinely distinct from what the biaffine PROP can produce that is known to the author is instead the connectivity PROP of [1, Ex. 3.3], which embellishes the cyclomatic number of the underlying graph.

So far, neither of these have been of much use when completing the rewrite system consisting of the rule in Figure 2a (this *braid identity* constitutes an abstract form of the Yang–Baxter equation, and is mentioned as an example in e.g. [5]). In the case of the biaffine PROP, choosing an order of the networks amounts to picking a value for the  vertices, i.e., to assign values to the elements of the corresponding (1) matrix; there are nine elements whose values are not fixed, and these may be taken as variables parametrising the space of binaffine PROP-based orderings of networks with only  vertices. The claim that a particular rule is oriented in a particular direction gives rise to a system of polynomial inequalities in those nine variables. Considering only the rule of Figure 2a, that system has a solution with strict inequality. Completion will however immediately proceed to derive the rule in Figure 2b (by vertical symmetry of the first rule, either orientation of the derived rule is possible), and if adding also the inequalities resulting from that comparison, there is no longer a solution with strict inequality; the biaffine PROP fails to distinguish one side of a rule as strictly larger than the other. Switching to $\mathrm{Baff}(\mathcal{R})$ for a more general semiring $\mathcal{R}$ has been tried, but so far without much luck.

## References


1 Lars Hellström. *Network Rewriting I: The Foundation*, 2012. arXiv:1204.2421v1 [math.RA].

2 Yves Lafont. Towards an algebraic theory of Boolean circuits. *J. Pure Appl. Algebra* **184** (2003), 257–310.

3 Saunders MacLane. Categorical Algebra. *Bull. Amer. Math. Soc.* **71** (1965), 40–106.

4 Shahn Majid. Cross Products by Braided Groups and Bosonization. *J. Algebra* **163** (1994), 165–190.

5 Samuel Mimram. *Computing Critical Pairs in 2-Dimensional Rewriting Systems.* arXiv:1004.3135v1 [cs.FL].

6 Roger Penrose. Applications of negative dimensional tensors. In Dominic J.A. Welsh, editor, *Combinatorial Mathematics and its Applications*, pages 211–244. Academic Press, 1971.

# On the derivational complexity of Kachinuki orderings

**Dieter Hofbauer**

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

**Abstract**

For string rewriting systems compatible with the standard Kachinuki ordering it is known that their derivational complexity is primitive recursively bounded. However, in case the definition of Kachinuki orderings comprises a possibly different lexicographic status for different letters, the standard upper bound proof method by monotone interpretations into the natural numbers fails, and primitive recursive complexity bounds are an open problem, to the best of our knowledge. In this talk, such an upper bound result is shown by examining worst case rewriting strategies.

**Keywords and phrases** Kachinuki ordering, string rewriting, termination, derivational complexity

## 1 Introduction

We assume familiarity with basic notations for string rewriting. Here, $\Sigma^*$ is the set of strings over an alphabet $\Sigma$, $|x|$ denotes the length of a string $x$, $|x|_a$ denotes the number of occurrences of letter $a$ in $x$, and $\epsilon$ is the empty string. A rewriting system $R$ is *compatible* with a relation $>$ on $\Sigma^*$ if $R \subseteq >$, and $\mathrm{Nf}(>)$ denotes the set of normal forms modulo $>$. For a given ordering $>$ on a set $A$, let $>^{\triangleleft\text{-lex}}$ denote the *right-to-left* length-lexicographic extension of $>$ to $A^*$, analogously define the *left-to-right* variant $>^{\triangleright\text{-lex}}$. For $x = c_0 \dots c_n \in \Sigma^*$, $c_i \in \Sigma$, let $x^{\mathrm{rev}} = c_n \dots c_0$ denote the *reversal* of $x$, and for a rewriting system $R$ define $R^{\mathrm{rev}} = \{\ell^{\mathrm{rev}} \to r^{\mathrm{rev}} \mid \ell \to r \in R\}$. Presently, we consider only finite alphabets and rewriting systems.

The *derivation height* of a string $x$ modulo some finitely branching and well-founded relation $>$ on $\Sigma^*$ is $\mathrm{dh}_>(x) = \max\{n \mid \exists y : x >^n y\}$, and the *derivational complexity* of $>$ maps a natural number $n$ to the maximal derivation height of strings of size at most $n$, i. e., $\mathrm{dc}_>(n) = \max\{\mathrm{dh}_>(x) \mid |x| \le n\}$. For a string rewriting system $R$ let $\mathrm{dh}_R$ and $\mathrm{dc}_R$ abbreviate $\mathrm{dh}_{\to_R}$ and $\mathrm{dc}_{\to_R}$, respectively.

## 2 Kachinuki orderings

Throughout, we use $\succ$ as an ordering on $\Sigma$, a so-called *precedence*. Since we consider chains of maximal length, we assume $\succ$ to be total. Further, each letter $a \in \Sigma$ has an assigned *status* $s(a) \in \{\triangleleft, \triangleright\}$, either *right-to-left* ($\triangleleft$) or *left-to-right* ($\triangleright$).

▶ **Definition 1.** The *Kachinuki ordering* with status function $s$ on $\Sigma^*$ is defined by $x \succ_s y$ provided one of the following holds, where $a$ is the maximal letter modulo $\succ$ occurring in $x$ or $y$:

- $|x|_a > |y|_a$, or
- $|x|_a = |y|_a = n$, $x = x_0 a x_1 \dots a x_n$, $y = y_0 a y_1 \dots a y_n$ for $x_i, y_i \in (\Sigma \setminus \{a\})^*$, and $(x_0, x_1, \dots, x_n) \succ_s^{s(a)\text{-lex}} (y_0, y_1, \dots, y_n)$.

Varying the status function $s$, there are $2^{|\Sigma|-1}$ many different Kachinuki orderings, as is easily seen, all of them being well-founded and closed under contexts. Let $\succ^{\lhd}$ denote the particular *standard* Kachinuki ordering where each letter has right-to-left status, cf. [7].

▶ **Example 2.** The rewriting system $R_0 = \{ac \to ca,\ cb \to bc\}$ is compatible with $\succ_s$ for $a \succ b \succ c$ and $s(a) = \lhd$, $s(b) = \rhd$. For this example, also the standard Kachinuki ordering could be used as $R_0 \subseteq \succ^{\lhd}$ for $a \succ c \succ b$. In contrast, $R_1 = \{ac \to cca,\ cb \to bcc\} \subseteq \succ_s$ (as for $R_0$), but both $R_1$ and $R_1^{\mathrm{rev}}$ are incompatible with $\succ^{\lhd}$, whatever precedence is chosen.

▶ Remark. Under the usual isomorphism between $\Sigma^*$ and the set of terms $\mathcal{T}_\Sigma(\{x\})$ via $a_0a_1 \dots a_n \mapsto a_0(a_1(\dots a_n(x)\dots))$, with variable $x$ and $\Sigma$ be seen as a set of unary function symbols, the Kachinuki ordering $\succ^{\lhd}$ corresponds to the precedence based path ordering on terms, i. e., the *multiset path ordering* without exploiting multisets. Therefore, if a finite string rewriting system $R$ is compatible with $\succ^{\lhd}$, then $\mathrm{dc}_R$ is primitive recursively bounded, and the relationship between the size of the underlying alphabet and the corresponding level of the Grzegorczyk hierarchy for $\mathrm{dc}_R$ is known, see [3, 4].

▶ Remark. The standard upper bound proof method by monotone interpretations into the natural numbers fails for Kachinuki orderings since no such interpretation exists for $\{cb \to bcc\}$, see [8]. Therefore, a monotone interpretation for system $R_1$ (and also for $R_1^{\mathrm{rev}}$) from Example 2 doesn't exist.

Depending on the status function $s$, there is a simple order isomorphism $\lambda x.x^{\lhd}$ on $\Sigma^*$ such that, for $x, y \in \Sigma^*$ and arbitrary precedence $\succ$,

$$x \succ_s y \quad \text{if, and only if,} \quad x^{\lhd} \succ^{\lhd} y^{\lhd}.$$

For $a$ being the maximal letter modulo $\succ$ occurring in $x = x_0a \dots ax_n$ with $x_i \in (\Sigma \setminus \{a\})^*$, define the *straightening* of $x$ as $x^{\lhd} = x_0^{\lhd}a \dots ax_n^{\lhd}$ for $s(a) = \lhd$, and $x^{\lhd} = x_n^{\lhd}a \dots ax_0^{\lhd}$ for $s(a) = \rhd$.

Considering a rewriting system $R$ compatible with some Kachinuki ordering $\succ_s$, at first glance it might be tempting to assume that derivations modulo $R$ are isomorphic to derivations modulo $R^{\lhd} = \{\ell^{\lhd} \to r^{\lhd} \mid \ell \to r \in R\}$. Since $R \subseteq \succ_s$ holds if, and only if, $R^{\lhd} \subseteq \succ^{\lhd}$, upper bound results for $\succ^{\lhd}$ would then imply upper bound results for $\succ_s$. However, the crucial observation is that

$$x \to_R y \quad \text{does not imply} \quad x^{\lhd} \to_{R^{\lhd}} y^{\lhd}.$$

▶ **Example 3.** For $R = \{ab \to ba\}$ over alphabet $\{a, b, c\}$, precedence $a \succ b \succ c$, status function $s$ with $s(a) = \lhd$, $s(b) = \rhd$ and arbitrary $n \neq 0$, we have $R^{\lhd} = R$ and

$$c^nab \to_R c^nba, \text{ but } (c^nab)^{\lhd} = c^nab \not\to_{R^{\lhd}} bc^na = (c^nba)^{\lhd}.$$

This shows that in general rewriting modulo some finite system $R$ translates into rewriting on straightened strings modulo some infinite system (or rule scheme) with unbounded size of right-hand sides of rules. For infinite rewriting systems compatible with the standard Kachinuki ordering, however, the derivational complexity is not necessarily primitive recursively bounded, as the following example shows.

▶ **Example 4.** The infinite string rewriting system $A = \{ab \to bba\} \cup \{b^iaab \to ab^ia \mid i \geq 0\}$ is compatible with $\succ^{\lhd}$ for $a \succ b$, but $\mathrm{dc}_A$ is not primitive recursively bounded [5]. (In fact, $A$ is a string version of the term rewriting system from [5], slightly simplified.) Note that $A$ has infinitely many right-hand sides, although $\to_A$ is finitely branching.

## 3 A worst case derivation strategy

For the rest of the exposition let $\Sigma = \{a_0, \ldots, a_n\}$, and consider the total precedence $a_n \succ \cdots \succ a_0$. First, we give a characterization of Kachinuki orderings on strings of bounded size in terms of rewriting relations.

▶ **Definition 5.** Let $P_0 = \{a_i \to \epsilon \mid 0 \le i \le n\}$ and, for $k > 0$,

$$P_k = \{a_0 \to \epsilon, a_{i+1} \to a_i^k, a_{i+1}a_0 \to a_i^{k-1}a_{i+1} \mid 0 \le i < n\}.$$

Note that $P_k$ is terminating because of $P_k \subseteq \succ^{\lhd}$. Further, $a_i \to_{P_k}^+ \epsilon$, thus $a_{i+1} \to_{P_k}^+ a_i$ for $k > 0$. Since $x \to_{P_k}^+ \epsilon$ for $x \in \Sigma^*$, normal forms modulo $P_k$ are unique, thus $P_k$ is confluent. For other characterizations of path orderings (on general terms) by rewriting systems see [6, 2, 1]; in contrast to $P_k$, these systems are non-terminating, however.

▶ **Lemma 6.** *For $x, y \in \Sigma^*$ and $k \ge 0$, if $x \succ^{\lhd} y$ and $|y| \le k$, then $x \to_{P_k}^+ y$.*

**Proof.** By induction on $x$ along $\succ^{\lhd}$. ◀

▶ **Corollary 7.** *For a rewriting system $R$ compatible with the standard Kachinuki ordering where $\{|r| \mid \ell \to r \in R\}$ is finite, $\mathrm{dc}_R$ is primitive recursively bounded.*

**Proof.** Sketch (a special case of [4]): For $\max\{|r| \mid \ell \to r \in R\} = k$, define a strictly monotone interpretation for $P_k$ as follows. Inductively define functions $\bar{a}_i$ on the natural numbers by $\bar{a}_0(m) = m + 1$, $\bar{a}_{i+1}(0) = \bar{a}_i^k(m) + 1$, and $\bar{a}_{i+1}(m+1) = \bar{a}_i^{k-1}(\bar{a}_{i+1}(m)) + 1$. Then $\mathrm{dc}_R(m) \le \mathrm{dh}_{P_k}(a_n^m) \le \bar{a}_n^m(0)$. ◀

Next, we determine a worst case rewriting strategy for $P_k$ in the sense that it induces maximal derivation lengths. The cases $k \le 1$ are uninteresting as all derivations have the same length and the derivational complexity of $P_k$ is linear, therefore we assume $k > 1$ from now on.

▶ **Definition 8.** Let $\rightharpoonup_k$ on $\Sigma^*$ be defined as the following relation, where $u \in \Sigma^*$, $v \in \{a_0\}^*$, $0 \le i < n$, $m \ge 0$:

$$\begin{aligned} a_0^{m+1} &\rightharpoonup_k a_0^m, \\ ua_{i+1} &\rightharpoonup_k ua_i^k, \\ ua_{i+1}a_0v &\rightharpoonup_k ua_i^{k-1}a_{i+1}v. \end{aligned}$$

Observe that $\rightharpoonup_k \subseteq \to_{P_k}$, that $\epsilon$ is the only normal form modulo $\rightharpoonup_k$, and that $\rightharpoonup_k$ is deterministic in the sense that every string except $\epsilon$ has exactly one successor modulo $\rightharpoonup_k$. Further note that $\rightharpoonup_k$ does not correspond to an innermost (i. e., rightmost) derivation strategy for $P_k$. The following commutation property is needed later.

▶ **Lemma 9.** *For $x, y_1, y_2 \in \Sigma^*$, if $y_1 \leftarrow_{P_k} x \rightharpoonup_k y_2$, then $y_1 \rightharpoonup_k \circ \leftarrow_{P_k} y_2$ or $y_1 \leftarrow_{P_k}^* y_2$.*

**Proof.** We proceed by case analysis, using the notations from Definition 8. Recall that we assume $k > 1$. Case 1: $x = a_0^{m+1}$, thus $y_1 = y_2 = a_0^m$. Case 2: $x = ua_{i+1}$, thus $y_2 = ua_i^k$. Then either $y_1 = y_2$, or $y_1 = u'a_{i+1}$ for $u \to_{P_k} u'$. In the latter case the underlying redexes to not overlap, thus $y_1 \rightharpoonup_k u'a_i^k \leftarrow_{P_k} y_2$. Case 3: $x = ua_{i+1}a_0v$, thus $y_2 = ua_i^{k-1}a_{i+1}v$. If $y_1 = y_2$ we are done. If $y_1 = u'a_{i+1}a_0v$ for $u \to_{P_k} u'$, then $y_1 \rightharpoonup_k u'a_i^{k-1}a_{i+1}v \leftarrow_{P_k} y_2$. Similarly, if $y_1 = ua_{i+1}a_0v'$ for $v \to_{P_k} v'$, then $v' \in \{a_0\}^*$ and $y_1 \rightharpoonup_k ua_i^{k-1}a_{i+1}v' \leftarrow_{P_k} y_2$. If $y_1 = ua_{i+1}v$ using rule $a_0 \to \epsilon$, then $y_2 \to_{P_k}^* y_1$ since $a_i^{k-1} \to_{P_k}^* \epsilon$. Eventually, if $y_1 = ua_i^kv$ using rule $a_{i+1} \to a_i^k$, then $y_2 \to_{P_k}^* y_1$ since, for $k > 1$, $a_i^{k-1}a_{i+1} \to_{P_k} a_i^{k-1}a_i^k = a_i^ka_i^{k-1} \to_{P_k}^* a_i^ka_i \to_{P_k}^* a_i^ka_0$. ◀

▶ **Lemma 10.** *Let $\to$ and $\rightharpoonup$ be binary relations on the same set such that* $\mathrm{Nf}(\rightharpoonup) \subseteq \mathrm{Nf}(\to)$ *and* $\leftarrow \circ \rightharpoonup \subseteq (\rightharpoonup \circ \leftarrow^+) \cup \leftarrow^*$. *Then for every reduction sequence modulo $\to$ there is a reduction sequence modulo $\rightharpoonup$ of the same length and starting from the same element.*

**Proof.** First we show $\leftarrow^n \circ \rightharpoonup \subseteq \rightharpoonup^= \circ \leftarrow^{\geq n-1}$ for $n > 0$ by induction on $n$: We have $\leftarrow^1 \circ \rightharpoonup \subseteq (\rightharpoonup \circ \leftarrow^+) \cup \leftarrow^* \subseteq \rightharpoonup^= \circ \leftarrow^{\geq 0}$ and, using the induction hypothesis, $\leftarrow^{n+1} \circ \rightharpoonup = \leftarrow^n \circ \leftarrow \circ \rightharpoonup \subseteq \leftarrow^n \circ ((\rightharpoonup \circ \leftarrow^+) \cup \leftarrow^*) = (\leftarrow^n \circ \rightharpoonup \circ \leftarrow^+) \cup (\leftarrow^n \circ \leftarrow^*) \subseteq (\rightharpoonup^= \circ \leftarrow^{\geq n-1} \circ \leftarrow^+) \cup \leftarrow^{\geq n} = \rightharpoonup^= \circ \leftarrow^{\geq n}$.

Now for $n \geq 0$, if $x \to^n y$, then there is some element $y'$ such that $x \rightharpoonup^n y'$. Again, this is shown by induction on $n$. The case $n = 0$ being trivial, assume $x \to \circ \to^n y$. By $\mathrm{Nf}(\rightharpoonup) \subseteq \mathrm{Nf}(\to)$ we obtain $x \rightharpoonup z$ for some $z$, and from $y \leftarrow^{n+1} x \rightharpoonup z$ we get $y \rightharpoonup^= \circ \leftarrow^{\geq n} z$ by the first claim. By induction hypothesis, $z \rightharpoonup^{\geq n} z'$ for some $z'$, so $x \rightharpoonup z \rightharpoonup^n y' \rightharpoonup^* z'$ for some $y'$, concluding the proof. ◀

Combining these two lemmas, we arrive at the announced worst case property of the derivation strategy $\rightharpoonup_k$.

▶ **Corollary 11.** *If $R$ is a rewriting system compatible with $\succ^\lhd$ and* $\max\{|r| \mid \ell \to r \in R\} = k$, *then* $\mathrm{dh}_{\rightharpoonup_k}(x) = \mathrm{dh}_{P_k}(x) \geq \mathrm{dh}_R(x)$.

This result can now be generalized to arbitrary Kachinuki orderings, as follows.

▶ **Theorem 12.** *If $R$ is a rewriting system compatible with $\succ_s$ for status function $s$ and* $\max\{|r| \mid \ell \to r \in R\} = k$, *then* $\mathrm{dc}_{\rightharpoonup_k}(m) \geq \mathrm{dc}_R(m)$.

**Proof.** Sketch: Analogously to $P_k$ and $\rightharpoonup_k$, define a rewriting system $P_{s,k}$ and a corresponding worst case strategy $\rightharpoonup_{s,k}$. Let $L = \{x \in \Sigma^* \mid \exists m \geq 0 : a_n^m \rightharpoonup_{s,k} x\}$. We then show that $x \rightharpoonup_{s,k} y$ is equivalent to $x^\lhd \rightharpoonup_k y^\lhd$ for $x, y \in L$, and conclude by $\mathrm{dc}_{\rightharpoonup_k}(m) = \mathrm{dh}_{\rightharpoonup_k}(a_n^m) = \mathrm{dh}_{\rightharpoonup_{s,k}}(a_n^m) = \mathrm{dh}_{P_{s,k}}(a_n^m) \geq \mathrm{dc}_R(m)$. ◀

▶ **Corollary 13.** *Finite rewriting systems that are compatible with a Kachinuki ordering have primitive recursively bounded derivational complexity.*

Acknowledgments: Thanks to Johannes Waldmann for posing the question addressed here.

## References


1 N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Volume B: Formal Methods and Semantics, Chapter 6, pp. 243–320. North-Holland, 1990.

2 A. Geser. *Relative Termination.* Dissertation, Universität Passau, Germany, 1990.

3 D. Hofbauer. *Termination proofs and derivation lengths in term rewriting systems.* Dissertation, Technische Universität Berlin, Germany, 1991.

4 D. Hofbauer. Termination proofs by multiset path orderings imply primitive recursive derivation lengths. *Theoretical Computer Science*, 105(1):129–140, 1992.

5 D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations. In N. Dershowitz (Ed.), *Proc. Rewriting Techniques and Applications (RTA)*, Volume 355 of *LNCS*, pp. 167–177. Springer, 1989.

6 J. W. Klop. Term rewriting systems. Report CS-R9073, Centrum voor Wiskunde en Informatica, Amsterdam, 1990.

7 K. Sakai. Knuth-Bendix algorithm for Thue system based on Kachinuki ordering. ICOT Technical Memorandum TM-0087, 1984.

8 H. Zantema. Termination of term rewriting: Interpretation and type elimination. *Journal of Symbolic Computation*, 17(1):23–50, 1994.

# Automatic Termination Analysis for GPU Kernels*

**Jeroen Ketema and Alastair F. Donaldson**

**Imperial College London, {jketema,afd}@imperial.ac.uk**

**Abstract**

We describe a method for proving termination of massively parallel GPU kernels. An implementation in KITTeL is able to show termination of 94% of the 598 kernels in our benchmark suite.

## 1 Introduction

Graphics processing units (GPUs) are highly parallel shared-memory processors that can accelerate compute intensive applications. To leverage the power of a GPU, a programmer identifies a part of an application that exhibits parallelism; this part can then be extracted into computational *kernel* and *offloaded* to a GPU.

Kernel programming languages such as CUDA [14] and OpenCL [11] are data-parallel languages that use *barriers* for synchronisation. Roughly, when a thread reaches a barrier, it waits until all other threads have also reached a barrier. Once a barrier has been reached by every thread, execution stalls until all outstanding writes to shared memory have been committed. Committing all writes ensures that any write to shared memory that occurs before a barrier is visible to any thread after the barrier; this enables threads to communicate.

As GPUs are separate devices to which kernels are offloaded, it is generally difficult to perform live debugging. Hence, different means are needed to identify bugs. In previous work [3, 6], we have looked at uncovering data races. Here we consider termination.

Unlike CPU applications, which may be reactive, GPU kernels are *required* to terminate: any data computed by a kernel is inaccessible from the CPU as long as the kernel has not terminated. Besides this practical consideration, termination is also important from a theoretical perspective: the data race detection method described in [3], which underpins our data race detection tool GPUVerify, is only sound for terminating kernels.

We describe and evaluate a method for proving termination of kernels. Termination analysis is complicated by concurrency, but there is no need to reason about recursive calls or dynamically changing data structures since recursion and dynamic memory allocation are not generally supported by kernel programming languages.

The contributions of this paper are two-fold:

1. We leverage termination analysis for sequential programs to obtain an analysis technique for GPU kernels; the technique abstracts from all threads except an arbitrary one.
2. We adapt an existing termination analysis tool—KITTeL [8, 9]—and show that it can be successfully applied to a large set of real-world kernels.

## 2 Reduction to Sequential Termination Analysis

We present the kernel programming language from [5], which has the following grammar:

$$\begin{array}{lll} \text{expr} & e & ::= c \mid v \mid A[e] \mid e_1 \; op \; e_2 \\ \text{stmt} & s & ::= v := e \mid A[e_1] := e_2 \mid \text{if } (e) \, \{ss_1\} \, \text{else} \, \{ss_2\} \mid \text{while } (e) \, \{ss\} \mid \text{barrier} \\ \text{stmts} & ss & ::= \varepsilon \mid s; ss \end{array}$$

Here, $c$ and $v$ represent constants and *thread-private* variables; $A$ and $op$ represent *shared* arrays and arbitrary binary operators. As explained in the introduction, the barrier statement

* This work was supported by the EU FP7 STREP project CARP (project number 287767).

$$\frac{[\![e]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}}}{(\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, \text{while}\,(e)\,\{ss\}; ss') \to_{\mathrm{s}} (\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, ss \cdot \text{while}\,(e)\,\{ss\}; ss')}\ (\textsc{Loop-T}) \qquad \frac{\neg[\![e]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}}}{(\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, \text{while}\,(e)\,\{ss\}; ss') \to_{\mathrm{s}} (\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, ss')}\ (\textsc{Loop-F})$$

**(a)** The thread-level rules (operating over thread states $(\sigma_{\mathrm{v}}, ss)$ and shared memory $\sigma_{\mathrm{A}}$)

$$\frac{K(t) = (\sigma_{\mathrm{v}}, ss) \qquad (\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, ss) \to_{\mathrm{s}} (\sigma'_{\mathrm{v}}, \sigma'_{\mathrm{A}}, ss') \qquad K' = K[t \mapsto (\sigma'_{\mathrm{v}}, ss')]}{(\sigma_{\mathrm{A}}, K) \to_{\mathrm{k}} (\sigma'_{\mathrm{A}}, K')}\ (\textsc{Step})$$

$$\frac{\left(\forall t : \exists \sigma_{\mathrm{v}} : \bigvee \begin{array}{c}(\exists ss : K(t) = (\sigma_{\mathrm{v}}, \text{barrier}; ss) \wedge K'(t) = (\sigma_{\mathrm{v}}, ss)) \\ (K(t) = (\sigma_{\mathrm{v}}, \varepsilon) \qquad \wedge K'(t) = (\sigma_{\mathrm{v}}, \varepsilon))\end{array}\right) \qquad \exists t, \sigma_{\mathrm{v}}, ss : K(t) = (\sigma_{\mathrm{v}}, \text{barrier}; ss)}{(\sigma_{\mathrm{A}}, K) \to_{\mathrm{k}} (\sigma_{\mathrm{A}}, K')}\ (\textsc{Barrier})$$

**(b)** The Kernel-level rules (operating over kernel states $(\sigma_{\mathrm{A}}, K)$)

**Figure 1** Operational semantics of our kernel programming language

allows for synchronisation between threads; $\varepsilon$ represents the empty sequence of statements. A *kernel program* $P$ is a sequence of statements $ss$; all threads execute the same sequence $ss$. For technical reasons we assume that an expression $e$ has at most one sub-expression of the form $A[e']$, so that evaluating an expression involves reading at most once from the shared state; a kernel can be trivially preprocessed to satisfy this restriction.

A *thread state* is a pair $(\sigma_{\mathrm{v}}, ss)$, where $\sigma_{\mathrm{v}}$ represents the private memory of a thread—mapping private variables to values—and where $ss$ is the sequence of the statements the thread needs to execute. A *kernel state* is a pair $(\sigma_{\mathrm{A}}, K)$, where $\sigma_{\mathrm{A}}$ represents the shared memory of the kernel—mapping shared arrays to sequences of values—and where $K$ is a map from a *finite* set of thread identifiers $t$ to thread states. The *initial kernel state* of a kernel program $P$ is any state such that the second component of each $K(t)$ is $P$.

Figure 1 gives the operational semantics of the language. For brevity, we omit the rules for the assignments and if-statement, which are straightforward, and refer the reader to [5]. The rules for the while-statement evaluate the guard $e$ under $\sigma_{\mathrm{v}}$ and $\sigma_{\mathrm{A}}$, denoted $[\![e]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}}$, and proceed accordingly. As can be seen from rule Step, the language has an interleaving semantics. Rule Barrier is used for synchronisation between threads: no thread can proceed beyond a barrier unless all threads have either reached a barrier or have terminated. The rule requires that at least one thread is actually at a barrier; this ensures that the rule no longer fires once all threads have terminated (i.e., once all have reached a state $(\sigma_{\mathrm{v}}, \varepsilon)$).

We next reduce the termination problem for kernel programs to a sequential termination problem. The reduction makes termination analysis for kernel programs thread-modular by checking termination of a single, arbitrary thread under an environmental abstraction that over-approximates the effects of the other threads.

To obtain the abstraction, existentially quantify the premise of each thread-level rule over all array stores $\sigma_{\mathrm{A}}$ and replace rule Barrier by the thread-level rule $(\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, \text{barrier}; ss) \to (\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, ss)$. Denote by $\to_{\mathrm{s},\star}$ the over-approximating thread-level reduction relation such obtained. The relation ensures that the contents of $\sigma_{\mathrm{A}}$ is irrelevant and that a thread no longer has to wait for any other thread once it reaches a barrier. We have the following.

▶ **Theorem 2.1.** *Let $P$ be a kernel program. If for each $\sigma_{\mathrm{v}}$ and $\sigma_{\mathrm{A}}$ it holds that all reductions $(\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, P) \to_{\mathrm{s},\star} \cdots \to_{\mathrm{s},\star} \cdots$ are finite, then $P$ terminates under the semantics of Figure 1.*

**Proof.** Suppose the contrary, then there exists an infinite reduction $\rho$ of $P$. As the number of threads is finite, there is a thread $t$ that is selected an infinite number of times by rule Step. We construct an infinite reduction for $t$ under $\to_{\mathrm{s},\star}$: (i) for each application of Step

selecting $t$ apply the over-approximating version of the thread-level rule employed and (ii) for each application of rule BARRIER employ the over-approximating barrier rule. The over-approximating rules fire, as (i) the existential quantification over all array stores $\sigma_A$ ensures that $e$ can be evaluated precisely as in $\rho$ and as (ii) the thread-level barrier rule essentially skips a barrier. Hence, we have an infinite reduction for $t$ under $\rightarrow_{s,\star}$, contradiction. ◀

A theorem related to the one above underpins the soundness of GPUVerify, where shared state abstraction allows race-freedom to be verified by considering just *two* arbitrary threads [3]. Observe that the theorem only modifies the operational semantics; kernel programs are left unchanged. Furthermore, the reverse of the theorem does not hold: termination might depend on shared memory sub-expressions evaluating to specific values.

## 3 Experimental Evaluation

To evaluate the effectiveness of Theorem 2.1, we adapted the `KITTeL` termination analysis tool [8, 9] and applied it to a suite of 598 kernels, 381 of which have loops. To demonstrate that our approach works out-of-the-box, we included the loop-free kernels in our evaluation. The kernels have on average 86 lines of code and originate from nine sources:

- *AMD Accelerated Parallel Processing SDK* v2.6 [1] (78 kernels, 54 of which have loops).
- *NVIDIA GPU Computing SDK* v5.0 [13] (183 kernels, 109 of which have loops); we also include 8 kernels from v2.0 of the SDK, 7 of which have loops.
- *C++ AMP Sample Projects* [12] (20 kernels, 16 of which have loops)
- The *gpgpu-sim* benchmarks [2] (33 kernels, 24 of which have loops)
- The *Parboil* benchmarks v2.5 [16] (25 kernels, 19 of which have loops)
- The *Rodinia* benchmark suite v2.4 [4] (36 kernels, 24 of which have loops)
- The *SHOC* benchmark suite [7] (87 kernels, 53 of which have loops)
- The *PolyBench/GPU* benchmark suite [10] (64 kernels, 49 of which have loops)
- Rightware *Basemark CL* v1.1 [15] (64 kernels, 26 of which have loops)

Each suite is publicly available except for *Basemark CL* which was provided to us under an academic license. The collection covers all the publicly available GPU benchmark suites we are aware of. The kernel counts above do not include 5 kernels that we manually removed because they use CUDA surfaces or thread fences [14], which we currently do not support.

**KITTeL** The `KITTeL` termination analysis tool [8, 9] consists of a front-end, `llvm2KITTeL`, which takes llvm bitcode[1] and translates this into an integer-based rewrite system. The back-end automatically tries to prove termination of the generated rewrite system.

We adapted `llvm2KITTeL` to handle kernels (compiled to bitcode by Clang[2]); we did not make any changes to the termination analysis back-end. As `llvm2KITTeL` models only a single thread and already abstracts from most memory operations (yielding nondeterministic values for loads from memory), the changes we needed to make were minimal. To summarise: (i) we ensured that `llvm2KITTeL` abstracts loads even in cases where it usually does not (e.g., when a pointer points to a unique global variable representing a single integer), (ii) we disabled the hoisting of loop-invariant loads from loops (due to concurrency the loaded value might differ between loop iterations), and (iii) we made `llvm2KITTeL` aware of the fact that

[1] `http://llvm.org/`
[2] `http://clang.llvm.org/`

the number of threads executing a kernel is constant for the duration of an execution (the number of threads is often referred to in loop guards; hence, awareness that this number is constant—or at least bounded—is often critical for showing termination).

**Loop Invariants** Currently, `KITTeL` does not infer loop invariants that may be needed for proving termination. We provided these invariants by hand and proved them correct with GPUVerify prior to running our experiments. In principle we could extend GPUVerify's invariant inference engine [3] to generate the needed invariants; this would require infrastructure to feed the generated invariants into `KITTeL`.

We required loop invariants stating: (i) the loop counter must be positive (31 kernels), (ii) the step value for the loop counter is positive (18 kernels), (iii) the loop counter is always smaller than or equal to a value which is subtracted from it in the loop guard (2 kernels).

**Experimental Setup** All experiments were conducted on a Mid 2009 MacBook Pro with a 2.53GHz Intel Core 2 Duo and 4GB RAM running OS X 10.9.2 and Clang/llvm 3.4. The reported times are averages over three runs and include the time needed to compile a kernel into bitcode. The timeout used was 10 minutes. We adapted `llvm2KITTeL` as described above and always invoked the tool with its `-increase-strength` option—turning left and right shifts into multiplications and divisions, respectively. The latter facilitates termination analysis of kernels where the loop counter is being shifted. The SMT solver used with `KITTeL` was Z3 v4.3.1. Both `llvm2KITTeL` and `KITTeL` were downloaded on 21-04-2014.[3]

**Results** Unsurprisingly, `KITTeL` managed to prove termination of all 217 loop-free kernels. On average termination of these kernels was shown in 0.63s and the maximum time required was 6.15s. Six of the kernels needed over 1s; this was either due to a long compilation time or the kernel consisting of a large number of subroutines.

Of the 381 kernels with loops, 346 could be shown terminating. On average termination was shown in 7.23s and the maximum time needed was 254.17s (see also Figure 2). Of the 35 kernels for which termination could not be shown, 31 reached the timeout of 10 minutes. In 4 cases `KITTeL` indicated that the constructed rewrite system was nonterminating (this does not imply that the original kernels are nonterminating, as the constructed rewrite system in general over-approximates the behaviour of a thread).

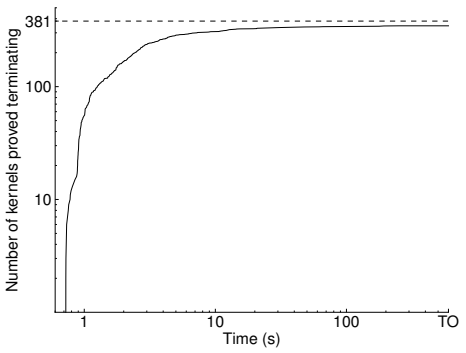



**Figure 2** Cumulative histogram showing the time taken to prove termination of the kernels with loops

We manually inspected the 35 kernels to see why termination could not be shown. All 4 cases where `KITTeL` indicated nontermination would require reasoning over floating point numbers. In 4 other cases built-in atomic increment operations would need to be modelled as returning monotonically increasing values—instead of arbitrary ones, as is currently the case. In 19 cases termination would require reasoning about shared memory and, hence, the over-approximation from Theorem 2.1 is too coarse.

The above leaves 8 kernels, all of which timed out. Of these, 2 could be shown terminating using a very coarse over-approximation of division—yielding unconstrained nondeterministic

[3] `https://github.com/s-falke/{llvm2kittel,kittel-koat}`

values. One kernel could be shown terminating with `llvm2KITTeL`'s `-only-loop-conditions` option, which abstracts all basic blocks except those from which loops can be exited.

In the case of 2 kernels the function bodies were very large which resulted in a timeout in `llvm2KITTeL` (these were the only timeouts in `llvm2KITTeL`). In the 3 remaining cases a timeout occurred in `KITTeL`, although the generated rewrite system was terminating.

## 4 Conclusion

We have described an approach for proving termination of massively parallel GPU kernels by reducing the termination problem for these kernels to a sequential termination problem. With the help of an adapted version of `KITTeL` the reduction allowed us to prove termination of 94% of the kernels in our benchmark set and of 91% of the kernels with loops.

As part of future work we would like to automatically infer loop invariants that are required for proving termination. Moreover, we would like to investigate whether performance can be improved by outlining—as opposed to inlining—loops into separate procedures.

**Acknowledgements** The authors wish to thank Adam Betts, Nathan Chong and Stephan Falke for feedback on the paper. The authors also wish to thank Stephan Falke for making `KITTeL`'s source code publicly available and for answering questions regarding the tool.

## References

1 AMD. AMD Accelerated Parallel Processing (APP) SDK. `http://developer.amd.com/sdks/amdappsdk`.
2 A. Bakhoda et al. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*, pages 163–174, 2009.
3 A. Betts et al. GPUVerify: a verifier for GPU kernels. In *OOPSLA*, pages 113–132, 2012.
4 S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization*, pages 44–54, 2009.
5 N. Chong, A. F. Donaldson, and J. Ketema. A sound and complete abstraction for reasoning about parallel prefix sums. In *POPL*, pages 397–410, 2014.
6 P. Collingbourne, A. F. Donaldson, J. Ketema, and S. Qadeer. Interleaving and lock-step semantics for analysis and verification of GPU kernels. In *ESOP*, pages 270–289, 2013.
7 A. Danalis et al. The scalable heterogeneous computing (SHOC) benchmark suite. In *GPGPU*, pages 63–74, 2010.
8 S. Falke, D. Kapur, and C. Sinz. Termination analysis of C programs using compiler intermediate languages. In *RTA*, pages 41–50, 2011.
9 S. Falke, D. Kapur, and C. Sinz. Termination analysis of imperative programs using bitvector arithmetic. In *VSTTE*, pages 261–277, 2012.
10 S. Grauer-Gray et al. Auto-tuning a high-level language targeted to GPU codes. In *InPar*, 2012.
11 Khronos OpenCL Working Group. The OpenCL specification, version 1.2, 2012.
12 Microsoft Corporation. C++ AMP sample projects for download (MSDN blog). `http://goo.gl/eb8ob`.
13 NVIDIA. GPU Computing SDK. `https://developer.nvidia.com/gpu-computing-sdk`.
14 NVIDIA. CUDA C programming guide, version 5.0, 2012.
15 Rightware Oy. Basemark CL. `http://www.rightware.com/benchmarking-software/basemark-cl/`.
16 J. Stratton et al. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, UIUC, 2012.

# Geometric Series as Nontermination Arguments for Linear Lasso Programs

**Jan Leike[1] and Matthias Heizmann[2]**

1 **The Australian National University**
**Canberra, Australia**
jan.leike@anu.edu.au
2 **University of Freiburg**
**Freiburg, Germany**
heizmann@informatik.uni-freiburg.de

**Abstract**

We present a new kind of nontermination argument for linear lasso programs, called *geometric nontermination argument*. A geometric nontermination argument is a finite representation of an infinite execution of the form $(\vec{x} + \sum_{i=0}^{t} \lambda^i \vec{y})_{t \geq 0}$. The existence of this nontermination argument can be stated as a set of nonlinear algebraic constraints. We show that every linear loop program that has a bounded infinite execution also has a geometric nontermination argument. Furthermore, we discuss nonterminating programs that do not have a geometric nontermination argument.

**1998 ACM Subject Classification** D.2.4 Software/Program Verification

**Keywords and phrases** Nontermination analysis, Infinite execution, Constraint-based synthesis, Linear lasso program

## 1 Introduction

The problem of automatically proving termination of programs has been extensively studied. For restricted classes of programs there are methods proving termination [1, 7] and hence nontermination follows from the absence of a termination proof. For broader classes of programs no complete method for proving termination is known or termination is undecidable. Methods that address these broader classes of programs only check the existence of a certain kind of termination argument, e.g. a specific kind of ranking function. The existence of this termination argument proves termination, however the absence of such a termination argument does not imply nontermination and hence these termination analyses cannot be used to prove nontermination.

Analyses for nontermination proceed in a similar manner. They do not check the existence of a general nontermination proof, instead they check for the existence of a certain kind of nontermination argument, e.g. a recurrence set [3, 5] or an underapproximation of the program that does not terminate for any input [2].

In this paper we present a new kind of nontermination argument for linear lasso programs, called *geometric nontermination argument*. A geometric nontermination argument is a finite representation of an infinite execution that can be denoted as a geometric series. The existence of a geometric nontermination argument can be encoded by a set of nonlinear constraints. Over the reals these constraints are decidable. The advantage of our nontermination arguments lies in their simplicity. In contrast to recurrence sets [3, 5], the constraints that state the existence of our geometric nontermination arguments do not contain quanti-

fier alternation and contain only a small number of nonlinear terms. Unlike [2] we do not need a safety checker to compute nontermination arguments.

## 2 Preliminaries

We consider the following class of programs whose states are real-valued vectors.

▶ **Definition 1** (Linear lasso program). A *(conjunctive) linear lasso program* $P = (\text{STEM}, \text{LOOP})$ consists of two binary relations STEM and LOOP, that are each defined by a formula whose free variables are $\vec{x}$ and $\vec{x}'$ and that have the form $A \begin{pmatrix} \vec{x} \\ \vec{x}' \end{pmatrix} \leq \vec{b}$ for some matrix $A \in \mathbb{R}^{n \times m}$ and some vector $\vec{b} \in \mathbb{R}^m$. We call a linear lasso program *linear loop program* if the formula that defines the relation STEM is equivalent to *true*.

▶ **Definition 2** (Infinite execution). An infinite sequence of states $(\vec{x}_t)_{t \geq 0}$ is an *infinite execution* of the linear lasso program $P = (\text{STEM}, \text{LOOP})$ iff $(\vec{x}_0, \vec{x}_1) \in \text{STEM}$ and $(\vec{x}_t, \vec{x}_{t+1}) \in \text{LOOP}$ for all $t \geq 1$.

## 3 Geometric Nontermination Arguments

▶ **Definition 3.** Let $P = (\text{STEM}, \text{LOOP})$ be a linear lasso program such that LOOP is defined by the formula $A \begin{pmatrix} \vec{x} \\ \vec{x}' \end{pmatrix} \leq \vec{b}$. The tuple $N = (\vec{x}_0, \vec{x}_1, \vec{y}, \lambda)$ is called a *geometric nontermination argument* for $P$ iff the following properties hold.

$$\begin{aligned} \text{(domain)} \quad & \vec{x}_0, \vec{x}_1, \vec{y} \in \mathbb{R}^n,\ \lambda \in \mathbb{R} \text{ and } \lambda > 0. \\ \text{(init)} \quad & (\vec{x}_0, \vec{x}_1) \in \text{STEM} \\ \text{(point)} \quad & A \begin{pmatrix} \vec{x}_1 \\ \vec{x}_1 + \vec{y} \end{pmatrix} \leq \vec{b} \\ \text{(ray)} \quad & A \begin{pmatrix} \vec{y} \\ \lambda \vec{y} \end{pmatrix} \leq \vec{0} \end{aligned}$$

The constraints (init), (point), and (ray) given in Definition 3 are (quantifier free) nonlinear algebraic constraints, the existence of a solution is decidable [6], and hence the existence of a geometric nontermination argument is decidable. We can check the existence of a geometric nontermination argument by passing the constraints of Definition 3 to an SMT solver for nonlinear real arithmetic [4]. If a satisfying assignment is found, this constitutes a nontermination proof in form of an infinite execution according to the following theorem.

▶ **Theorem 4** (Soundness). *If the conjunctive linear lasso program* $P = (\text{STEM}, \text{LOOP})$ *has a geometric nontermination argument* $N = (\vec{x}_0, \vec{x}_1, \vec{y}, \lambda)$ *then* $P$ *has the following infinite execution.*

$$\vec{x}_0,\ \vec{x}_1,\ \vec{x}_1 + \vec{y},\ \vec{x}_1 + (1+\lambda)\vec{y},\ \vec{x}_1 + (1+\lambda+\lambda^2)\vec{y},\ \ldots$$

**Proof.** Define $\vec{z}_0 := \vec{x}_0$ and $\vec{z}_t := \vec{x}_1 + \sum_{i=0}^{t} \lambda^i \vec{y}$. Then $(\vec{z}_t)_{t \geq 0}$ is an infinite execution of $P$: by (init), $(\vec{z}_0, \vec{z}_1) = (\vec{x}_0, \vec{x}_1) \in \text{STEM}$ and

$$A \begin{pmatrix} \vec{z}_t \\ \vec{z}_{t+1} \end{pmatrix} = A \begin{pmatrix} \vec{x}_1 + \sum_{i=0}^{t} \lambda^i \vec{y} \\ \vec{x}_1 + \sum_{i=0}^{t+1} \lambda^i \vec{y} \end{pmatrix} = A \begin{pmatrix} \vec{x}_1 \\ \vec{x}_1 + \vec{y} \end{pmatrix} + \sum_{i=0}^{t} \lambda^i A \begin{pmatrix} \vec{y} \\ \lambda \vec{y} \end{pmatrix} \leq \vec{b} + \sum_{i=0}^{t} \lambda^i \vec{0} = \vec{b},$$

by (point) and (ray). ◀

▶ **Example 5.** Consider the linear loop program $P = (\mathit{true}, \text{LOOP})$ depicted as pseudocode on the left and whose relation $\text{LOOP}(a, b, a', b')$ is defined by the formula depicted on the right.

```
while (a ≥ 7):
    a := b;
    b := a + 1;
```

$$\begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ -1 & 0 & 0 & 1 \\ 1 & 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} a \\ b \\ a' \\ b' \end{pmatrix} \leq \begin{pmatrix} 7 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

Note that in this example, the relation LOOP is defined by an affine-linear transformation and a guard $a \geq 7$. In general, linear lasso programs are defined with linear constraints, which also allow nondeterministic updates of variables.

For $x_0 = \binom{7}{8}$, $x_1 = \binom{7}{8}$, $y = \binom{1}{1}$ and $\lambda = 1$, the tuple $N = (x_0, x_1, y, \lambda)$ is a geometric nontermination argument and the following sequence of states is an infinite execution of $P$.

$$\binom{7}{8}, \binom{7}{8}, \binom{8}{9}, \binom{9}{10}, \binom{10}{11}, \dots$$

We are able to decide the existence of a geometric nontermination argument, however we are not able to decide the existence of an infinite execution because there are programs that have an infinite execution but no geometric nontermination argument as the following example illustrates.

▶ **Example 6.** The following linear lasso program has an infinite execution, e.g. $\binom{2^t}{3^t}_{t\geq 0}$, but it does not have a geometric nontermination argument.

```
while (a ≥ 1 ∧ b ≥ 1):
    a := 2 · a;
    b := 3 · b;
```

## 4 Bounded Infinite Executions

In this section we show that we can always prove nontermination of linear loop programs if there is a bounded infinite execution.

Let $|\cdot| : \mathbb{R}^n \to \mathbb{R}$ denote some norm. We call an infinite execution $(\vec{x}_t)_{t\geq 0}$ *bounded* iff there is a real number $d \in \mathbb{R}$ such that for each state its norm in bounded by $d$, i.e. $|\vec{x}_t| \leq d$ for all $t$.

▶ **Lemma 7** (Fixed Point). *Let $P = (true, \text{LOOP})$ be a linear loop program. The loop $P$ has a bounded infinite execution if and only if there is a fixed point $\vec{x}^* \in \mathbb{R}^n$ such that $(\vec{x}^*, \vec{x}^*) \in \text{LOOP}$.*

**Proof.** If there is a fixed point $\vec{x}^*$, then the loop has the infinite bounded execution $\vec{x}^*, \vec{x}^*, \dots$. Conversely, let $(\vec{x}_t)_{t\geq 0}$ be an infinite bounded execution. Boundedness implies that there is an $d \in \mathbb{R}$ such that $|\vec{x}_t| \leq d$ for all $t$. Consider the sequence $\vec{z}_k := \frac{1}{k}\sum_{t=1}^{k} \vec{x}_t$.

$$\begin{aligned} |\vec{z}_k - \vec{z}_{k+1}| &= \left| \frac{1}{k}\sum_{t=1}^{k} \vec{x}_t - \frac{1}{k+1}\sum_{t=1}^{k+1} \vec{x}_t \right| = \frac{1}{k(k+1)} \left| (k+1)\sum_{t=1}^{k} \vec{x}_t - k\sum_{t=1}^{k+1} \vec{x}_t \right| \\ &= \frac{1}{k(k+1)} \left| \sum_{t=1}^{k} \vec{x}_t - k\vec{x}_{k+1} \right| \leq \frac{1}{k(k+1)} \left( \sum_{t=1}^{k} |\vec{x}_t| + k|\vec{x}_{k+1}| \right) \\ &\leq \frac{1}{k(k+1)}(k \cdot d + k \cdot d) = \frac{2d}{k+1} \longrightarrow 0 \text{ as } k \to \infty. \end{aligned}$$

Hence the sequence $(\vec{z}_k)_{k\geq 1}$ is a Cauchy sequence and thus converges to some $\vec{z}^* \in \mathbb{R}^n$. We will show that $\vec{z}^*$ is the desired fixed point.

For all $t$, the polyhedron $Q := \{\binom{\vec{x}}{\vec{x}'} \mid A\binom{\vec{x}}{\vec{x}'} \leq b\}$ contains $\binom{\vec{x}_t}{\vec{x}_{t+1}}$ and is convex. Therefore for all $k \geq 1$,

$$\frac{1}{k}\sum_{t=1}^{k}\binom{\vec{x}_t}{\vec{x}_{t+1}} \in Q.$$

Together with

$$\binom{\vec{z}_k}{\frac{k+1}{k}\vec{z}_{k+1}} = \frac{1}{k}\binom{\vec{0}}{\vec{x}_1} + \frac{1}{k}\sum_{t=1}^{k}\binom{\vec{x}_t}{\vec{x}_{t+1}}$$

we infer

$$\left(\binom{\vec{z}_k}{\frac{k+1}{k}\vec{z}_{k+1}} - \frac{1}{k}\binom{\vec{0}}{\vec{x}_1}\right) \in Q,$$

and since $Q$ is closed we have

$$\binom{\vec{z}^*}{\vec{z}^*} = \lim_{k\to\infty}\left(\binom{\vec{z}_k}{\frac{k+1}{k}\vec{z}_{k+1}} - \frac{1}{k}\binom{\vec{0}}{\vec{x}_1}\right) \in Q. \quad \blacktriangleleft$$

Because fixed points give rise to trivial geometric nontermination arguments, we can derive a criterion for the existence of geometric nontermination arguments from Lemma 7.

▶ **Corollary 8.** *If the linear loop program $P = (true, \text{LOOP})$ has a bounded infinite execution, then it has a geometric nontermination argument.*

**Proof.** By Lemma 7 there is a fixed point $\vec{x}^*$ such that $(\vec{x}^*, \vec{x}^*) \in \text{LOOP}$. We choose $\vec{x}_1 = \vec{x}^*$, $\vec{y} = \vec{0}$, and $\lambda = 1$, which satisfies (point) and (ray) and thus is a geometric nontermination argument for $P$. ◀

▶ **Example 9.** Note that according to our definition of a linear lasso program, the relation LOOP is a topologically closed set. If we allowed the formula defining LOOP to also contain strict equalities, Lemma 7 no longer holds: the following program is nonterminating and has a bounded infinite execution, but it does not have a fixed point. However, the topological closure of the relation LOOP contains the fixed point $x^* = 0$.

```
while (x > 0):
    x := 1/2 · x;
```

# 5 Discussion

## 5.1 Recurrence Sets

Nontermination arguments related to ours are *recurrence sets* [3, 5]. A recurrence set $S$ is a set of states such that

- at least one state of $S$ is in the range of STEM, i.e.

$$\exists\vec{x}, \vec{x}'.(\vec{x}, \vec{x}') \in \text{LOOP} \wedge \vec{x'} \in S, \text{ and}$$

- for each state in $S$ there is at least one LOOP-successor that is in $S$, i.e.,

$$\forall\vec{x}.\vec{x} \in S \to \exists\vec{x}'(\vec{x}, \vec{x}') \in \text{LOOP}.$$

If we restrict the form of $S$ to a convex polyhedron, we can encode its existence using algebraic constraints [3, 5] and hence decide the existence of such a recurrence set. However these algebraic constraints are not easy to solve; they contain nonlinear arithmetic and quantifier

alternation that cannot be eliminated with Farkas lemma if the program is nondeterministic. In contrast to these constraints, our constraints (init), (point), and (ray) contain at most one nonlinear term for each dimension of the state space.

However, recurrence sets are more general nontermination arguments than geometric nontermination arguments as shown by the following lemma.

▶ **Lemma 10.** *Let $P = (\text{STEM}, \text{LOOP})$ be a linear lasso program and $N = (\vec{x}_0, \vec{x}_1, \vec{y}, \lambda)$ be a geometric nontermination argument for $P$. The following set $S$ is a recurrence set for $P$.*

$$S = \left\{ \vec{x}_1 + \sum_{i=0}^{t} \lambda^i \vec{y} \mid t \in \mathbb{N} \right\}$$

**Proof.** The state $\vec{x}_1$ is in the range of STEM by (init). Furthermore, for $\vec{x}_1 + \sum_{i=0}^{t} \lambda^i \vec{y} \in S$, $\vec{x}_1 + \sum_{i=0}^{t+1} \lambda^i \vec{y} \in S$ and $(\vec{x}_1 + \sum_{i=0}^{t} \lambda^i \vec{y},\ \vec{x}_1 + \sum_{i=0}^{t+1} \lambda^i \vec{y}) \in \text{LOOP}$ according to the proof of Theorem 4. ◀

Furthermore, for every geometric nontermination argument $N = (\vec{x}_0, \vec{x}_1, \vec{y}, \lambda)$ there exists a recurrence set $S$ that is a polyhedron.

$$S = \{\vec{x} \in \mathbb{R}^n \mid \vec{y}^T(\vec{x} - \vec{x}_1) \geq 0 \ \wedge \bigwedge_{i \in I} \vec{z}_i^T(\vec{x} - \vec{x}_1) = 0\},$$

where $(\vec{z}_i)_{i \in I}$ is a span of the vector space orthogonal to $\vec{y}$. (For $\lambda < 1$ we need to add the additional constraint $\vec{y}^T(\vec{x} - \vec{x}_1) \leq \vec{y}^T(\vec{x}_1 + \frac{1}{1-\lambda}\vec{y})$.)

## 5.2 Integers vs. Reals

A nonterminating program over the reals may terminate over the integers. If we restrict the states of the linear lasso program to integer-valued vectors,then Theorem 4 only holds if we restrict the values for the variables $\vec{x}_0, \vec{x}_1, \vec{y}$, and $\lambda$ in the constraints (init), (point), and (ray) to integers. Satisfiability of nonlinear arithmetic over the integers is undecidable and we do not know if our constraints fall into a decidable subclass of this problem. However, we may fix the value of $\lambda$ in advance to a finite set of values. If we do so, we do not have completeness (we may not find every geometric nontermination argument) but we obtain linear arithmetic constraints, which can be solved efficiently.

## References


1 Mark Braverman. Termination of integer linear programs. In *CAV*, pages 372–385. Springer, 2006.

2 Hong-Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. Proving nontermination via safety. In *TACAS*, 2014.

3 Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *POPL*, pages 147–158, 2008.

4 Dejan Jovanović and Leonardo De Moura. Solving non-linear arithmetic. In *IJCAR*, pages 339–354. Springer, 2012.

5 Andrey Rybalchenko. Constraint solving for program verification theory and practice by example. In *CAV*, pages 57–71. Springer, 2010.

6 A. Tarski. A decision method for elementary algebra and geometry. Technical report, RAND Corporation, 1951.

7 Ashish Tiwari. Termination of linear programs. In *CAV*, pages 70–82. Springer, 2004.

# On Improving Termination Preservability of Transformations from Procedural Programs into Rewrite Systems by Using Loop Invariants

**Naoki Nishida and Takumi Kataoka**

**Graduate School of Information Science, Nagoya University**
**Furo-cho, Chikusa-ku, 4648603 Nagoya, Japan**
nishida@is.nagoya-u.ac.jp
kataoka@trs.cm.is.nagoya-u.ac.jp

**Abstract**

Recently, to analyze procedural programs by using techniques in the field of term rewriting, several transformations of a program into a rewrite system have been developed. Such transformations are basically complete in the sense of computation, and e.g., termination of the rewrite system ensures termination of the program. However, in general, termination of the program is not preserved by the transformations and, thus, the preservation of termination is a common interesting problem. In this paper, we discuss the improvement of a transformation from a simple procedural program over integers into a constrained term rewriting system by appending loop invariants to loop conditions of "while" statements so as to preserve termination as much as possible.

## 1 Introduction

Recently, program verification methods for procedural programs using term rewriting techniques have been developed e.g., [8, 12, 16, 5]. In these methods, to verify a property (e.g., termination), a program is transformed into a rewrite system, and then the rewrite system is verified instead of the program. A common problem in these methods is that in general they do not preserve termination of the program when transforming it into a rewrite system. The transformed rewrite system covers all computations of the program, but can reduce a term which does not correspond to any intermediate state for computation of the program.

This problem is crucial in verifying the equivalence of functions by using *inductive theorem proving* methods [15, 6], which are based on *rewriting induction* [13]. Rewriting induction requires a rewrite system to be terminating. For this reason, when termination of a program is not preserved in transforming it, we cannot apply the methods based on rewriting induction to the verification of the transformed rewrite system.

In this paper, we aim at improving transformations of simple procedural programs over integers, which are based on "while" programs, into rewrite systems so as to preserve termination as much as possible. To this end, We first introduce the transformation in [8] by using an example, for which the transformation does not preserve termination. Then, we show that appending a loop invariant to the loop condition of a "while" statement in the program makes the transformation preserve termination of the program. Finally, we briefly introduce how to automatically obtain the loop invariant by using an existing technique. Note that a preliminary version of this work is [10].

We deal with constrained term rewriting systems [8, 2, 14], but the idea in this paper is applicable to other frameworks (transformations and their resulting rewrite systems), e.g., *integer rewriting systems* [7, 16] and *logically constrained term rewriting systems* [11]. The improvement is useful to prove termination of a procedural program.

One may think that we can prove correctness of programs if we find loop invariants which are useful to preserve termination in applying the transformation. However, this would not be sufficient for the purpose of proving equivalence of two functions defined by programs.

This paper is organized as follows. Section 2 briefly introduces constrained TRSs. Section 3 introduces the transformation in [8] by an example. Section 4 shows an idea for improving the transformation by using loop invariants. Section 5 concludes this paper.

## 2 Preliminaries

In this section, we briefly recall some basic notions and notations of *constrained TRSs* [8, 2, 14]. We assume familiarity of readers with the basic notions and notations of term rewriting (see [1]). Throughout this paper, we use $\mathcal{V}$ as a countably infinite set of variables.

We restrict constrained TRSs to be over a fixed interpretable signature for integers. We use function symbols $\mathsf{0}$ (a constant), $\mathsf{s}$ (unary), $\mathsf{p}$ (unary), and $\mathsf{plus}$ (binary), which are interpreted to in the usual way: $\mathsf{0}$ means 0, $\mathsf{s}(x)$ means $x+1$, $\mathsf{p}(x)$ means $x-1$, and $\mathsf{plus}(x,y)$ means $x+y$. In the following, we write $t_1+t_2$ instead of $\mathsf{plus}(t_1,t_2)$. Moreover, we use the usual comparison operators (e.g., $=$, $\leq$, $<$) as predicates with the usual semantics, and use quantifier-free formulas over the interpreted symbols, the predicates, and variables, as *constraints*.

Let $\mathcal{F}$ be a signature which is basically specified by users. A *constrained rewrite rule* is a triple $(l, r, \phi)$, written as $l \to r \ [\![\phi]\!]$, such that $l$ and $r$ are terms over $\mathcal{F}$ and the interpreted symbols, $l$ is not a variable, $\phi$ is a satisfiable constraint, and $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r) \cup \mathcal{V}ar(\phi)$. A *constrained term rewriting system* (constrained TRS) is a finite set $\mathcal{R}$ of constrained rewrite rules. We omit the following rules for interpreted symbols from constrained TRSs shown in this paper:

$$\{\ \mathsf{s}(\mathsf{p}(x)) \to x,\quad \mathsf{p}(\mathsf{s}(x)) \to x,\quad \mathsf{0}+y \to y,\quad \mathsf{s}(x)+y \to \mathsf{s}(x+y),\quad \mathsf{p}(x)+y \to \mathsf{p}(x+y)\ \}$$

The rewrite relation $\to_{\mathcal{R}}$ of $\mathcal{R}$ is defined as follows: $\to_{\mathcal{R}} = \{(C[l\sigma], C[r\sigma]) \mid l \to r \ [\![\phi]\!] \in \mathcal{R},\ \phi\sigma \text{ has no symbol in } \mathcal{F},\ \phi\sigma \text{ is valid}\}$. A term $t$ is called *terminating* if there is no infinite rewrite sequence of $\to_{\mathcal{R}}$ which is starting from $t$. $\mathcal{R}$ is called *terminating* if every term is terminating.

## 3 Transforming C Programs over Integers into Constrained TRSs

In this section, we briefly introduce a transformation of simple procedural programs over integers into constrained TRSs [8].

We deal with programs of the following form:

$$\texttt{int}\ f\texttt{(int}\ x_1\texttt{, ..., int}\ x_n\texttt{)\{}\ D\texttt{;}\ S\texttt{;}\ \texttt{return}\ e\texttt{;}\ \texttt{\}}$$

where $D$ is a sequence of declarations for local variables with initialization, $S$ is a so-called "while" program (having assignments, "while", and "if" statements), $e$ is an expression, and the code is compiled successfully as a C program.

▶ **Example 1.** The program `sum` illustrated in Listing 1 is the one we deal with.

The transformation in [8] (cf. [5, 16]) converts programs in a natural way: constrained rewrite rules are generated so as to represent control flows of programs, and loop or branch conditions of "while" and "if" statements are added to rewrite rules as constraints. The transformation preserves termination of a program w.r.t. terms corresponding to calls of functions defined in the program.

**Listing 1** A program computing the sum from 0 to $x$

```
int sum(int x){
  int i=0, z=0;
  if(x > 0)
    while(i != x){
      i = i+1;
      z = z+i;
    }
  return z;
}
```

▶ **Example 2.** Consider the program `sum` in Listing 1 again. This program is transformed into the following constrained TRS:

$$\mathcal{R}_{\texttt{sum}} = \left\{ \begin{array}{ll} \mathsf{sum}(x) \to \mathsf{u}_1(x,0,0), & \\ \mathsf{u}_1(x,i,z) \to \mathsf{u}_2(x,i,z)\ [\![x > 0]\!], & \mathsf{u}_2(x,i,z) \to \mathsf{u}_2(x,\mathsf{s}(i),z+\mathsf{s}(i))\ [\![i \neq x]\!], \\ \mathsf{u}_1(x,i,z) \to z\ [\![\neg\ x > 0]\!], & \mathsf{u}_2(x,i,z) \to z\ [\![\neg\ i \neq x]\!] \end{array} \right\}$$

The function symbols $\mathsf{u}_1$ and $\mathsf{u}_2$ represent the "if" and "while" statements in the program `sum`. All the terms of the form $\mathsf{sum}(n)$ with an interpreted term $n$ are terminating, but $\mathcal{R}_{\texttt{sum}}$ is not terminating. For example, $\mathsf{u}_2(0,\mathsf{s}(0),0)$ which is not reachable from any term of the form $\mathsf{sum}(n)$ is not terminating.

## 4 Improving Preservation of Termination

In this section, we improve termination preservability of the transformation by appending loop invariants to loop conditions of "while" statements.

Loop invariants are constraints which are satisfied whenever the bodies of "while" statements start to be evaluated. Thanks to this feature of loop invariants, appending a loop invariant to the corresponding loop condition does not change the computation of programs at all.

▶ **Example 3.** Consider the program `sum` in Listing 1 again. By using the method in [3] with the SMT solver Z3 [4], we obtain the constraint "`x > i`" as a loop invariant of the "while" statement in the program. Thus, by appending the loop invariant to the loop condition "`i != x`" of the "while" statement, we obtain the equivalent program illustrated in Listing 2. The modified program is transformed into the following constrained TRS:

$$\mathcal{R}'_{\texttt{sum}} = \left\{ \begin{array}{lll} \mathsf{sum}(x) \to \mathsf{u}_1(x,0,0), & & \\ \mathsf{u}_1(x,i,z) \to \mathsf{u}_2(x,i,z)\ [\![x > 0]\!], & \mathsf{u}_2(x,i,z) \to \mathsf{u}_2(x,\mathsf{s}(i),z+\mathsf{s}(i))\ [\![i \neq x \wedge x > i]\!], \\ \mathsf{u}_1(x,i,z) \to z\ [\![\neg\ x > 0]\!], & \mathsf{u}_2(x,i,z) \to z\ [\![\neg(i \neq x \wedge x > i)]\!] \end{array} \right\}$$

In this case, $\mathcal{R}'_{\texttt{sum}}$ is terminating, and thus, termination of the program `sum` in Listing 2 is preserved by the transformation. Note that the termination was proved by the implementation of the technique proposed by Sakata et al [14]. Moreover, the termination prover AProVE 1.2 [9] succeeded in proving termination of the following *integer term rewriting system* [7] which corresponds to $\mathcal{R}'_{\texttt{sum}}$:

$$\left\{ \begin{array}{lll} \mathsf{sum}(x) \to \mathsf{u}_1(x,0,0,x > 0), & & \\ \mathsf{u}_1(x,i,z,\mathsf{true}) \to \mathsf{u}_2(x,i,z,x \neq i \wedge x > i), & & \mathsf{u}_1(x,i,z,\mathsf{false}) \to z, \\ \mathsf{u}_2(x,i,z,\mathsf{true}) \to \mathsf{u}_2(x,i+1,z+i+1,x \neq i+1 \wedge x > i+1), & & \mathsf{u}_2(x,i,z,\mathsf{false}) \to z \end{array} \right\}$$

**Listing 2** A variant of the program `sum` where the loop invariant is introduced

```
int sum(int x){
  int i=0, z=0;
  if(x > 0)
    while(i != x && x > i){
      i = i+1;
      z = z+i;
    }
  return z;
}
```

We summarize the idea explained above as follows:

1. we generate a loop invariant for each "while" statement in the program;
2. we append the loop invariant to the corresponding loop condition;
3. we apply the transformation to the modified program, getting a constrained TRS.

Note that not for all terminating programs, the transformed rewrite system is terminating.

## 5 Conclusion

In this paper, by using an example, we showed the idea for improving preservation of termination in applying transformations from simple procedural programs over integers into constrained TRSs. However, this idea does not always work as we expected. For example, `true` is a trivial loop invariant which does not improve anything. For this reason, the improvement relies on the power of methods to generate loop invariants.

One may think that we do not have to append a loop invariant to the negation of the corresponding loop condition, which is coupled with rewrite rules controlling the computation flow of the "while" statements. For example, for the computation of `sum`, we do not need to append `x > i` to $\mathsf{u}_2(x,i,z) \to z \ [\![\neg\ i \neq x]\!]$, i.e., $\mathsf{u}_2(x,i,z) \to z \ [\![\neg\ i \neq x]\!]$ is enough. However, if `x > i` is not appended to $\neg\ i \neq x$, then the rewrite system is not *sufficiently complete w.r.t. interpretable terms*. This can be said of the case that we replace the rule by $\mathsf{u}_2(x,i,z) \to z \ [\![(\neg\ i \neq x) \wedge x > i]\!]$ — since `x > i` is a loop invariant, the invariant holds when leaving out of the "while" loop, and thus, to make the reduction more precise, we may append $x > i$ to $\neg\ i \neq x$. In many methods based on rewriting induction, however, sufficient completeness of a rewrite system is required, and thus, the transformed rewrite system is expected to be sufficient complete. For this reason, we do not employ the more strict rule $\mathsf{u}_2(x,i,z) \to z \ [\![(\neg\ i \neq x) \wedge x > i]\!]$.

As future work, we will make an experiment in order to evaluate how successful the improvement is, and will further improve the preservation of termination.

**Acknowledgement** We thank the anonymous referees for their useful comments to improve the paper.

## References

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

2 A. Bouhoula and F. Jacquemard. Automated induction with constrained tree automata. In *Proceeding of the 4th International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pp. 539–554, Springer, 2008.

**3** M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *Proceedings of the 15th International Conference on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pp. 420–432, Springer, 2003.

**4** L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pp. 337–340, Springer, 2008.

**5** S. Falke and D. Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In *Proceedings of the 22nd International Conference on Automated Deduction*, volume 5663 of *Lecture Notes in Computer Science*, pp. 277–293, Springer, 2009.

**6** S. Falke and D. Kapur. Rewriting induction + linear arithmetic = decision procedure. In *Proceedings of the 6th International Joint Conference on Automated Reasoning*, volume 7364 of *Lecture Notes in Computer Science*, pp. 241–255, Springer, 2012.

**7** C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving termination of integer term rewriting. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications*, volume 5595 of *Lecture Notes in Computer Science*, pp. 32–47, Springer, 2009.

**8** Y. Furuichi, N. Nishida, M. Sakai, K. Kusakari, and T. Sakabe. Approach to procedural-program verification based on implicit induction of constrained term rewriting systems. *IPSJ Transactions of Programming*, 1(2):100–121, 2008 (in Japanese).

**9** J. Giesl, P. Schneider-Kamp, and R. Thiemann. Aprove 1.2: Automatic termination proofs in the dependency pair framework. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pp. 281–286, Springer, 2006.

**10** T. Kataoka, N. Nishida, M. Sakai, T. Sakabe, and K. Kusakari. Use of loop invariants for improving termination preservability of transformations from procedural programs to rewriting systems. In *Record of 2013 Tokai-Section Joint Conference on Electrical and Related Engineering*, no. M2-6, 1 page, 2013 (in Japanese).

**11** C. Kop and N. Nishida. Term rewriting with logical constraints. In *Proceedings of the 9th International Symposium on Frontiers of Combining Systems*, volume 8152 of *Lecture Notes in Computer Science*, pp. 343–358, Springer, 2013.

**12** C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of java bytecode by term rewriting. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *LIPIcs*, pp. 259–276, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010.

**13** U. S. Reddy. Term rewriting induction. In *Proceedings of the 10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pp. 162–177, Springer, 1990.

**14** T. Sakata, N. Nishida, and T. Sakabe. On proving termination of constrained term rewrite systems by eliminating edges from dependency graphs. In *Proceedings of the 20th International Workshop on Functional and Constraint Logic Programming*, volume 6816 of *Lecture Notes in Computer Science*, pp. 138–155, Springer, 2011.

**15** T. Sakata, N. Nishida, T. Sakabe, M. Sakai, and K. Kusakari. Rewriting induction for constrained term rewriting systems. *IPSJ Transactions of Programming*, 2(2):88–96, 2009 (in Japanese).

**16** G. Vidal. Closed symbolic execution for verifying program termination. In *Proceedings of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 34–43, IEEE Computer Society, 2012.

# Non-termination of Dalvik bytecode *via* compilation to CLP

**Étienne Payet and Fred Mesnard**

**Université de La Réunion, EA2525-LIM**
**Saint-Denis de La Réunion, F-97490, France**
{etienne.payet,frederic.mesnard}@univ-reunion.fr

**Abstract**

We present a set of rules for compiling a Dalvik bytecode program into a logic program with array constraints. Non-termination of the resulting program entails that of the original one, hence the techniques we have presented before for proving non-termination of constraint logic programs can be used for proving non-termination of Dalvik programs.

**1998 ACM Subject Classification** D.2.4 Software/Program Verification (Formal methods), F.3.1 Specifying and Verifying and Reasoning about Programs (Mechanical verification), F.3.2 Semantics of Programming Languages (Program analysis)

**Keywords and phrases** Non-Termination, Android, Dalvik, Constraint Logic Programming

## 1 Introduction

Android is currently the most widespread operating system for mobile devices. Applications running on this system can be downloaded from anywhere, hence reliability is a major concern for its users. In this paper, we consider applications that may run into an infinite loop, which may cause a resource exhaustion, for instance the battery if the loop continuously uses a sensor as the GPS. Android programs are written in Java and compiled to the Google's Dalvik Virtual Machine (DVM) bytecode format [3] before installation on a device. We provide a set of rules for compiling a Dalvik bytecode program into a constraint logic program [5]. Non-termination of the resulting program entails that of the original one, hence the technique we have presented before [6] for proving non-termination of constraint logic programs can be used for proving non-termination of Dalvik programs. We model the memory and the objects it contains with *arrays*, so we compile Dalvik programs to logic programs with array constraints and we consider the theory of arrays presented in [1].

## 2 The Dalvik Virtual Machine

We briefly describe the operational semantics of the DVM (see [3] for a complete description). Unlike the JVM which is stack-based, the DVM is register-based. Each method uses its own array of registers and invoked methods do not affect the registers of invoking methods. The number of registers used by a method is statically known. At the beginning of an execution, the $N$ arguments to a method land in its last $N$ registers and the other registers are initialized to 0. Many Dalvik bytecode instructions are similar, so we concentrate on a restricted set which exemplifies the operations that the DVM performs.

- *const* $d, c$ Move constant $c$ into register $d$ (*i.e.,* the register at index $d$ in the array of registers of the method where this instruction occurs).
- *move* $d, s$ Move the content of register $s$ into register $d$.
- *add* $d, s, c$ Store the sum of the content of register $s$ and constant $c$ into register $d$.

- *if-lt* $i, j, q$ If the content of register $i$ is less than the content of register $j$ then jump to program point $q$, otherwise execute the immediately following instruction.
- *goto* $q$ Jump to program point $q$.
- *invoke* $S, m$ where $S = s_0, s_1, \ldots, s_p$ is a sequence of register indexes and $m$ is a method. The content $r^{s_0}$ of register $s_0$, $\ldots$, $r^{s_p}$ of register $s_p$ are the *actual parameters* of the call. Value $r^{s_0}$ is called *receiver* of the call and must be 0 (the equivalent of `null` in Java) or a reference to an object $o$. In the former case, the computation stops with an exception. Otherwise, a lookup procedure is started from the class of $o$ upwards along the superclass chain, looking for a method with the same signature as $m$. That method is run from a state where its last registers are bound to $r^{s_0}, r^{s_1}, \ldots, r^{s_p}$.
- *return* Return from a void method.
- *new-instance* $d, \kappa$ Move a reference to a new object of class $\kappa$ into register $d$.
- *iget* $d, i, f$ (resp. *iput* $s, i, f$) The content $r^i$ of register $i$ must be 0 or a reference to an object $o$. If $r^i$ is 0, the computation stops with an exception. Otherwise, $o(f)$ (the value of field $f$ of $o$) is stored into register $d$ (resp. the content of register $s$ is stored into $o(f)$).

## 3 Compilation to CLP clauses

We model a *memory* as a pair $(a, i)$ where $a$ is an array of *objects* and $i$ is the index into this array where the next insertion will take place. An *object* $o$ is an array of terms of the form $[w, f_1(v_1), \ldots, f_n(v_n)]$ where $w$ is the name of the class of $o$, $f_1$, $\ldots$, $f_n$ are the names of the fields defined in this class and $v_1$, $\ldots$, $v_n$ are the current values of these fields in $o$. So, the first component of a memory is an array of arrays of terms and a memory location is an index into this array. Memory locations start at 1 and 0 corresponds to the `null` value.

Our compilation rules are given in Fig. 1–3. We associate a predicate symbol $p_q$ to each program point $q$ of the Dalvik program $P$ under consideration. We generate clauses with constraints on integer and array terms. Our constraint theory combines the theory of integers with that of arrays defined in [1]. Our CLP domain of computation $\mathcal{D}$ (values interpreting constraints) is the union of $\mathbb{Z}$ with the set *Obj* of arrays of terms of the form $f(i)$ where $i$ is an integer and with the set of arrays of elements of *Obj*. The read $a[i]$ returns the value stored at position $i$ of the array $a$ and the write $a\{i \leftarrow e\}$ is $a$ modified so that position $i$ has value $e$. For multidimensional arrays, we abbreviate $a[i] \cdots [j]$ with $a[i, \ldots, j]$.

Each rule considers an instruction *ins* occurring at a program point $q$. We let $\tilde{V} = V_0, \ldots, V_{r-1}$ and $\tilde{V'} = V'_0, \ldots, V'_{r-1}$ be sequences of distinct variables where $r$ is the number of registers used by the method where *ins* occurs. For each $i \in [0, r-1]$, variable $V_i$ (resp. $V'_i$) models the content of register $i$ before (resp. after) executing *ins*. We let $M$ denote the input memory and $M'$ the output memory. So, $\tilde{V}$ and $M$ (or $[A, I]$) in the head of the clauses are input parameters while $M'$ is an output parameter. We let *id* denote the sequence $(V'_0 = V_0, \ldots, V'_{r-1} = V_{r-1})$ and $id_{-i}$ (where $i \in [0, r-1]$) the sequence $(V'_0 = V_0, \ldots, V'_{i-1} = V_{i-1}, V'_{i+1} = V_{i+1} \ldots, V'_{r-1} = V_{r-1})$. By $|\tilde{X}|$ we mean the length of sequence $\tilde{X}$. For any method $m$, $q_m$ is the program point where $m$ starts, $reg(m)$ is the number of registers used by $m$ and $sign(m)$ is the set of all the methods with the same signature as $m$.

Some compilation rules are rather straightforward. For instance, *const* $d, c$ moves constant $c$ into register $d$, so in Fig. 1 the output register variable $V'_d$ is set to $c$ while the other register variables remain unchanged (modelled with $id_{-d}$). Rules for *move*, *add* and *goto* are similar. In Fig. 2, we consider method calls. The instruction *invoke* $s_0, \ldots, s_p, m$ is compiled into a set of clauses (one for each method with the same signature as $m$) which

$$\frac{const\ d, c}{p_q(\tilde{V}, M, M') \leftarrow \{V'_d = c\} \cup id_{-d},\ p_{q+1}(\tilde{V}', M, M')} \tag{1a}$$

$$\frac{if\text{-}lt\ i, j, q'}{\begin{array}{ll}\{ & p_q(\tilde{V}, M, M') \leftarrow \{V_i < V_j\} \cup id,\ p_{q'}(\tilde{V}', M, M'), \\ & p_q(\tilde{V}, M, M') \leftarrow \{V_i \geq V_j\} \cup id,\ p_{q+1}(\tilde{V}', M, M') \quad \}\end{array}} \tag{1b}$$

**Figure 1** Compilation of some simple Dalvik instructions.

$$\frac{invoke\ s_0, \ldots, s_p, m}{\left\{\begin{array}{l|l} p_q(\tilde{V}, M, M') \leftarrow \{V_{s_0} > 0\} \cup id, & m' \in sign(m) \\ \qquad lookup_P(M, V_{s_0}, m, q_{m'}), & \text{and } \tilde{X}_{m'} = 0, \ldots, 0, V_{s_0}, \ldots, V_{s_p} \\ \qquad p_{q_{m'}}(\tilde{X}_{m'}, M, M_1), & \text{with } |\tilde{X}_{m'}| = reg(m') \\ \qquad p_{q+1}(\tilde{V}', M_1, M') & \end{array}\right\}} \tag{2a}$$

$$\frac{return}{p_q(\tilde{V}, M, M') \leftarrow \{M' = M\}} \tag{2b}$$

**Figure 2** Compilation of some Dalvik instructions related to method calls.

impose that $V_{s_0}$ (the receiver of the call) is a non-`null` location (*i.e.,* $V_{s_0} > 0$). Therefore, if $V_{s_0} \leq 0$, the execution of the generated CLP program fails, as the original Dalvik program. If $V_{s_0} > 0$, the lookup procedure begins. For each $m' \in sign(m)$, this is modelled with the call $lookup_P(M, V_{s_0}, m, q_{m'})$ which starts from the class of the object at location $V_{s_0}$ in memory $M$ and searches for the closest method $m''$ with the same signature as $m$ upwards along the superclass chain. If $m'' = m'$, this call succeeds, otherwise it fails. Then, $m'$ is executed, modelled with $p_{q_{m'}}(\tilde{X}_{m'}, M, M_1)$, with some registers $\tilde{X}_{m'}$ initialized as expected. When the execution of $m'$ has finished, control jumps to the following instruction (*i.e.,* $p_{q+1}(\tilde{V}', M_1, M')$). In Fig. 3, we consider some memory-related instructions that we compile to clauses with array constraints.

▶ **Theorem 1.** *Let $P$ be a Dalvik bytecode program and $P_{CLP}$ its CLP compilation. If there is a computation $p_{q_0}p_{q_1}\ldots$ in $P_{CLP}$ then there is an execution $q_0q_1\ldots$ of $P$.*

More precisely, if there is a finite (resp. infinite) computation in $P_{CLP}$ starting from a query $p_{q_0}(\tilde{v}, [a, i], M')$ (where $\tilde{v}$, $a$ and $i$ are values in $\mathcal{D}$ and $M'$ is an output variable), then there is a finite (resp. infinite) execution of $P$, using the same program points, starting from values corresponding to $\tilde{v}$ and $a$ in the DVM registers and memory.

## 4 Non-termination inference

The following proposition is a CLP reformulation of a result presented in [4].

▶ **Proposition 2.** *Let $r = p(\tilde{x}) \leftarrow c, p(\tilde{y})$ and $r' = p'(\tilde{x}') \leftarrow c', p(\tilde{y}')$ be some clauses. Suppose there exists a set $\mathcal{G}$ such that formulæ $\left[\forall\tilde{x}\exists\tilde{y}\ \tilde{x} \in \mathcal{G} \Rightarrow (c \wedge \tilde{y} \in \mathcal{G})\right]$ and $\left[\exists\tilde{x}'\exists\tilde{y}'\ c' \wedge \tilde{y}' \in \mathcal{G}\right]$ are true. Then, $p'$ has an infinite computation in $\{r, r'\}$.*

$$\frac{\begin{array}{c}\mathit{new\text{-}instance}\ d, \kappa \\ w \text{ is the name of class } \kappa \text{ and } f_1, \ldots, f_n \text{ are the names of the fields defined in } \kappa\end{array}}{\begin{array}{l}p_q(\tilde{V}, [A, I], M') \leftarrow \big\{O[0] = w,\ O[1] = f_1(0),\ \ldots,\ O[n] = f_n(0), \\ \qquad A_1 = A\{I \leftarrow O\},\ V'_d = I,\ I_1 = I + 1\big\} \cup id_{-d},\ p_{q+1}(\tilde{V}', [A_1, I_1], M')\end{array}} \tag{3a}$$

$$\frac{\mathit{iget}\ d, i, f}{p_q(\tilde{V}, [A, I], M') \leftarrow \big\{V_i > 0,\ A[V_i, F] = f(V'_d)\big\} \cup id_{-d},\ p_{q+1}(\tilde{V}', [A, I], M')} \tag{3b}$$

$$\frac{\mathit{iput}\ s, i, f}{\begin{array}{l}p_q(\tilde{V}, [A, I], M') \leftarrow \big\{V_i > 0,\ O = A[V_i],\ O[F] = f(X),\ O_1 = O\{F \leftarrow f(V_s)\}, \\ \qquad A_1 = A\{V_i \leftarrow O_1\}\big\} \cup id,\ p_{q+1}(\tilde{V}', [A_1, I], M')\end{array}} \tag{3c}$$

**Figure 3** Compilation of some memory-related instructions.

Consider the Android program in Fig. 4, with the Java syntax on the left and the corresponding Dalvik bytecode $P$ on the right, where `v0`, `v1`, ... denote registers 0, 1, ... Method `loop` in class `MyActivity` is called when the user taps a button displayed by the application. Execution of this method does not terminate because in the call to `m`, the objects `o1` and `o2` are aliased and therefore by decrementing `x.i` we are also decrementing `this.i` in the loop of method `m`. We get the following clauses for program points 0 and 14:

$$p_0(\tilde{V}, [A, I], M') \leftarrow \{A[V_1, F] = i(V'_0)\} \cup id_{-0},\ p_1(\tilde{V}', [A, I], M')$$

$$\begin{array}{l}p_{14}(\tilde{V}, M, M') \leftarrow \{V_0 > 0\} \cup id,\ \mathit{lookup}_P(M, V_0, \texttt{Loops->m(ILoops)V}, 0), \\ \qquad p_0(0, V_0, V_2, V_1, M, M_1),\ p_{15}(\tilde{V}', M_1, M')\end{array}$$

Let $P_{CLP}$ denote the CLP program resulting from the compilation of $P$. The set of *binary unfoldings* [2] of $P_{CLP}$ contains the following clauses

$$\begin{array}{ll}r: & p_0(\tilde{V}, [A, I], M') \leftarrow \big\{V_1 > 0,\ O = A[V_1],\ O[F] = i(X),\ X < V_2, \\ & \qquad O_1 = O\{F \leftarrow i(X + 1)\},\ A_1 = A\{V_1 \leftarrow O_1\}, \\ & \qquad V_3 > 0,\ O' = A_1[V_3],\ O'[F'] = i(X'),\ V'_0 = X' - 1, \\ & \qquad O'_1 = O'\{F' \leftarrow i(V'_0)\},\ A_2 = A_1\{V_3 \leftarrow O'_1\}\big\} \cup id_{-0},\ p_0(\tilde{V}', [A_2, I], M') \\ r': & p_{10}(\tilde{V}, [A, I], M') \leftarrow \{O[0] = \mathit{loops},\ O[1] = i(0),\ A_1 = A\{I \leftarrow O\}, \\ & \qquad I_1 = I + 1,\ I > 0\},\ p_0(0, I, 2, I, [A_1, I_1], M_1)\end{array}$$

where $r$ corresponds to the path $0 \to 1 \to 3 \to 4 \to \cdots \to 9 \to 0$ and $r'$ to the path $10 \to 11 \to 12 \to 13 \to 14 \to 0$ in $P$. In $r'$, $O$ corresponds to both $o_1$ and $o_2$, which expresses that $o_1$ and $o_2$ are aliased. Note that $I$, the address of $O$, is passed to $p_0$ both as second and fourth parameter, which corresponds in $r$ to $V_1$ (`this` in method `m`) and $V_3$ (`x` in `m`). Moreover, when $V_1 = V_3$ in $r$, we have $O' = O_1$, $F' = F$ and $X' = X + 1$, hence $V'_0 = X' - 1 = X$. Therefore, we have $O'_1 = O$, so $A_2 = A$. The logical formulæ of Proposition 2 are true for the set $\mathcal{G} = \{(\tilde{v}, \mathit{mem}, \mathit{mem}') \in \mathcal{D}^3 | v_1 = v_3\}$. Hence, $p_{10}$ has an infinite computation in $\{r, r'\}$, which implies [2] that $p_{10}$ has an infinite computation in $P_{CLP}$. So by Theorem 1, $P$ has an infinite execution from program point 10.

```
public class Loops {
  int i;
  public void m(int n, Loops x) {
    while (this.i < n) {
      this.i++;
      x.i--;
    }
  }
}
```

```
.method public m(ILoops)V
    .registers 4
0:  iget v0, v1, Loops->i:I
1:  if-lt v0, v2, 3
2:  return-void
3:  iget v0, v1, Loops->i:I
4:  add-int/lit8 v0, v0, 0x1
5:  iput v0, v1, Loops->i:I
6:  iget v0, v3, Loops->i:I
7:  add-int/lit8 v0, v0, -0x1
8:  iput v0, v3, Loops->i:I
9:  goto 0
.end method
```

```
public class MyActivity extends Activity {
  ...
  public void loop(View v) {
    Loops o1 = new Loops();
    Loops o2 = o1;
    o1.m(2, o2);
  }
  ...
}
```

```
.method public loop(Landroid/view/View;)V
    .registers 5
10:  new-instance v0, Loops
11:  invoke-direct {v0}, Loops-><init>()V
12:  move-object v1, v0
13:  const/16 v2, 0x2
14:  invoke-virtual {v0, v2, v1}, Loops->m(ILoops)V
15:  return-void
.end method
```

**Figure 4** The non-terminating method `loop` is called when the user taps a button.

## 5 Future Work

We plan to implement the technique described above and to write a solver for array constraints. Currently, our compilation rules only consider the operational semantics of Dalvik, a part of the Android platform. We also plan to extend them by considering the operational semantics of other components of Android, for instance *activities* that we have studied in [7].

## References


1 A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In E. A. Emerson and K. S. Namjoshi, editors, *Proc. of VMCAI'06*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.

2 M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.

3 Dalvik docs mirror. `http://www.milk.com/kodebase/dalvik-docs-mirror/`.

4 A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving non-termination. In G. C. Necula and P. Wadler, editors, *Proc. of POPL'08*, pages 147–158. ACM Press, 2008.

5 J. Jaffar, M. J. Maher, K. Marriott, and P. J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.

6 É. Payet and F. Mesnard. A non-termination criterion for binary constraint logic programs. *Theory and Practice of Logic Programming*, 9(2):145–164, 2009.

7 É. Payet and F. Spoto. An operational semantics for Android activities. In W.-N. Chin and J. Hage, editors, *Proc. of PEPM'14*, pages 121–132. ACM, 2014.

# Specifying and verifying liveness properties of QLOCK in CafeOBJ

**Norbert Preining, Kazuhiro Ogata, and Kokichi Futatsugi**

**Japan Advanced Institute of Science and Technology**
**Research Center for Software Verification**
**Nomi, Ishikawa, Japan**
{preining,ogata,futatsugi}@jaist.ac.jp

**Abstract**

We provide an innovative development of algebraic specifications and proof scores in CafeOBJ of QLOCK's safety and liveness properties. The particular interest of the development is two-fold: Firstly, it extends base specifications in order-sorted and rewriting logics to a meta-level, which requires behavioral logic, thus using the three logics together to achieve the proofs. Secondly, we use a search predicate and covering state patterns that allow us to prove the validity of a property over all possible one-step transitions, by which safety and liveness properties in the base and meta-level can be proven.

**1998 ACM Subject Classification** D.2.4 Formal methods, Correctness proofs

**Keywords and phrases** fairness, liveness, CafeOBJ, verification, algebraic specification

## 1 Introduction

QLOCK, an abstract version of Dijkstra's binary semaphore, is a protocol to guarantee exclusive access to a resource. Besides the initial specification and verification in CafeOBJ (see for example [2]), it saw implementations in Coq and Maude. Most of these specifications only consider safety properties, in the current case the mutual exclusion property, that no two agents will have access to the resource at the same time. However, liveness properties are normally left open. These properties ensure that 'there is progress'. In our particular case, they ensure that agents do not block out other agents from acquiring access to the resource.

### 1.1 The QLOCK protocol

The QLOCK protocol regulates access of an arbitrary number of agents to a resource by providing a queue (first-in-first-out list). Agents start in the *remainder section*, henceforth indicated by rs. If an agent wants to use the resource, it puts a unique identifier into the queue, and by this transitions into the *waiting section* (ws). In the waiting section, an agent checks the top of the queue. If it is it's unique `Id`, the agent transitions into the *critical section* (cs), during which the agent can use the resource. After having finished with the resource usage, the agent removes the head of the queue and transitions back into rs.

### 1.2 Verification properties

The basic safety property of QLOCK is the *mutual exclusion property* (mp):

▶ Property 1 (mutual exclusion property). At any time, at most one agent is in the critical section.

While this is the most important property for safety concerns, it does not guarantee that an agent wanting to use the resource ever gets the chance to use it. To guarantee this, we

define two liveness properties: The first concerns the transition from ws to cs, and is called the *progress property* (pp). This property has already been discussed in [7] as *lockout freedom property*.

▶ Property 2 (progress property). An agent that has entered into ws (waiting section), i.e., has put his `Id` into the queue, will eventually progress to the top of the queue and gain access to the resource, i.e., transition into cs (critical section).

The last one concerns the transition from rs to ws, called *entrance property* (ep):

▶ Property 3 (entrance property). An agent will eventually transition from rs to ws.

It might sound counter-intuitive that the entrance property should hold for each agent at all times, but what we are effectively proving is that any given arbitrary finite sequence of transitions (where the agent might not want to enter at all) can be extended to an infinite fair transition sequence, and within this transition sequence the property holds. Reflecting back to the finite transition sequence, we obtain that either the agent can enter the queue or the execution stops. Thus, this last property is conceptually different from the first two, as it requires an additional assumption on the fairness of the transition sequence, see Section 3.2.

## 1.3 Observational transition systems

Observational Transition System, henceforth OTS, is a modeling scheme for arbitrary systems that tries to describe the system by a set of observers. Like a state machine, an OTS features transitions, describing the change of observation values due to activity of/in the system.

OTSs have been successfully employed to specify and verify various protocols in CafeOBJ [5]. We are convinced that the usage of observations, that is, not inherent properties of the implementation, but abstract properties of the system exhibited by the change of observations, allow for algebraic specifications with good properties, in particular verification by induction on the reachable state space.

In our specific case, it means that we will describe the system by the observations of the states, i.e., in which section agents are, as well as the state of the queue.

## 1.4 Verification by induction and exhaustive search

While we cannot give a full description of the theory behind verification by induction of an OTS system, the following should explain the basic idea.

Verification of properties of an OTS is often done by induction on reachable states. That is, we show that a certain property (`invprop`) holds in the set of initial states, characterized by `init`. Furthermore, as we proceed through transitions (state changes), the `invprop` is preserved.

But to show liveness properties, considering only invariant properties on states is not enough. We thus use an extended method that does inductive proofs on the reachable state space, and in parallel proves properties (`transprop`) on all transitions between reachable states. To be a bit more specific, assume that $S \Rightarrow SS$ is a transition from one state pattern term $S$ to $SS$. We show that if `invprop`$(S)$ holds, then also `invprop`$(SS)$ (the induction on reachable states), but also that for this transition `transprop`$(S, SS)$ holds.

Both of these are done with CafeOBJ's builtin search predicate, which exhaustively searches and tests all possible transitions from a given state pattern. The concepts introduced here are an extension and generalization of *transition invariants* [8], details in a forthcoming publication.

## 2 Specification

We are using CafeOBJ as specification and verification language. CafeOBJ is a many- and order-sorted algebraic specification language from the OBJ family, related to languages like CASL and Maude. CafeOBJ allows us to have both the specification and the verification in the same language. It is based on powerful logical foundations (order-sorted algebra, hidden algebra, and rewriting logic) with an executable semantics [3, 5, 6].

### 2.1 Specification and verification of invariant properties

As mentioned above, we are building upon a previously obtained specification and verification of QLOCK [2]. We will give only the list of defined modules, and refer the reader to the web page[1] for the full code. We cannot give a full introduction to the CafeOBJ language here, but users acquainted to Maude will find it easy to read the code.

Skipping the definitions of labels, agent identifiers and the queue, the following code specifies agent observers, codifying the agent id and the current section of the agent in one term. The meaning of the term `lb[A]:S` is that the agent `A` is in section `S`:

```
mod! AOB {protecting(LABEL) protecting(AID) [Aob]
  op (lb[_]:_) : Aid Label -> Aob {constr} .  }
```

The state of the whole OTS is described as a pair of a queue and a set of agent observers, where the pairing is achieved by the `$` construct (CafeOBJ allows nearly arbitrary syntax):

```
mod! STATE{ protecting(AID-QUEUE)
  protecting(SET(AOB{sort Elt -> Aob})*{sort Set -> Aobs})
  [State]  op _$_ : Qu Aobs -> State {constr} .  }
```

The transitions are defined by transition rules over state patterns:

```
mod! WaitTrans { protecting(STATE)
trans[wt]: (Q:Qu   $ ((lb[A:Aid]: rs) AS:Aobs))
       => ((Q & A) $ ((lb[A    ]: ws) AS)) .          }
mod! TryTrans { protecting(STATE)
trans[ty]: ((A:Aid & Q:Qu) $ ((lb[A]: ws) AS:Aobs))
        => ((A     & Q)    $ ((lb[A]: cs) AS)) .       }
mod! ExitTrans {protecting(STATE)
trans[ex]: ((A1:Aid & Q:Qu) $ ((lb[A2:Aid]: cs) AS:Aobs))
        =>           (Q     $ ((lb[A2    ]: rs) AS)) . }
```

Based on the above specification, it is possible to provide a *proof score*, i.e., a program in CafeOBJ that verifies the mutual exclusion property mp.

### 2.2 Original progress property

The property that an agent being in the queue, i.e., in ws, will eventually progress to the cs state and gain access to the resource, was originally not part of the verification, but it turned out that it can be shown under the sole assumption that the number of agents is finite (but arbitrary). The core idea is to keep track of the position of an agent in the queue.

During the lift to the meta-level, we will re-prove this fact, and thus will not go into details of the original verification.

[1] http://www.preining.info/blog/cafeobj/

# 3 Going to the meta-level

To verify the last property, ep, operational considerations alone do not suffice. On the level of observers, we cannot guarantee that an agent will ever enter the queue, since we have no control over which transitions are executed by the system. To discuss (verify) this property, we have to assume a certain meta-level property, in this case the fairness of the transition sequence. A similar approach has been taken in [4] for the Alternating Bit Protocol, where fair *event mark streams* are considered.

## 3.1 Relation to other concepts of fairness

The methodology of using OTS in CafeOBJ has been strongly influenced by UNITY [1], which provides an *ensures* operator allowing to model fairness.

Another approach to the concept of fairness is taken by LTL logic [9], where two types of fairness, strong and weak, are considered, referring to *enabled* and *applied* state of transitions.

## 3.2 Transition sequence

Modeling fairness requires recurring to a meta-assumption, namely that the sequence of transitions is fair, i.e., every instance of a transition appears infinitely often in the sequence. In our case we wanted to have a formalization of this meta-assumption that can be expressed with the rewriting logic of CafeOBJ.

The approach we took models transition sequences using behavioral specification with hidden algebra [4], often used to express infinite entities. Note that we are modeling the transition sequence by an infinite stream of agent ids, since the agent uniquely defines the instance of transition to be used, depending on the current state of the agent:

```
mod* TRANSSEQ { protecting(AID)
  *[ TransSeq ]*
  op (_&_) : Aid TransSeq -> TransSeq .  }
```

The transition sequence is then used to model a *meta-state*, i.e., the combination of the original state of the system as specified in the base case, together with the list of upcoming transitions:

```
mod! METASTATE { protecting(STATE + ... )   [MetaState]
  op _^_ : State TransSeq -> MetaState {constr} .  ... }
```

In the same way, transitions from the base case are lifted to the meta-level. We give the meta-version for the transition into ws as an example exhibiting how the state changes while transitions are used up from the transition sequence.

```
mod! MWT {protecting(METASTATE)
trans[meta-wt]:
   ( (Q:Qu   $ ((lb[A:Aid]: rs) AS:Aobs)) ^ (A:Aid & T:TransSeq))
=> ( (Q & A) $ ((lb[A    ]: ws) AS)       ^          T          ) . }
```

To express the fairness condition, we recurred to an equivalent definition, namely that every finite sequence of agent ids can be found as a subsequence of the transition sequence, rephrased here in an indirect way:

```
  eq ( find ( Q:Qu, T:TransSeq ) = empQ ) = false .
```

### 3.3 Verification of properties

By defining (and computing) the wait-time of an agent as the number of meta-transitions (or alternatively, the number of steps in a computation according to the transition sequence), we were able to describe the two liveness properties mentioned above. This was done by proving two invariant properties, namely: (1) If an agent does not change its section during a transition, then its wait-time is decreasing. (2) If the wait-time of an agent reaches 0, then it will change its section. These two invariant properties together show that both the progress property pp and the entrance property ep hold. Note that these properties do not only depend on the states, but properties that hold on the combination of states and transitions. With the same method, we re-proved the *mutual exclusion property*, as well as an extended version of the *progress property*: The original verification requires that there is a finite number of agents. In contrast, if we assume fairness of the execution sequence, even in the case of an infinite number of agents, both *progress* and *entrance* properties hold.

## 4 Discussion and conclusion

We have to note that what we called here *progress property* has already been shown in different settings [7]. The key contribution is the extension to the *entrance property*, meaning that an agent always gets a chance to enter the queue. In addition, we could extended the proof of the progress property to infinitely many agents. The current work also serves as an example of reflecting meta-properties into specifications, allowing for the verification of additional properties.

Assuming a meta-level fairness property to prove liveness properties of the specification might be considered a circular argument, but without regress to meta-level fairness, *no* proof of the entrance property can be achieved. Keeping this in mind, our goal is to provide a reasonable simple and intuitive definition of fairness on the meta-level, that can be used for verification of the necessary properties, similar to any axiomatic approach where trust is based on simple axioms. We are confident that this approach proves successful in other circumstances like the mentioned Alternating Bit Protocol.

## References


1 K.M. Chandy and J. Misra. *Parallel program design - a foundation*. Addison-Wesley, 1989.

2 K. Futatsugi. Generate & check methods for invariant verification in CafeOBJ. JAIST Research Report IS-RR-2013-006, 2013. http://hdl.handle.net/10119/11536.

3 K. Futatsugi, D. Gâinâ, and K. Ogata. Principles of proof scores in CafeOBJ. *Theor. Comput. Sci.*, 464:90–112, 2012.

4 J.A. Goguen and K. Lin. Behavioral verification of distributed concurrent systems with BOBJ. In *QSIC*, pages 216–. IEEE Computer Society, 2003.

5 Sh. Iida, J. Meseguer, and K. Ogata, editors. *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi*, volume 8373 of *LNCS*. Springer, 2014.

6 J. Meseguer. Twenty years of rewriting logic. *J. Log. Algebr. Program.*, 81(7-8):721–781, 2012.

7 K. Ogata and K. Futatsugi. Proof score approach to verification of liveness properties. *IEICE Transactions*, 91-D(12):2804–2817, 2008.

8 A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, pages 32–41. IEEE Computer Society, 2004.

9 V. Rybakov. Linear temporal logic with until and next, logical consecutions. *Annals of Pure and Applied Logic*, 155(1):32–45, 2008.

# Real-world loops are easy to predict : a case study

**Raphael E. Rodrigues[1], Péricles R. O. Alves[2], Fernando M. Q. Pereira[3], and Laure Gonnord[4]**

**1..3 Department of Computer Science, UFMG**
**6627 Antônio Carlos Av, 31.270-010, Belo Horizonte, Brazil**
{raphael,periclesrafael,fernando}@dcc.ufmg.br

**4 Université Lyon1 & Laboratoire d'Informatique du Parallélisme**
**(UMR CNRS / ENS Lyon / UCB Lyon 1 / INRIA)**
**46 allée d'Italie, 69 364 Lyon, France**
laure.gonnord@ens-lyon.fr

## Abstract

In this paper we study the relevance of fast and simple solutions to compute approximations of the number of iterations of loops (*loop trip count*) of imperative real-world programs. The context of this work is the use of these approximations in compiler optimizations: most of the time, the optimizations yield greater benefits for large trip counts, and are either innocuous or detrimental for small ones.

In this particular work, we argue that, although predicting exactly the trip count of a loop is undecidable, most of the time, there is no need to use computationally expensive state-of-the-art methods to compute (an approximation of) it.

We support our position with an actual case study. We show that a fast predictor can be used to speedup the JavaScript JIT compiler of Firefox - one of the most well-engineered runtime environments in use today.

We have accurately predicted over 85% of all the interval loops found in typical JavaScript benchmarks, and in millions of lines of C code. Furthermore, we have been able to speedup several JavaScript programs by over 5%, reaching 24% of improvement in one benchmark.

**1998 ACM Subject Classification** D.3.4 [Processors]: Compilers, F.3.2 [Semantics of Programming Languages]: Program Analysis

**Keywords and phrases** Just-in-time Compilation, Loop Analysis, Trip Count Prediction

## 1 Introduction

The *Trip Count* of a loop determines how many times this loop iterates during the execution of a program. The problem of estimating this value before the loop executes is important in several ways. In critical real-time systems, a (safe) overapproximation of the actual number of loops is required to compute safe Worst-Case Execution Times (WCETs). Therefore, academia has spent substantial effort in the development of accurate methods to estimate the trip count of loops [1, 4, 7]. These usual techniques rely on expensive deduction systems, typically based on SAT solvers, Linear Programming or costly relational analyses.

In compilers however, there is no need for such precise and costly solutions. But there is still a need for a *trip count* analysis, because loops that tend to run for long time are good candidates for unrolling and automatic parallelization. Thus, walking in the opposite direction, we make a case for a fast trip count predictor in this paper.

The contributions of the paper are thus :

- A simple and easy-to-implement heuristic for approximating the number of loops during the execution of a given program;
- The use of this simple heuristic inside the Mozilla Javascript Just in Time Compiler and the application to speed up the execution of some javascript programs from the literature.

With this case study, we want to argue that :

- Apart from classical applications for termination and safe precise predictions of trip counts, compilers and JIT compilers are potential clients for fast (and sometimes non safe) methods;
- Most of the loops of classical benchmarks are easy to predict;
- There is still a place for much clever heuristics or methods for the remaining loops !

## 2 Fast Trip Count Prediction

In this section we explain our motivation to only focus on simple loops, coming from previous work on invariant generation. We also give an algorithm to dynamically infer an approximation of the *trip count* of a given loop just before the actual execution of the first loop in the code (thus, dynamically).

### 2.1 Motivation and inspiration: simple loops invariant generation techniques

Previous work on numerical invariant generation with abstract interpretation techniques [2, 6] led us to ask ourselves how complicated are the loops in actual programs.

In these works, the authors locally use (an adaptation of the so-called) *acceleration techniques* [3] that compute exact fixpoints of (a small class of) numerical transitions in the more general context of abstract interpretation. These *abstract acceleration* techniques have shown their effectiveness in terms of precision at a minimum supplementary cost. The experiments show that in many of the analyzed programs, static analyzers get more precise results just because they are able to precisely deal with loops of the form `for (int i = M; i < N; i++)` (which is an example of an *accelerable loop* [6]) in the most precise way.

Following these experiments, we decided to implement a light and fast heuristic to dynamically compute an approximation of the number of executions of loop to be executed, whose main specification is to be as precise as possible in the case of such simple loops.

### 2.2 A fast trip count prediction for "simple loops"

We apply our heuristic dynamically. In other words, we instrument a program - or its interpreter - to estimate the trip count immediately before the first iteration of the loops. Our instrumentation inspects the state of the variables used in the stop condition of each loop.

In the sequel, we only consider perfect loops with a single "interval" exit-condition, *i.e.* loops of the form : `while (e1` $\bowtie$ `e2) { some computation }`, where $\bowtie \in \{<, \leq, >, \geq\}$ and `e1` and `e2` are numeric variables. For this kind of loops, we assume that the trip count will be the absolute difference between $e_1$ and $e_2$. For instance, in a loop such as `for (int i = M; i < N; i++)`, we say that its trip count will be `|val(N) - val(i)|` (`val(x)` is the runtime value of `x` when the test is performed).

For each loop of this form, we insert a new instruction before the loop, according to Algorithm 1[1]. Let us point out that this algorithm only performs a single $O(1)$ operation per loop, without actually looking inside the body of the loop.

However, for loops of the form `for (int i = M; i < N; i=i+s)` ($s$ and $N$ invariants in the loops), this heuristic gives an overapproximation of the total number of loops, and the most precise result if $s = 1$. There is no guarantee of precision for the general case, of course, but the experiments will show that this simple-blind instrumentation fits our needs in practice. Because our heuristic is so simple, we can execute it quickly. This perfectly suits the needs of a JIT compiler.

[1] Obvious adaptations are necessary to handle $\leq$ and $\geq$.

**Algorithm 1** Trip Count Instrumentation Heuristic

**Input:** Loop $L$
**Output:** Loop $L'$ with new instructions that estimate its minimum trip count
1: **if** comparison $e_1 < e_2$ controls loop exit **then**
2: Insert instruction $tripcount = |e_1 - e_2|$ before $L$, giving $L'$.
3: **end if**

# 3 Use in Just-in-time compilation, implementation and results

## 3.1 Hot code detection in JIT compilers

Virtual environments that combine interpretation and compilation face a difficult question: when to invoke the JIT compiler [5]? Premature compilation might produce binaries that do not run long enough to amortize the cost of the JIT transformation. On the other hand, late compilation might delay the optimization of critical parts of the program. As an example, the Firefox browser separates native execution in two parts. After a few rounds of interpretation, the *baseline compiler* translates the program into non-optimized native code. Once this basic native code is deemed hot, it is re-compiled, this time by *IonMonkey*, an optimizing compiler. Figure 1 illustrates this behavior.

The moment when any of these code transformations happens is determined by *thresholds*, which count *discrete events*. A discrete event is either an invocation of a function, or an iteration of a loop [5]. Once a threshold is reached, the execution environment sends that code unit to the JIT compiler. We use our trip count predictor to identify that a threshold will be reached and call the JIT compiler earlier.

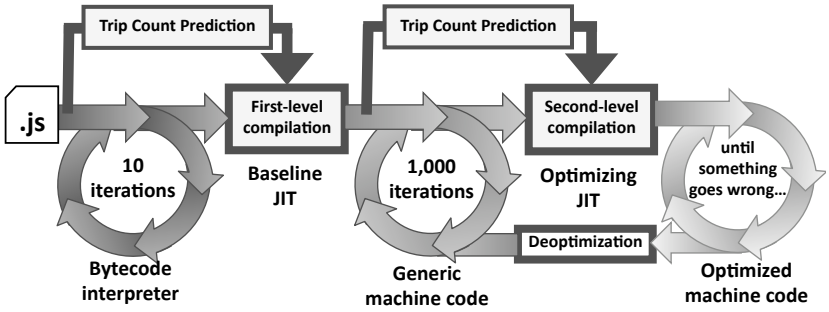



**Figure 1** Life cycle of a JavaScript program in our runtime environment. We can perform trip count prediction at two different execution stages.

We have deployed our predictor in the interpreter and in the baseline compiler used in Firefox, as we illustrate in Figure 1. The current distribution of Firefox calls the baseline compiler after 10 events, and the optimizing compiler after 1,000 events. If we predict that a loop will run for more than 10 iterations, we call the baseline compiler immediately, bypassing the warm-up period. Similarly, once in native mode, we call the optimizing compiler immediately upon finding a loop that we estimate to run more than 1,000 times.

## 3.2 Experiments

We measure the precision of our heuristic by comparing its results against the actual trip count observed during the concrete execution of programs. We have implemented our algorithm both in the LLVM compiler and in Mozilla Firefox. LLVM gives us the opportunity to test our approach in very large programs; Firefox lets us demonstrate its effectiveness in one of the most well-engineered JIT compilers in use today.

**Precision.** We group results in *interval orders* to measure the precision of our predictor. $N$ denotes the number of iterations of a loop observed via profiling, so the interval $[N, N]$ gives

| C programs | $[0, \sqrt{N}]$ | $]\sqrt{N}, N/2]$ | $]N/2, N[$ | $[N, N]$ | $]N, 2*N]$ | $]2*N, N^2]$ | $]N^2, +\infty]$ |
|---|---|---|---|---|---|---|---|
| 433.milc | 14 | 0 | 0 | 435,514,912 | 38,360 | 9,984 | 1,032,930 |
| 444.namd | 0 | 0 | 0 | 21,602,688 | 8,065 | 3,174 | 0 |
| 450.soplex | 1,851 | 367 | 112 | 186,939 | 12,784 | 10,231 | 43,338 |
| 470.lbm | 0 | 0 | 0 | 53,333 | 0 | 64 | 0 |
| 401.bzip2 | 5,270,006 | 2 | 311,724 | 14,386,219 | 15,987,072 | 1,502,759 | 28,939,274 |
| 403.gcc | 420,390 | 17 | 326 | 17,252,944 | 1,841,701 | 283,373 | 343,054 |
| 429.mcf | 96,576 | 87 | 42 | 555 | 2,643,736 | 634,623 | 1,705,369 |
| 445.gobmk | 8,392 | 20 | 400 | 651,081 | 70,492 | 117 | 20,141 |
| 456.hmmer | 0 | 0 | 0 | 31,551,408 | 8,512,797 | 3,893,744 | 3,273,429 |
| 458.sjeng | 0 | 620 | 2,565,378 | 41,787,788 | 3,423,766 | 7,917 | 1,038,075 |
| 462.libquantum | 0 | 0 | 0 | 8,182,095 | 0 | 1 | 0 |
| 464.h264ref | 367,010 | 0 | 0 | 302,394,768 | 12,636,622 | 5,636,387 | 10,703,859 |
| 473.astar | 7,147 | 0 | 0 | 74,614,711 | 602,244 | 2,550 | 609,708 |
| Total | 6,171,386 | 1,113 | 2,877,982 | 948,179,441 | 45,777,639 | 11,984,924 | 47,709,177 |
| Total (%) | 0.58% | 0.00% | 0.27% | 89.22% | 4.31% | 1.13% | 4.49% |

| JavaScript | $]1, \sqrt{N}]$ | $]\sqrt{N}, N/2]$ | $]N/2, N]$ | $[N, N]$ | $[N, 2N[$ | $[2N, N^2[$ | $[N^2, \infty[$ |
|---|---|---|---|---|---|---|---|
| SunSpider 1.0 | 0.0% | 0.0% | 0.0% | 89.2% | 2.0% | 4.7% | 4.1% |
| V8 v6 | 0.6% | 1.7% | 0.0% | 94.8% | 2.3% | 0.0% | 0.6% |
| Kraken 1.1 | 0.8% | 0.0% | 3.2% | 83.9% | 1.6% | 2.4% | 8.1% |

**Figure 2** Hit rate of our simple heuristic for C (top) and JavaScript (Bottom).

us the exact predictions. The intervals $[0, \sqrt{N}]$, $]\sqrt{N}, N/2]$, and $]N/2, N[$ represent loops that iterate less times than the prediction. The intervals $]N, 2*N]$, $]2*N, N^2]$, and $]N^2, +\infty]$ represent loops that iterate more times than the prediction. The interval $[N, N]$ is marked in gray in Figure 2. The farther from the center column, the worse is the precision. We only produce estimates for interval loops, that account for 71% of the loops in our benchmarks. We collect results for each execution of the loops; hence, the same loop might contribute several times to our final averages.

Figure 2 compares estimated and actual trip counts that we have collected with our profiler. We have correctly predicted 89.2% of the interval loops in the SPEC benchmarks and approximately 90% in the JavaScript benchmarks. To put these results in perspective, we compare them against the numbers presented recently by Tetzlaff and Glesner [8]. These authors also propose a heuristic to estimate the trip count of loops, based on dynamic profiling. Our precision is similar to the one they report. However, we do not need any sort of profiling, multiple compilation phases, nor annotations. Thus, we can be equally as precise, even though we run a much simpler algorithm. Furthermore, because our approach is simpler, it can be used to identify hot spots of programs in both real-world static compilers and JIT compilers.

**Speeding up Just-In-Time Compilers.** Figure 3 shows the speedup that we obtain using our trip count predictor to perform earlier invocation of the JIT compiler. Each number is the average of 100 runs. We call the baseline compiler immediately once we predict that a loop will iterate 10 times, and we call IonMonkey immediately once we predict that a loop will iterate 1,000 times. Figure 3, reveals that our technique has been able to speed up some benchmarks by a substantial factor. We have also detected slowdowns in a few scripts. This negative behavior happens in benchmarks that iterate for a very short time. In this case, early compilation is not able to pay off the cost of code generation.

## 4 Conclusion

In this paper, we have presented a heuristic to predict the trip count of loops. We have performed experiments in well-known public benchmarks showing that our technique is able to achieve a good precision, despite of the simplicity of our approach. Our source code is publicly available at `https://code.google.com/p/dynamic-loop-prediction/`

Moreover these experiments show that interval loops represent 70% of our benchmarks and a simple heuristic is the most precise in 90% of these interval loops. These results advocate the use of simple preprocessing for proving the termination (or counting the number of loops) of real-world programs that may include proper slicing and simple pattern-matching.

| SunSpider 1.0 | |
|---|---|
| 3d-cube | -1% |
| 3d-morph | -1% |
| 3d-raytrace | 1% |
| access-binary-trees | 0% |
| access-fannkuch | 2% |
| access-nbody | -1% |
| access-nsieve | 2% |
| bitops-3bit-in-byte | 0% |
| bitops-bits-in-byte | 1% |
| bitops-bitwise-and | 3% |
| bitops-nsieve-bits | 3% |
| ctrlflow-recursive | -1% |
| crypto-aes | 0% |
| crypto-md5 | 3% |
| crypto-sha1 | 10% |
| date-format-tofte | 2% |
| date-format-xparb | 2% |

| SunSpider 1.0 | |
|---|---|
| math-cordic | 2% |
| math-partial-sums | -1% |
| math-spectral-norm | -1% |
| regexp-dna | 0% |
| string-base64 | 24% |
| string-fasta | -7% |
| string-tagcloud | 1% |
| string-unpack-code | 0% |
| string-validate-input | -5% |

| V8 version 6.0 | |
|---|---|
| crypto | 0% |
| deltablue | 2% |
| earley-boyer | 0% |
| raytrace | 0% |
| regexp | 3% |
| richards | -1% |
| splay | 1% |

| Kraken 1.1 | |
|---|---|
| ai-astar | 1% |
| audio-beat-detect | 0% |
| audio-dft | -4% |
| audio-fft | 0% |
| audio-oscillator | 1% |
| img-gaussian-blur | -3% |
| imaging-darkroom | -3% |
| imaging-desaturate | 0% |
| json-parse-financial | 1% |
| json-stringify-tinder | 1% |
| crypto-aes | 6% |
| crypto-ccm | 2% |
| crypto-pbkdf2 | 7% |
| crypto-sha256-itrv | 6% |

**Figure 3** Speedup due to trip count predictor for benchmarks distributed with Firefox.

Future work may include the extraction of the remaining challenging loops for the community because there is still an open space to be explored.

## References


1 C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS*, pages 117–133. Springer, 2010.

2 C. Ancourt, F. Coelho, and F. Irigoin. A modular static analysis approach to affine loop invariants detection. *Electronic Notes in Theoretical Computer Science*, 267(1):3 – 16, 2010.

3 S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. Fast: Fast acceleration of symbolic transition systems. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 118–121. Springer, 2003.

4 G. Bhargav and S. Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *CAV*, pages 370–384. Springer, 2008.

5 E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *ASPLOS*, pages 202–211. ACM, 2000.

6 L. Gonnord and P. Schrammel. Abstract Acceleration in Linear Relation Analysis. *Science of Computer Programming, under press*, 2013.

7 J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *RTSS*, pages 57–66. IEEE, 2006.

8 Dirk Tetzlaff and Sabine Glesner. Static prediction of loop iteration counts using machine learning to enable hot spot optimizations. In *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on*, pages 300–307. IEEE, 2013.

# A Satisfiability Encoding of Dependency Pair Techniques for Maximal Completion*

**Haruhiko Sato[1] and Sarah Winkler[2]**

1 **Graduate School of Information Science and Technology, Hokkaido University, Sapporo, Japan**
2 **Institute of Computer Science, University of Innsbruck, Innsbruck, Austria**

**Abstract**

We present a general approach to encode termination in the dependency pair framework as a satisfiability problem, and include encodings of dependency graph and reduction pair processors. We use our encodings to increase the power of the completion tool Maxcomp.

## 1 Introduction

Maximal completion [4] is a simple yet efficient Knuth-Bendix completion approach which relies on MaxSAT solving. It can thus only compute convergent term rewrite systems (TRSs) whose termination can be expressed by satisfiability constraints.

Encoding termination techniques for TRSs via satisfiability problems has become common practice. However, to the best of our knowledge all previous encodings restrict to a single termination technique such as a specific reduction order or interpretations into a particular domain. Hence the maximal completion tool Maxcomp was so far restricted to LPO and KBO, and could not handle input problems such as $\mathsf{CGE}_2$ [6], which describes two commuting group endomorphisms as given by the following set of equations $\mathcal{E}$:

$$\begin{array}{lll} \mathsf{e} \cdot x \approx x & \mathsf{f}(x \cdot y) \approx \mathsf{f}(x) \cdot \mathsf{f}(y) & x \cdot (y \cdot z) \approx (x \cdot y) \cdot z \\ \mathsf{i}(x) \cdot x \approx \mathsf{e} & \mathsf{g}(x \cdot y) \approx \mathsf{g}(x) \cdot \mathsf{g}(y) & \mathsf{f}(x) \cdot \mathsf{g}(y) \approx \mathsf{g}(y) \cdot \mathsf{f}(x) \end{array}$$

In this paper we present a uniform layout for an SMT encoding of compound termination strategies that combine different techniques from the dependency pair framework. We give encodings of dependency pairs, a rule removal processor, and two versions of dependency graph approximations. We implemented our encodings on top of Maxcomp. Our experimental results show that this allows Maxcomp to complete problems like $\mathsf{CGE}_2$, and boosts its power beyond simple termination.

## 2 Preliminaries

We assume familiarity with term rewriting [1]. Knuth-Bendix completion aims to transform an equational system (ES) $\mathcal{E}$ into a TRS $\mathcal{R}$ which is convergent for $\mathcal{E}$, i.e., terminating, confluent and equivalent to $\mathcal{E}$. We write $\mathrm{CP}(\mathcal{R})$ for the set of critical pairs of a TRS $\mathcal{R}$, and $\downarrow_{\mathcal{R}}$ for $\to_{\mathcal{R}}^* \cdot {}_{\mathcal{R}}^*\!\leftarrow$. Maximal completion is a simple completion approach based on MaxSAT solving. For an input ES $\mathcal{E}$, it tries to compute $\varphi(\mathcal{E})$ where $\varphi$ is defined as

$$\varphi(\mathcal{C}) = \begin{cases} \mathcal{R} & \text{if } \mathcal{E} \cup \mathrm{CP}(\mathcal{R}) \subseteq \downarrow_{\mathcal{R}} \text{ for some } \mathcal{R} \in \mathfrak{R}(\mathcal{C}) \\ \varphi(\mathcal{C} \cup S(\mathcal{C})) & \text{otherwise} \end{cases}$$

$\mathfrak{R}(\mathcal{C})$ consists of terminating TRSs $\mathcal{R}$ such that $\mathcal{R} \subseteq \mathcal{C} \cup \mathcal{C}^{-1}$, and $S(\mathcal{C}) \subseteq \bigcup_{\mathcal{R} \in \mathfrak{R}(\mathcal{C})} \mathrm{CP}(\mathcal{R})$.

* This research was supported by the Austrian Science Fund project I963.

▶ **Theorem 1** ([4]). *The TRS $\varphi(\mathcal{E})$ is convergent for $\mathcal{E}$ if it is defined.*

In the maximal completion tool Maxcomp, $\mathfrak{R}(\mathcal{C})$ is computed by maximizing the number of satisfied clauses in $\bigvee_{s\approx t\in\mathcal{C}}[s > t] \vee [t > s]$, subject to the side constraints implied by the SAT/SMT encoding $[\cdot > \cdot]$ of some reduction order $>$.

In this paper we use the dependency pair (DP) framework to show termination of TRSs [3]. A DP problem is a pair of two TRSs $(\mathcal{P}, \mathcal{R})$, it is finite if it does not admit an infinite chain. A DP processor Proc is a function which maps a DP problem to either a set of DP problems or "no". It is sound if a DP problem $d$ is finite whenever $\mathrm{Proc}(d) = \{d_1, \ldots, d_n\}$ and all of $d_i$ are finite.

For an ES $\mathcal{C}$, we define the set of *dependency pair candidates* $\mathrm{DPC}(\mathcal{C})$ as all rules $\ell^\# \to u^\#$ such that $\ell \approx r \in \mathcal{C}$, $\ell \to r$ is a rewrite rule, and $r \trianglerighteq u$ but $\ell \not\trianglerighteq u$.

## 3 Encodings

We first illustrate the idea of our encodings by means of an example.

▶ **Example 2.** Suppose we want to orient a maximal number of equations from the ES $\mathcal{E}$ given in the introduction, where termination is to be shown by computing dependency pairs, applying a reduction pair processor based on a polynomial interpretation and finally a reduction pair processor based on LPO with argument filterings.

Let $\mathcal{P} = \mathrm{DPC}(\mathcal{E} \cup \mathcal{E}^{-1})$. For all equations $s \approx t$ in $\mathcal{C} = \mathcal{E} \cup \mathcal{E}^{-1} \cup \mathcal{P}$ we use *strict* variables $S^i_{s\to t}$ as well as *weak* variables $W^i_{s\to t}$ for all $0 \le i \le 3$. Moreover, boolean variables $X^{\mathrm{def}}_f$ encode whether $f$ is a defined symbol. We maximize the number of satisfied clauses in the disjunction $\bigvee_{s\approx t\in\mathcal{E}} S^0_{s\to t} \vee S^0_{t\to s}$ subject to the following constraints:

$$\bigwedge_{s\approx t\in\mathcal{E}\cup\mathcal{E}^{-1}} S^0_{s\to t} \to (W^1_{s\to t} \wedge X^{\mathrm{def}}_{\mathsf{root}(s)} \wedge \bigwedge_{\ell\to r\in\mathrm{DPC}(s\to t)} X^{\mathrm{def}}_{\mathsf{root}(r)} \to S^1_{\ell\to r}) \tag{a}$$

$$\bigwedge_{i=2}^{3} \bigwedge_{\ell\to r\in\mathcal{P}} (S^{i-1}_{\ell\to r} \to [\ell \geqslant^i r]) \wedge (\neg[\ell >^i r] \to S^i_{\ell\to r}) \tag{b}$$

$$\bigwedge_{i=2}^{3} \bigwedge_{s\approx t\in\mathcal{E}\cup\mathcal{E}^{-1}} W^{i-1}_{s\to t} \to (W^i_{s\to t} \wedge [s \geqslant^i t]) \tag{c}$$

$$\bigwedge_{\ell\to r\in\mathcal{P}} \neg S^3_{\ell\to r} \tag{d}$$

Clauses (a) trigger DPs and 'move' rules to the weak component, (b) expresses that if a DP is not oriented it remains to be considered, (c) requires rules to be weakly oriented, and (d) demands that finally no DP remains unoriented. Here $[\ell >^i r]$ ($[\ell \geqslant^i r]$) refers to strict (weak) orientation constraints imposed by polynomial interpretations for $i = 2$ and LPO with argument filterings for $i = 3$.

The following paragraphs transfer standard notions of the DP framework to our satisfiability setting. A *DP problem encoding* is a tuple $\mathcal{D} = (\mathcal{S}, \mathcal{W}, \phi)$ consisting of two sets of boolean variables $\mathcal{S} = \{S_{\ell\to r} \mid \ell \to r \in \mathcal{P}\}$ and $\mathcal{W} = \{W_{\ell\to r} \mid \ell \to r \in \mathcal{R}\}$ for TRSs $\mathcal{P}$ and $\mathcal{R}$, and a formula $\phi$. An assignment $\alpha$ is *finite for* a DP problem encoding $\mathcal{D} = (\mathcal{S}, \mathcal{W}, \phi)$ if $\alpha(\phi) = \top$ and the DP problem $(\mathcal{P}^{\mathcal{S}}_\alpha, \mathcal{R}^{\mathcal{W}}_\alpha)$ given by the TRSs

$$\mathcal{P}^{\mathcal{S}}_\alpha = \{\ell \to r \mid S_{\ell\to r} \in \mathcal{S},\ \alpha(S_{\ell\to r}) = \top\} \quad \mathcal{R}^{\mathcal{W}}_\alpha = \{\ell \to r \mid W_{\ell\to r} \in \mathcal{W},\ \alpha(W_{\ell\to r}) = \top\}$$

is finite. A *DP processor encoding* Proc maps a DP problem encoding $\mathcal{D} = (\mathcal{S}, \mathcal{W}, \phi)$ to a finite set of DP problem encodings $\mathrm{Proc}(\mathcal{D}) = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$. A DP processor encoding Proc is *sound* if for any $\mathcal{D}$ such that $\mathrm{Proc}(\mathcal{D}) = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ and any assignment $\alpha$ that is finite for all $\mathcal{D}_i$, it also holds that $\alpha$ is finite for $\mathcal{D}$.

For an ES $\mathcal{C}$ its set of *initial variables* is $\mathcal{I}_\mathcal{C} = \{I_{\ell\to r} \mid \ell \approx r \in \mathcal{C}\}$.

▶ **Definition 3.** For an ES $\mathcal{C}$ with initial variables $\mathcal{I}_\mathcal{C}$ the *initial DP problem encoding* is given by $\mathcal{D}_\mathcal{C} = (\mathcal{S}, \mathcal{W}, \phi)$ where $\mathcal{S} = \{S_{\ell\to r} \mid \ell \to r \in \mathrm{DPC}(\mathcal{C})\}$, $\mathcal{W} = \{W_{\ell\to r} \mid \ell \approx r \in \mathcal{C}\}$ and

$$\phi = \bigwedge_{\ell\approx r\in\mathcal{C}} I_{\ell\to r} \to \left( W_{\ell\to r} \wedge X^{\mathrm{def}}_{\mathsf{root}(\ell)} \wedge \bigwedge_{s\to t\in \mathrm{DPC}(\ell\to r)} X^{\mathrm{def}}_{\mathsf{root}(t)} \to S_{s\to t} \right)$$

▶ **Lemma 4.** *Let $\mathcal{C}$ be an ES. Suppose there is a tree whose nodes are labelled with DP problem encodings satisfying the following conditions:*

- *The root is labelled with the initial DP problem encoding $\mathcal{D}_\mathcal{C}$.*
- *For every non-leaf node labelled $\mathcal{D}$ with $n$ children labelled $\mathcal{D}_1, \ldots, \mathcal{D}_n$ there is a sound processor encoding* Proc *such that* $\mathrm{Proc}(\mathcal{D}) = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$.

*Let the leaves be labelled $\{(\mathcal{S}_i, \mathcal{W}_i, \phi_i) \mid 1 \le i \le k\}$. If the formula*

$$\phi = \bigwedge_{i=1}^{k} \phi_i \wedge \bigwedge_{s\to t\in\mathcal{S}_i} \neg S_{s\to t}$$

*is satisfied by an assignment $\alpha$ then the TRS $\mathcal{R} = \{\ell \to r \mid \alpha(I_{\ell\to r}) = \top\}$ is terminating.*

**Proof.** By induction on the tree structure, $\alpha$ is finite for all DP problem encodings occurring as labels. Termination of $\mathcal{R}$ follows from finiteness of $\alpha$ for the root label $\mathcal{D}_\mathcal{C}$. ◀

▶ **Definition 5** (Reduction pair processor)**.** Let $(>, \geqslant)$ be a reduction pair and $\pi$ an argument filtering, with satisfiability encodings $[\cdot \geqslant_\pi \cdot]$ and $[\cdot >_\pi \cdot]$

A DP problem encoding $(\mathcal{S}, \mathcal{W}, \phi)$ is mapped to $\{(\mathcal{S}', \mathcal{W}', \phi \wedge T_S \wedge T_W)\}$ where $\mathcal{S}' = \{S'_{\ell\to r} \mid S_{\ell\to r} \in \mathcal{S}\}$, $\mathcal{W}' = \{W'_{\ell\to r} \mid W_{\ell\to r} \in \mathcal{W}\}$, and

$$T_S = \bigwedge_{S_{\ell\to r}\in\mathcal{S}} S_{\ell\to r} \to [\ell \geqslant_\pi r] \wedge (\neg[\ell >_\pi r] \to S'_{\ell\to r})$$

$$T_W = \bigwedge_{W_{\ell\to r}\in\mathcal{W}} W_{\ell\to r} \to W'_{\ell\to r} \wedge [\ell \geqslant_\pi r]$$

Concrete encodings $[\cdot \geqslant_\pi \cdot]$ and $[\cdot >_\pi \cdot]$ for LPO/RPO, KBO as well as reduction orders given by polynomial and matrix interpretations—also in combination with argument filterings and usable rules—are well-studied, see for instance [5, 9, 2, 8].

Note that Definition 5 can easily be modified to admit rule removal by setting

$$T_W = \bigwedge_{W_{\ell\to r}\in\mathcal{W}} W_{\ell\to r} \to [\ell \geqslant_\pi r] \wedge (\neg[\ell >_\pi r] \to W'_{\ell\to r})$$

▶ **Definition 6** (Dependency graph processor)**.** A DP problem encoding $(\mathcal{S}, \mathcal{W}, \phi)$ is mapped to the set $\{(\mathcal{S}', \mathcal{W}', \psi)\}$ such that $\mathcal{S}' = \{S'_{\ell\to r} \mid S_{\ell\to r} \in \mathcal{S}\}$, $\mathcal{W}' = \{W'_{\ell\to r} \mid S_{\ell\to r} \in \mathcal{S}\} \cup \{W'_{\ell\to r} \mid W_{\ell\to r} \in \mathcal{W}\}$, and $\psi = \phi \wedge T_S \wedge T_W$ where

$$T_S = \bigwedge_{S_{p_1}, S_{p_2}\in\mathcal{S}} S_{p_1} \wedge S_{p_2} \wedge [p_1 \xrightarrow{\mathsf{edge}} p_2] \wedge \neg S'_{p_1} \wedge \neg S'_{p_2} \to X^{\mathrm{w}}_{p_1} > X^{\mathrm{w}}_{p_2}$$

$$T_W = \left( \bigwedge_{S_{\ell\to r}\in\mathcal{S}} S_{\ell\to r} \to W'_{\ell\to r} \right) \wedge \left( \bigwedge_{W_{\ell\to r}\in\mathcal{W}} W_{\ell\to r} \to W'_{\ell\to r} \right)$$

Here $T_S$ encodes cycle analysis of the graph in the sense that a cycle $p_1 \to p_2 \to \cdots \to p_n \to p_1$ issues the unsatisfiable constraint $X^{\mathrm{w}}_{p_1} > X^{\mathrm{w}}_{p_2} > \cdots > X^{\mathrm{w}}_{p_n} > X^{\mathrm{w}}_{p_1}$. For the formula $[s \to t \xrightarrow{\mathsf{edge}} u \to v]$ encoding the presence of an edge from $s \to t$ to $u \to v$ one can simply use $\top$ if $\mathsf{root}(t) = \mathsf{root}(u)$ and $\bot$ otherwise. (We also experimented with an encoding in terms of the unifiability between $\mathsf{REN}(\mathsf{CAP}(t))$ and $u$, but due to reasons of space do not present it here.)

The above encoding does not allow to use different orderings in SCCs, in contrast to what is commonly done in termination provers. However, it can be modified to consider SCCs by mapping a problem encoding to $k$ independent problem encodings.

▶ **Definition 7** (Dependency graph processor with $k$ SCCs)**.** A DP problem encoding $\mathcal{D} = (\mathcal{S}, \mathcal{W}, \phi)$ is mapped to $\{\mathcal{D}_i\}_{1 \le i \le k} = \{(\mathcal{S}_i, \mathcal{W}_i, \psi_i)\}_{1 \le i \le k}$ where $\mathcal{S}_i = \{S_{i,\ell \to r} \mid S_{\ell \to r} \in \mathcal{S}\}$, $\mathcal{W}_i = \{W_{i,\ell \to r} \mid S_{\ell \to r} \in \mathcal{S}\} \cup \{W_{i,\ell \to r} \mid W_{\ell \to r} \in \mathcal{W}\}$, $\psi_i = \phi \wedge T_{\mathrm{scc}}(k) \wedge T_S(i) \wedge T_W(i)$, and

$$T_{\mathrm{scc}}(k) = \bigwedge_{S_p \in \mathcal{S}} 1 \le X^{\mathrm{scc}}_p \le k \wedge \bigwedge_{S_{p_1}, S_{p_2} \in \mathcal{S}} S_{p_1} \wedge S_{p_2} \wedge [p_1 \xrightarrow{\mathsf{edge}} p_2] \to X^{\mathrm{edge}}_{p_1,p_2} \wedge X^{\mathrm{scc}}_{p_1} \ge X^{\mathrm{scc}}_{p_2}$$

$$T_S(i) = \bigwedge_{S_{p_1}, S_{p_2} \in \mathcal{S}} X^{\mathrm{edge}}_{p_1,p_2} \wedge X^{\mathrm{scc}}_{p_1} = i \wedge X^{\mathrm{scc}}_{p_2} = i \wedge \neg S_{i,p_1} \wedge \neg S_{i,p_2} \to X^{\mathrm{w}}_{p_1} > X^{\mathrm{w}}_{p_2}$$

$$T_W(i) = \bigwedge_{W_p \in \mathcal{W}} W_p \to W_{i,p} \wedge \bigwedge_{S_p \in \mathcal{S}} S_p \wedge X^{\mathrm{scc}}_p = i \wedge \left( \bigvee_{S_{p'} \in \mathcal{S} \setminus \{S_p\}} X^{\mathrm{scc}}_p = X^{\mathrm{scc}}_{p'} \right) \to W_{i,p}$$

Here $X^{\mathrm{edge}}_{p_1,p_2}$ is a boolean variable encoding the presence of both DPs $p_1$ and $p_2$ as well as an edge from $p_1$ to $p_2$, and $X^{\mathrm{scc}}_p$ is an integer variable assigning an SCC number to a DP $p$. Hence $T_{\mathrm{scc}}(k)$ encodes the separation of the graph into at most $k$ SCCs, and $T_S(i), T_W(i)$ encode conditions to orient the $i$th SCC.

Soundness of all the above encodings can be shown by relating them to their processor counterparts [3], but we omit the proofs here due to lack of space.

## 4 Experiments

We implemented our DP framework encoding in Maxcomp as described in Section 3. Besides enhancing the previous LPO and KBO implementations with argument filterings, we also added a (restricted version of) linear polynomial interpretations as reduction pair processors. Both versions of the DG processors were included as well.

Table 1 summarizes our experimental results[1] for the test bed comprising 115 equational systems from the distribution of mkbTT [7]. Each ES was given a time limit of 180 seconds, timeouts are marked $\infty$. Row (1) corresponds to the original Maxcomp using LPO. In setting (2) we use a strategy combining dependency pairs with reduction pair processors applying linear polynomials and LPO. Setting (3) enhances setting (2) with a simple DG processor encoding according to Definition 6, and setting (4) uses Definition 7 with 2 SCCs instead. A simple heuristic is applied by the *automatic mode* (5): one iteration is run with plain LPO and setting (2) in parallel, but afterwards only one strategy (which can orient more of the initial equations) is kept. The column # lists the number of successful completions, the next column gives the average time for a successful completion in seconds.

[1] Details available from `http://cl-informatik.uibk.ac.at/software/maxcompdp`

| | method | # | avg. time | $CGE_2$ | proofreduction | equiv_proofs |
|---|---|---|---|---|---|---|
| (1) | Maxcomp | 85 | 3.8 | $\infty$ | $\infty$ | $\infty$ |
| (2) | DPs, poly, LPO | 83 | 14.7 | 6.4 | $\infty$ | 1.6 |
| (3) | DG, poly, LPO | 40 | 2.8 | $\infty$ | $\infty$ | $\infty$ |
| (4) | DG/2SCCs, poly, LPO | 25 | 2.5 | $\infty$ | $\infty$ | $\infty$ |
| (5) | auto | 92 | 10.2 | 5.6 | 135.1 | 1.5 |

**Table 1** Experimental Results.

With the relatively lightweight DP strategy (2) we successfully complete the problems mentioned in Table 1, which cannot be completed using plain LPO or KBO. However, some other systems are lost, compared to Maxcomp using LPO. Typically, these problems require many iterations and/or give rise to many equations. Thus in total (2) completes not quite as many systems as (1), and the average time is tripled. Settings (3) and (4) require considerably more encoding effort and hence succeed on comparatively few systems. For instance, only proving termination of the convergent (unreduced) TRS for equiv_proofs (74 rules) produced in a completion run with setting (2) takes 1.4 seconds for strategy (2) (12K variables, 45K clauses) but 176 seconds with setting (3) (290K variables, 1.2M clauses). Overall the automatic mode turned out to be most powerful since it can often be efficient by applying LPO, but also switch to a more sophisticated strategy in case of unorientable equations. There are even some problems like proofreduction where (5) succeeds but (2) does not—apparently it can be preferable to apply LPO in the beginning before switching to the DP strategy.

## References


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

2 J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *JAR*, 40(2-3):195–220, 2008.

3 J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *LPAR*, volume 3452 of *LNCS*, pages 301–331, 2005.

4 D. Klein and N. Hirokawa. Maximal completion. In *RTA*, volume 10 of *LIPIcs*, pages 71–80, 2011.

5 P. Schneider-Kamp, R. Thiemann, E. Annov, M. Codish, and J. Giesl. Proving termination using recursive path orders and SAT solving. In *FroCoS*, volume 4720 of *LNCS (LNAI)*, pages 267–282, 2007.

6 A. Stump and B. Löchner. Knuth-Bendix completion of theories of commuting group endomorphisms. *IPL*, 98(5):195–198, 2006.

7 S. Winkler, H. Sato, A. Middeldorp, and M. Kurihara. Multi-completion with termination tools. *JAR*, 50(3):317–354, 2013.

8 H. Zankl, N. Hirokawa, and A. Middeldorp. Constraints for argument filterings. In *SOFSEM*, volume 4362 of *LNCS*, pages 579–590, 2007.

9 H. Zankl, N. Hirokawa, and A. Middeldorp. KBO orientability. *JAR*, 43(2):173–201, 2009.

# To Infinity… and Beyond!*

**Caterina Urban[1] and Antoine Miné[1]**

1 **École Normale Supérieure - CNRS - INRIA**
**Paris, France**
**urban@di.ens.fr,mine@di.ens.fr**

## Abstract

The traditional method for proving program termination consists in inferring a ranking function. In many cases (i.e., programs with unbounded non-determinism), a single ranking function over natural numbers is not sufficient. Hence, we propose a new abstract domain to automatically infer ranking functions over ordinals. We extend an existing domain for piecewise-defined natural-valued ranking functions to polynomials in $\omega$, where the polynomial coefficients are natural-valued functions of the program variables. The abstract domain is parametric in the choice of the state partitioning inducing the piecewise-definition and the type of functions used as polynomial coefficients. To our knowledge this is the first abstract domain able to reason about ordinals. Handling ordinals leads to a powerful approach for proving termination of imperative programs, which in particular allows us to take a first step in the direction of proving termination under fairness constraints and proving liveness properties of (sequential and) concurrent programs.

**1998 ACM Subject Classification** D.2.4 Software/Program Verification, F.3.1 Specifying and Verifying and Reasoning about Programs, F.3.2 Semantics of Programming Languages

**Keywords and phrases** Abstract Interpretation, Ranking Function, Ordinals, Termination

**Digital Object Identifier** 10.4230/LIPIcs.xxx.yyy.p

## 1 Introduction

The traditional method for proving program termination [6] consists in inferring ranking functions, namely mappings from program states to elements of a well-ordered set (e.g., ordinals) whose value decreases during program execution.

Intuitively, we can define a partial ranking function from the states of a program to ordinals in an incremental way: we start from the program final states, where the function has value $0$ (and is undefined elsewhere); then, we retrace the program backwards enriching the domain of the function with the co-reachable states mapped to the maximum number of program steps until termination. In [3], this intuition is formalized into a most precise ranking function that can be expressed in fixpoint form by abstract interpretation [2] of the program maximal trace semantics.

However, the most precise ranking function is not computable. In [11], we present a decidable abstraction for imperative programs by means of piecewise-defined ranking functions over natural numbers. These functions are attached to the program control points and represent an upper bound on the number of program execution steps remaining before termination. Nonetheless, in many cases (i.e., programs with unbounded non-determinism), natural-valued ranking functions are not powerful enough. For this reason, we propose a new abstract domain to automatically infer ranking functions over ordinals.

We extend the abstract domain of piecewise-defined natural-valued ranking functions to *piecewise-defined ordinal-valued ranking functions* represented as polynomials in $\omega$, where the polynomial coefficients are natural-valued functions of the program variables. The domain automatically infers such ordinal-valued functions through backward invariance analysis. To handle disjunctions arising

* The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no. 269335 (ARTEMIS project MBAT) (see Article II.9. of the JU Grant Agreement)

from tests and loops, the analysis automatically partitions the space of values for the program variables into abstract program states, inducing a piecewise definition of the functions. Moreover, the domain naturally infers sufficient preconditions for program termination. The analysis is sound: all program executions respecting these sufficient preconditions are indeed terminating, while an execution that does not respect these conditions might not terminate.

The abstract domain is parametric in the choices of the state abstraction used for partitioning (in particular, we can abstract the program states using any convex abstract domain such as intervals [1], octagons [9], polyhedra [4], ...) and the type of functions used as polynomial coefficients (e.g., affine, quadratic, cubic, exponential, ...).

To our knowledge this is the first abstract domain able to reason about ordinals. Handling ordinals leads to a powerful approach for proving termination of imperative programs, which in particular allows us to take a first step in the direction of proving termination under fairness constraints and proving liveness properties of (sequential and) concurrent programs.

## 2 Ordinal-Valued Ranking Functions

We derive a decidable program termination semantics by abstract interpretation of the program most precise ranking function [3]: first, we introduce the abstract domain $\mathsf{O}$ of ordinal-valued functions; then, in the next section, we employ state partitioning to lift this abstraction to piecewise-defined functions [11].

Let $\mathcal{X}$ be a finite set of program variables. We split the program state space $\Sigma \triangleq \mathcal{L} \times \mathcal{E}$ into program control points $\mathcal{L}$ and environments $\mathcal{E} \triangleq \mathcal{X} \to \mathbb{Z}$, which map each program variable to an integer value. No approximation is made on $\mathcal{L}$. On the other hand, each program control point $l \in \mathcal{L}$ is associated with an element $o \in \mathcal{O}$ of the abstract domain $\mathsf{O}$. Specifically, $o$ represents an abstraction of the partial function $\gamma_{\mathsf{O}}(o) \in \mathcal{E} \rightharpoonup \mathbb{O}$ defined on the environments related to the program control point $l$: $\langle \mathcal{E} \rightharpoonup \mathbb{O}, \sqsubseteq \rangle \xleftarrow{\gamma_{\mathsf{O}}} \langle \mathcal{O}, \sqsubseteq_{\mathsf{O}} \rangle$. Intuitively, where defined, the partial function provides a ranking function proving (definite) termination; where undefined, it denotes (potential) non-termination.

**Natural-Valued Functions.** We assume we are given an abstraction $\langle \mathcal{S}, \sqsubseteq_{\mathsf{S}} \rangle$ of environments: $\langle \mathcal{P}(\mathcal{E}), \subseteq \rangle \xleftarrow{\gamma_{\mathsf{S}}} \langle \mathcal{S}, \sqsubseteq_{\mathsf{S}} \rangle$ (i.e., any abstract domain such as intervals [1], octagons [9], convex polyhedra [4], ...), and an abstraction $\langle \mathcal{S} \times \mathcal{F}, \sqsubseteq_{\mathsf{F}} \rangle$ of $\langle \mathcal{E} \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$ by means of *natural-valued* functions of the program variables: $\langle \mathcal{E} \rightharpoonup \mathbb{O}, \sqsubseteq \rangle \xleftarrow{\gamma_{\mathsf{F}}} \langle \mathcal{S} \times \mathcal{F}, \sqsubseteq_{\mathsf{F}} \rangle$. More specifically, the abstraction $\langle \mathcal{S} \times \mathcal{F}, \sqsubseteq_{\mathsf{F}} \rangle$ encodes a partial function $v \in \mathcal{E} \rightharpoonup \mathbb{O}$ by a pair $\langle s, f \rangle$ of a natural-valued (total) function (e.g., an affine function [11]) $f \in \mathcal{F}$ and an abstract state $s \in \mathcal{S}$ which restricts its domain. For instance, $\langle [1,5], 3x+2 \rangle$ denotes the affine function $3x+2$ restricted to the interval $[1,5]$. We can now use the abstractions $\mathcal{S}$ and $\mathcal{F}$ to build the abstract domain $\mathsf{O}$.

**Ordinal-Valued Functions.** The elements of the abstract domain $\mathsf{O}$ belong to $\mathcal{O} \triangleq \mathcal{S} \times \mathcal{W}$ where $\mathcal{W} \triangleq \{\bot_{\mathsf{W}}\} \cup \{\sum_i \omega^i \cdot f_i \mid f_i \in \mathcal{F}\} \cup \{\top_{\mathsf{W}}\}$ is the set of ordinal-valued ranking functions of the program variables (in addition to the function $\bot_{\mathsf{W}}$ representing potential non-termination, and the function $\top_{\mathsf{W}}$ representing the lack of enough information to conclude[1]). More specifically, an abstract function $o \in \mathcal{O}$ is a pair of an abstract state $s \in \mathcal{S}$ and a polynomial in $\omega$ (i.e., an ordinal in Cantor Normal Form) $\omega^k \cdot f_k + \cdots + \omega^2 \cdot f_2 + \omega \cdot f_1 + f_0$ where the coefficients $f_0, f_1, f_2, \ldots, f_k$ belong to $\mathcal{F}$. Note that the ordinal $\omega^k \cdot f_k + \cdots + \omega^2 \cdot f_2 + \omega \cdot f_1 + f_0$ is isomorphic to the lexicographic tuple $(f_k, \ldots, f_1, f_0)$. In fact, our abstract domain is isomorphic to the set of all lexicographic ranking functions with finite (but unbounded) number of components. In the following, with abuse of notation, we use a map $s \mapsto p$ to denote the pair of $s \in \mathcal{S}$ and $p \in \mathcal{W}$, i.e., $p$ restricted to $s$. The abstract domain $\mathsf{O}$ is parameterized

[1] In fact, our abstract domain is equipped with an approximation and a computational ordering (here not discussed) which respectively do and do not distinguish between $\bot_{\mathsf{W}}$ and $\top_{\mathsf{W}}$. We refer to [11] for further discussion.

by the choices of the abstraction $\langle \mathcal{S}, \sqsubseteq_{\mathsf{S}} \rangle$ and the type (e.g., affine, quadratic, cubic, exponential, ...) of natural-valued functions used as polynomial coefficients $f_0, f_1, f_2, \ldots, f_n \in \mathcal{F}$. As an example, $[1,5] \mapsto \omega \cdot (3x+2)+(2x)$ uses intervals and affine functions respectively.

As for the operators of the abstract domain, we briefly describe only the join operator and the assignment transfer function. We refer to [12] for more details and examples.

The join operator $\sqcup_{\mathsf{O}}$, given two abstract functions $o_1 \triangleq s_1 \mapsto p_1$ and $o_2 \triangleq s_2 \mapsto p_2$, determines the function $o \triangleq s \mapsto p$, defined on their common domain $s \triangleq s_1 \sqcap_{\mathsf{S}} s_2$ with value $p \triangleq p_1 \sqcup_{\mathsf{P}} p_2$. Specifically, the unification $p_1 \sqcup_{\mathsf{P}} p_2$ of two polynomials $p_1$ and $p_2$ is done in ascending powers of $\omega$, joining the coefficients of similar terms (i.e., terms with the same power of $\omega$). The join of two coefficients $f_1$ and $f_2$ is provided by $f \triangleq f_1 \sqcup_{\mathsf{F}} f_2$ and is defined as a *natural-valued* function (of the same type of $f_1$ and $f_2$) greater than $f_1$ and $f_2$ (on the domain $s$). Whenever such function does not exist, we force $f$ to equal $0$ and we carry $1$ to the unification of terms with next higher degree. Intuitively, whenever natural-valued functions are not sufficient, we naturally resort to ordinals. Let us consider the join $\omega^k \cdot f$ of two terms $\omega^k \cdot f_1$ and $\omega^k \cdot f_2$. Forcing $f$ to equal $0$ and carrying $1$ to the terms with next higher degree is exactly the same as considering $f$ equal to $\omega$: $\omega^k \cdot f = \omega^k \cdot \omega = \omega^{k+1} \cdot 1 + \omega^k \cdot 0 = \omega^{k+1}$. To avoid computing infinite increasing chains of abstract functions, to analyze loops we use a widening operator [1] $\triangledown_{\mathsf{O}}$ which is similar to the join $\sqcup_{\mathsf{O}}$ but defaults to $\top_{\mathsf{P}}$ when the abstract function has increased between iterates.

In order to handle assignments, the abstract domain is equipped with an operation to substitute an arithmetic expression for a variable within a function $f \in \mathcal{F}$. Given an abstract function $o \triangleq s \mapsto p$, an assignment is carried out independently on the abstract state $s$ and on the polynomial $p$. In particular, an assignment on $p$ is performed in ascending powers of $\omega$, possibly carrying $1$ to the term with next higher degree. The need for carrying might occur in case of non-deterministic assignments: it is necessary to take into account all possible outcomes of the assignment, possibly using $\omega$ as approximation.

## 3 Piecewise-Defined Ordinal-Valued Ranking Functions

In the following, we lift the abstract domain $\mathsf{O}$ to piecewise-defined ranking functions [11].

The elements of the abstract domain belong to $\mathcal{V} \triangleq \mathcal{P}(\mathcal{S} \times \mathcal{W})$. More specifically, an element $v \in \mathcal{V}$ of the abstract domain has now the form:

$$v \triangleq \begin{cases} s_1 \mapsto p_1 \\ \quad \vdots \\ s_k \mapsto p_k \end{cases}$$

where the abstract states $s_1, \ldots, s_k \in \mathcal{S}$ induce a partition of the space of environments $\mathcal{E}$ and $p_1, \ldots, p_k$ are ranking functions represented as polynomials $\omega^k \cdot f_k + \cdots + \omega^2 \cdot f_2 + \omega \cdot f_1 + f_0$ whose coefficients $f_0, f_1, f_2, \ldots, f_n \in \mathcal{F}$ are natural-valued functions of the program variables.

The binary operators of the abstract domain rely on a partition unification algorithm that, given two piecewise-defined ranking functions $v_1$ and $v_2$, modifies the partitions on which they are defined into a common refined partition of the space of program environments. For example, in case of partitions determined by intervals with constant bounds, the unification simply introduces new bounds consequently splitting intervals in both partitions. Then, the binary operators are applied piecewise: the piecewise join $\sqcup_{\mathsf{V}}$ computes the piecewise-defined natural-valued ranking function greater than $v_1$ and $v_2$ using $\sqcup_{\mathsf{O}}$. The piecewise widening $\triangledown_{\mathsf{V}}$ keeps only the partition of the domain of the first function. In this way, it prevents the number of pieces of an abstract function from growing indefinitely. It also prevents the indefinite growth of the *value* of an abstract function by using $\triangledown_{\mathsf{O}}$.

The unary operators for assignments and tests are also applied piecewise. In particular, assignments are carried out independently on each abstract state and each ranking function. Then, the resulting covering induced by the over-approximated abstract states is refined (joining overlapping pieces) to obtain once again a partition.

The operators of the abstract domain are combined together to compute an abstract ranking function for a program, through backward invariance analysis. The starting point is the constant function equal

```
int : x1, x2
while ( x1≠0 ∧ x2≥0 ) do
   if ( x1>0 ) then
      if ( ? ) then
         x1 := x1 - 1
         x2 := [−∞,∞]
      else
         x2 := x2 - 1
   else /* x1<0 */
      if ( ? ) then
         x1 := x1 + 1
      else
         x2 := x2 - 1
         x1 := [−∞,∞]
```

**Figure 1** Program with no lexicographic ranking function.

```
int : x, b
x := 0, b := 1
[ while ( b > 0 ) do x := x + 1 ] ||
[ b := 0 ]
```

```
int : x, b, z1, z2
z1 := [0,+∞], z2 := [0,+∞], x := 0, b := 1
while ( b > 0 ) do
   if ( z1≤z2 ) then
      x := x + 1
      z1 := [0,+∞]
      z2 := z2 - 1
   else
      b := 0
```

**Figure 2** Concurrent variant (above) and non-deterministic variant (below) of Dijkstra's random number generator [5].

to 0 at the program final control point. The ranking function is then propagated backwards towards the program initial control point taking assignments and tests into account using join and widening for loops. As a consequence of the soundness of all abstract operators (see [11]), we can establish the soundness of the analysis for proving program termination: the program states for which the analysis finds a ranking function are states from which the program indeed terminates.

**Implementation.** We have incorporated the implementation of the abstract domain O of ordinal-valued ranking functions into our prototype static analyzer [10] based on piecewise-defined ranking functions. The prototype accepts programs written in (a subset of) C. It is written in OCaml and, at the time of writing, the available abstractions for program environments $\mathcal{S}$ are based on intervals [1], octagons [9] or convex polyhedra [4], and the available abstraction for natural-valued functions $\mathcal{F}$ is based on affine functions. The operators for the intervals, octagons and convex polyhedra abstract domains are provided by the APRON library [8]. The analysis proceeds by structural induction on the program syntax, iterating loops until an abstract fixpoint is reached. In case of nested loops, a fixpoint on the inner loop is computed for each iteration of the outer loop.

▶ **Example 1.** Let us consider the program in Figure 1. The variables $x_1$ and $x_2$ can have any initial integer value, and the program behaves differently depending on whether $x_1$ is positive or negative. In case $x_1$ is positive, the program either decrements the value of $x_2$ or decrements the value of $x_1$ and resets $x_2$ to any value. In case $x_1$ is negative, the program either increments the value of $x_1$ or decrements the value of $x_2$ and resets $x_1$ to any value (possibly positive). The loop exits when $x_1$ is equal to zero or $x_2$ is less than zero.

Note that there does not exist a lexicographic affine ranking function for the loop. In fact, the variables $x_1$ and $x_2$ can be alternatively reset to any value at each loop iteration: the value of $x_2$ is reset in the first branch of the first if statement (i.e., if $x_1 > 0$) while the value of $x_1$ is reset in the second branch of the first if statement (i.e., if $x_1 < 0$).

Nonetheless, the program always terminates, regardless of the initial values for $x_1$ and $x_2$, and regardless of the non-deterministic choices taken during execution. Our prototype is able to prove the program terminating in about 10 seconds (with a widening delay of 3 iterations). We automatically infer the following piecewise-defined ranking function (at loop entry):

$$f(x_1,x_2) = \begin{cases} \omega^2+\omega\cdot(x_2-1)-4x_1+9x_2-2 & x_1<0 \wedge x_2>0 \\ 1 & x_1=0 \vee x_2\leq 0 \\ \omega\cdot(x_1-1)+9x_1+4x_2-7 & x_1>0 \wedge x_2>0 \end{cases}$$

Note that, from any state where $x_1 < 0$ and $x_2 = k_2 > 0$, whenever the value of $x_1$ is reset, it is possible to jump to any state where $x_2 = k_2 - 1$. Thus, $f$ must go up to $\omega^2$ (... and beyond!) as it is possible to jump through unbounded non-determinism to states with value of the most precise ranking function equal to an arbitrary ordinal between $\omega$ and $\omega^2$.

Finally, note the expressions identified as coefficients of $\omega$: when $x_1 < 0$, the coefficient of $\omega$ is an expression in $x_2$ (since $x_2$ guides the progress towards the final states), and when $x_1 > 0$, the coefficient of $\omega$ is an expression in $x_1$ (because $x_1$ now rules the progress towards termination). The expressions are automatically inferred by the analysis without requiring assistance from the user. ◀

## 4 Conclusion and Future Work

In this paper, we proposed a parameterized abstract domain for proving termination of imperative programs. The domain automatically infers sufficient conditions for program termination, and synthesizes piecewise-defined ordinal-valued ranking functions through backward invariance analysis.

The full version of this short paper has been published in [12]. Due to space constraints, we refer to [11, 12] for a comparison with related work.

It remains for future work to extend our research to proving termination under fairness constraints and thus proving liveness properties of (sequential and) concurrent programs. However, as shown in the following example, handling ordinals already allows us to take a first step in this direction.

▶ **Example 2.** Let us consider the concurrent variant of Dijkstra's random number generator [5] in Figure 2. The program is terminating *under fairness assumptions*. Nonetheless, a program transformation can be applied in order to introduce *unbounded non-determinism* and thus explicitly represent the fair scheduler within the program [7]. Once this transformation is carried out, the resulting non-deterministic program (in Figure 2) can be proved terminating by our ordinal-valued ranking functions.

However, more abstractions are to be expected to handle all practical cases.

## References


1 Patrick Cousot and Radhia Cousot. Static Determination of Dynamic Properties of Programs. In *Symposium on Programming*, pages 106–130, 1976.
2 Patrick Cousot and Radhia Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.
3 Patrick Cousot and Radhia Cousot. An Abstract Interpretation Framework for Termination. In *POPL*, pages 245–258, 2012.
4 Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*, pages 84–96, 1978.
5 Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
6 Robert W. Floyd. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.
7 Nissim Francez. *Fairness*. Springer, 1986.
8 Bertrand Jeannet and Antoine Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV*, pages 661–667, 2009.
9 Antoine Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
10 Caterina Urban. FuncTion. `http://www.di.ens.fr/~urban/FuncTion.html`.
11 Caterina Urban. The Abstract Domain of Segmented Ranking Functions. In *SAS*, pages 43–62, 2013.
12 Caterina Urban and Antoine Miné. An Abstract Domain to Infer Ordinal-Valued Ranking Functions. In *ESOP*, 2014.

# Automating Elementary Interpretations

**Harald Zankl, Sarah Winkler, and Aart Middeldorp**

**Institute of Computer Science, University of Innsbruck, 6020 Innsbruck, Austria**

**Abstract**

We report on an implementation of elementary interpretations for automatic termination proofs.

## 1 Introduction

Proving termination of rewrite systems by polynomial interpretations is well-studied. In this work we go beyond polynomials by considering a subset of elementary functions including exponentiation. The approach is motivated by Lescanne's factorial example [3]

$$\begin{array}{lll} 0 + x \to x & 0 \cdot x \to 0 & \mathsf{fact}(0) \to \mathsf{s}(0) \\ \mathsf{s}(x) + y \to \mathsf{s}(x + y) & \mathsf{s}(x) \cdot y \to x \cdot y + y & \mathsf{fact}(\mathsf{s}(x)) \to \mathsf{s}(x) \cdot \mathsf{fact}(x) \\ x \cdot (y + z) \to x \cdot y + x \cdot z & & \end{array}$$

which does not admit a (direct) termination proof by polynomials. Consider the algebra $\mathcal{A}$ with carrier $\mathbb{N}_{\geqslant 1}$ and the interpretation functions $0_{\mathcal{A}} = 2$, $\mathsf{s}_{\mathcal{A}}(x) = x+2$, $x +_{\mathcal{A}} y = 2x+y+1$, $x \cdot_{\mathcal{A}} y = 2^x y$, and $\mathsf{fact}_{\mathcal{A}}(x) = 2^{2^x}$. They establish termination of the TRS, since for every rule the term interpretation of the left-hand side is larger than that of the right-hand side, i.e., for all $x, y, z \in \mathbb{N}_{\geqslant 1}$:

$$\begin{array}{lll} x + 5 > x & 2^2 x > 2 & 2^{2^2} > 4 \\ 2x + y + 5 > 2x + y + 3 & 2^{x+2} y > 2^{x+1} y + y + 1 & 2^{2^{x+2}} > 2^{x+2} 2^{2^x} = 2^{x+2+2^x} \\ 2^x(2y + z + 1) > 2^{x+1} y + 2^x z + 1 & & \end{array}$$

In this note we show how to automate the search for such interpretation functions. In particular one has to (a) find suitable coefficients for the interpretations, (b) evaluate the term interpretations, and (c) compare two term interpretations. In the sequel we address these issues in reverse order and also discuss the limitations of this approach.

## 2 Automation of Elementary Algebras

We assume familiarity with term rewriting [1]. For a set of function symbols $\mathcal{F}$, a (well-founded) algebra $\mathcal{A} = (A, \{f_{\mathcal{A}} \mid f \in \mathcal{F}\}, >)$ consists of a carrier $A$, (a set of) interpretation functions $f_{\mathcal{A}} : A \times \cdots \times A \to A$, and a well-founded order $>$ on $A$. An interpretation function $f_{\mathcal{A}}$ is *monotone* if $a > b$ implies $f_{\mathcal{A}}(\ldots, a_{i-1}, a, a_{i+1}, \ldots) > f_{\mathcal{A}}(\ldots, a_{i-1}, b, a_{i+1}, \ldots)$. An algebra is *monotone* if all its interpretation functions are monotone. A TRS $\mathcal{R}$ is *compatible* with an algebra $\mathcal{A}$ if $[\alpha]_{\mathcal{A}}(\ell) > [\alpha]_{\mathcal{A}}(r)$ for every $\ell \to r \in \mathcal{R}$ and assignment $\alpha$. Here $[\alpha]_{\mathcal{A}}(t)$ denotes the value resulting when interpreting the term $t$ in the algebra $\mathcal{A}$ under the assignment $\alpha$. A TRS is terminating if and only if it is compatible with a well-founded monotone algebra.

Inspired by the above example we use interpretation functions of the following shape:

▶ **Definition 1.** A *fixed-base elementary interpretation function* (FBI) of depth 0 is a linear function $f(\overline{x}) = \sum_{1 \leqslant i \leqslant n} x_i f_i + f_0$ and an FBI of depth $d+1$ has the shape

$$f(\overline{x}) = \sum_{1 \leqslant i \leqslant n} x_i f_i + f_0 + b^{f'(\overline{x})} \left( \sum_{1 \leqslant i \leqslant n} x_i \hat{f}_i + \hat{f}_0 \right) \tag{1}$$

where $f_0, f_1, \ldots, f_n, \hat{f}_0, \hat{f}_1, \ldots, \hat{f}_n$ are natural numbers, $f'(\overline{x})$ is an FBI of depth $d$, and $b \geqslant 2$ is a fixed natural number. We use the abbreviations $\dot{f}(\overline{x}) = \sum_{1 \leqslant i \leqslant n} x_i f_i + f_0$ and $\hat{f}(\overline{x}) = \sum_{1 \leqslant i \leqslant n} x_i \hat{f}_i + \hat{f}_0$. An *FBI algebra* has $\mathbb{N}_{\geqslant 1}$ as carrier and FBIs as interpretation functions for all function symbols in the signature.

We treat an FBI $f(\overline{x})$ of depth 0 as $\sum_{1 \leqslant i \leqslant n} x_i f_i + f_0 + b^0 0$ to avoid case distinctions. Hence in the sequel we will use FBIs $f(\overline{x})$ and $g(\overline{x})$ of the shape (1).

The following recursive definition reduces the comparison of FBIs to the comparison of non-linear polynomials.

▶ **Definition 2.** Let $\lfloor b^{f'(\overline{x})} \rfloor = \big((\dot{f}'(\overline{x}) + \hat{f}'(\overline{x}) = 0) \ ? \ 1 \ : \ b\,(\dot{f}'(\overline{x}) + \hat{f}'(\overline{x}))\big)$. Note that $b^{f(\overline{x})} \geqslant \lfloor b^{f(\overline{x})} \rfloor$. Further let $p(\overline{x}) = \dot{f}'(\overline{x}) + \hat{f}'(\overline{x}) - \dot{g}'(\overline{x}) - \hat{g}'(\overline{x})$ and $h(\overline{x}) = \lfloor b^{p(\overline{x})} \rfloor \hat{f}(\overline{x}) - \hat{g}(\overline{x})$. We define

$$
\begin{aligned}
&[f(\overline{x}) > g(\overline{x})] \ = \ (\hat{g}(\overline{x}) > 0 \to [f'(\overline{x}) \geqslant g'(\overline{x})]) \ \wedge \ \big( \\
&\quad (\hat{f}(\overline{x}) > 0 \wedge [f'(\overline{x})\, b > g(\overline{x})]) \ \vee && \text{(d)} \\
&\quad (\dot{f}(\overline{x}) \geqslant \dot{g}(\overline{x}) \wedge \hat{f}(\overline{x}) \geqslant \hat{g}(\overline{x}) \wedge \\
&\qquad ((\hat{f}(\overline{x}) > 0 \wedge [f'(\overline{x}) > g'(\overline{x})]) \vee \dot{f}(\overline{x}) > \dot{g}(\overline{x}) \vee \hat{f}(\overline{x}) > \hat{g}(\overline{x}))) \ \vee && \text{(e)} \\
&\quad (h(\overline{x}) \geqslant 0 \wedge p(\overline{x}) \geqslant 0 \wedge \hat{f}'(\overline{x}) \geqslant \hat{g}'(\overline{x}) \wedge \\
&\qquad \dot{f}(\overline{x}) + \lfloor b^{g'(\overline{x})} \rfloor \lfloor b^{p(\overline{x})} \rfloor \hat{f}(\overline{x}) > \dot{g}(\overline{x}) + \lfloor b^{g'(\overline{x})} \rfloor \hat{g}(\overline{x})) \ \big) && \text{(f)}
\end{aligned}
$$

The encodings of comparisons are sound.

▶ **Lemma 3.** *If* $[f(\overline{x}) > g(\overline{x})]$ *holds then* $[\alpha](f(\overline{x})) > [\alpha](g(\overline{x}))$ *for all assignments* $\alpha$.

The following example shows that FBIs are not closed under addition and composition, which complicates the evaluation of a term interpretation.

▶ **Example 4.** The sum $2^x + 2^y$ of the FBIs $2^x$ and $2^y$ has no FBI representation. Substituting the FBI $2^y + 1$ for $x$ in the FBI $2^x x$ results in $2^{2^y+1}(2^y + 1) = 2^{2^y+y+1} + 2^{2^y+1}$, which also has no equivalent FBI representation.

We thus define under- and overapproximations for arithmetic operations.

▶ **Definition 5.**

(a) Multiplication of an FBI by a scalar again yields an FBI, i.e.

$$f(\overline{x})\, a = \sum_{1 \leqslant i \leqslant n} x_i f_i a + f_0 a + b^{f'(\overline{x})} \Big( \sum_{1 \leqslant i \leqslant n} x_i \hat{f}_i a + \hat{f}_0 a \Big)$$

(b) For addition, we first introduce $\mathrm{fmin}(f, g)$ and $\mathrm{fmax}(f, g)$ as the coefficient-wise minimum and maximum of FBIs $f$ and $g$, respectively. These lower (upper) bounds admit the approximations

$$
\begin{aligned}
f(\overline{x}) +_\mu g(\overline{x}) &= \sum_{1 \leqslant i \leqslant n} x_i (f_i + g_i) + (f_0 + g_0) + b^{e_\mu(\overline{x})} \Big( \sum_{1 \leqslant i \leqslant n} x_i (\hat{f}_i + \hat{g}_i) + (\hat{f}_0 + \hat{g}_0) \Big) \\
f(\overline{x}) +_\nu g(\overline{x}) &= \sum_{1 \leqslant i \leqslant n} x_i (f_i + g_i) + (f_0 + g_0) + b^{e_\nu(\overline{x})} \Big( \sum_{1 \leqslant i \leqslant n} x_i (\hat{f}_i + \hat{g}_i) + (\hat{f}_0 + \hat{g}_0) \Big)
\end{aligned}
$$

with $e_\mu(\overline{x})$ abbreviating $\hat{f}(\overline{x}) = 0 \ ? \ g'(\overline{x}) : \big(\hat{g}(\overline{x}) = 0 \ ? \ f'(\overline{x}) : \mathrm{fmin}(f', g')(\overline{x})\big)$ and $e_\nu(\overline{x})$ abbreviating $\hat{f}(\overline{x}) = 0 \ ? \ g'(\overline{x}) : \big(\hat{g}(\overline{x}) = 0 \ ? \ f'(\overline{x}) : \mathrm{fmax}(f', g')(\overline{x})\big)$.

(c) To approximate multiplication of an expression of the form $b^{g'(\overline{x})}$ with $f(\overline{x})$ by an FBI, we may use

$$b^{g'(\overline{x})} \cdot_\mu f(\overline{x}) = \hat{f}(\overline{x}) > 0 \;?\; \dot{f}(\overline{x}) + b^{f'(\overline{x})+_\mu g'(\overline{x})}\hat{f}(\overline{x}) : b^{g'(\overline{x})}\dot{f}(\overline{x})$$

$$b^{g'(\overline{x})} \cdot_\nu f(\overline{x}) = \hat{f}(\overline{x}) > 0 \;?\; b^{f'(\overline{x})+_\nu g'(\overline{x})}\Big(\sum_{1\leqslant i\leqslant n} x_i(\hat{f}_i + f_i) + (\hat{f}_0 + f_0)\Big) : b^{g'(\overline{x})}\dot{f}(\overline{x})$$

(d) Finally we can give approximations for the composition $f(\overline{g})(\overline{x}) = f(g_1(\overline{x}), \ldots, g_n(\overline{x}))$:

$$f(\overline{g})_\mu(\overline{x}) = \sum_{1\leqslant i\leqslant n}^{\mu} g_i(\overline{x})f_i +_\mu f_0 +_\mu b^{f'(\overline{g})_\mu(\overline{x})} \cdot_\mu \Big(\sum_{1\leqslant i\leqslant n}^{\mu} g_i(\overline{x})\hat{f}_i +_\mu \hat{f}_0\Big)$$

The overapproximation $f(\overline{g})_\nu(\overline{x})$ is obtained by replacing $\mu$ by $\nu$ in the above expression.

(e) Let $t$ be a term and $\mathcal{A}$ an FBI algebra. We define FBIs $\mu_\mathcal{A}(t)$ and $\nu_\mathcal{A}(t)$ such that $\mu_\mathcal{A}(t) = t$ if $t \in \mathcal{V}$ and $\mu_\mathcal{A}(t) = f_\mathcal{A}(\mu_\mathcal{A}(t_1), \ldots, \mu_\mathcal{A}(t_n))_\mu$ if $t = f(t_1, \ldots, t_n)$. The overapproximation $\nu_\mathcal{A}(t)$ is defined similarly.

Definition 5 yields valid over- and underapproximations.

▶ **Lemma 6.** *If $\mathcal{A}$ is an FBI algebra and $t$ a term then $[\alpha](\mu_\mathcal{A}(t)) \leqslant [\alpha]_\mathcal{A}(t) \leqslant [\alpha](\nu_\mathcal{A}(t))$ for all assignments $\alpha$.*

The following example illustrates Definition 5.

▶ **Example 7.** We consider the cases for addition and multiplication.

(b) We have $\mathrm{fmin}(x+1, x) = x$ and $\mathrm{fmax}(x+1, x) = x+1$, thus $2^{x+1}y +_\mu 2^x(z+1) = 2^x(y+z+1)$ but $2^{x+1}y +_\nu 2^x(z+1) = 2^{x+1}(y+z+1)$.

In certain pathological cases the approximations of addition are not commutative. To be more precise, the resulting FBIs may be syntactically different but denote the same elementary function. For instance, $2^x \cdot 0 +_\mu 2^{x+1} \cdot 0 = 2^{x+1} \cdot 0$ while $2^{x+1} \cdot 0 +_\mu 2^x \cdot 0 = 2^x \cdot 0$. Still, we do not regard this a problem for our application as the encoding of comparisons takes these cases into account.

(c) For multiplication we have $2^{x+1} \cdot_\mu 2^{2^x} = 2^{(x+1)+_\mu 2^x} = 2^{x+1+2^x}$ and $2^{x+1} \cdot_\nu 2^{2^x} = 2^{x+1+2^x}$, the approximation is thus precise in these cases. On the other hand, as $(x+1) +_\mu 2^x = (x+1) +_\nu 2^x = x+1+2^x$ we have $2^{x+1} \cdot_\mu (z+1+2^{2^x}y) = z+1+2^{x+1+2^x}y$, while $2^{x+1} \cdot_\nu (z+1+2^{2^x}y) = 2^{x+1+2^x}(y+z+1)$.

The following example shows that in practice our approximations are very accurate, i.e., for the motivating example they are exact, i.e., we get the following constraints

$$\begin{array}{ccc} x+5 > x & 2^2x > 2 & 2^{2^2} > 4 \\ 2x+y+5 > 2x+y+3 & 2^{x+2}y > y+1+2^x2y & 2^{2^{x+2}} > 2^{x+2+2^x} \\ 2^x(2y+z+1) > 1+2^x(2y+z) & & \end{array}$$

Monotonicity of an FBI $f(\overline{x})$ is expressed by $\mathrm{mon}(f(\overline{x})) = \bigwedge_{1\leqslant i\leqslant n} \mathrm{mon}_i(f(\overline{x}))$ where $\mathrm{mon}_i(f(\overline{x})) = f_i > 0 \vee \hat{f}_i > 0 \vee (\mathrm{mon}_i(f'(\overline{x})) \wedge \hat{f}(\overline{x}) > 0)$. An FBI $f(\overline{x})$ is well-defined if $[f(\overline{x}) > 0]$ holds. The main result of this section can now be stated as follows.

▶ **Theorem 8.** *Let $\mathcal{R}$ be a TRS over a signature $\mathcal{F}$ and $\mathcal{A}$ be an FBI algebra on $\mathcal{F}$. If*

$$\bigwedge_{\ell\to r\in\mathcal{R}} [\mu_\mathcal{A}(\ell) > \nu_\mathcal{A}(r)] \wedge \bigwedge_{f\in\mathcal{F}} \big([f_\mathcal{A}(\overline{x}) > 0] \wedge \mathrm{mon}(f_\mathcal{A}(\overline{x}))\big)$$

*holds then $\mathcal{R}$ is terminating.*

| method | YES | avg. time | Lescanne's Example | [4, Example 1.1] | [3, Fig. 1] |
|---|---|---|---|---|---|
| poly | 125 | 0.3 | (0.2) | (0.4) | (0.3) |
| fbi | 41 | 29.7 | *1443.4* | *731.0* | *13540.5* |
| fbi[d] | 170 | 4.7 | 16.1 | 10.0 | 27.8 |
| fbi[d+] | 174 | 4.2 | 8.9 | 7.9 | 24.1 |

**Table 1** Experimental Results for FBI Algebras.

Finally we address the remaining problem of finding suitable coefficients. To this end we fix the depth of the FBIs used for interpreting function symbols by some heuristics and consider $f_0, \dots, f_n, \hat{f}_0, \dots, \hat{f}_n$ in (1) as *unknowns* in the natural numbers. The encodings from before then reduce the search for coefficients to finding models in the SMT logic QF_NIA, i.e., existentially quantified non-linear integer arithmetic, for which tools exist.

## 3 Limitations

There are TRSs where FBI termination proofs require interpretations of arbitrary depth.

▶ **Example 9.** Let $\mathcal{R}_n$ for $n > 0$ consist of the rules

$$x + \mathsf{0} \to x \qquad x + \mathsf{s}(y) \to \mathsf{s}(x + y) \qquad \mathsf{exp}_0(x) \to x$$
$$\mathsf{exp}_{i+1}(\mathsf{0}) \to \mathsf{exp}_i(\mathsf{s}(\mathsf{0})) \qquad \mathsf{exp}_{i+1}(\mathsf{s}(x)) \to \mathsf{exp}_i(\mathsf{exp}_1(x) + \mathsf{exp}_1(x))$$

for all $0 \leqslant i < n$. Termination of $\mathcal{R}_n$ can be shown by the FBI algebra $\mathcal{A}$ with base $b = 2$ and interpretations $\mathsf{0}_\mathcal{A} = 1$, $\mathsf{s}_\mathcal{A}(x) = x + 1$, $x +_\mathcal{A} y = x + 2y$, and $\mathsf{exp}_{i,\mathcal{A}}(x) = \exp_2^i(2x + 1)$ where $\exp_2^i(x)$ denotes $i$-fold exponentiation with base 2, i.e., $\exp_2^0(x) = x$ and $\exp_2^{i+1}(x) = 2^{\exp_2^i(x)}$. It is easy to see that any FBI algebra that orients $\mathcal{R}_n$ needs to have at least depth $n$.

It can be shown that already $\mathcal{R}_1$ admits multiple exponential complexity. As to be expected, actually any TRS compatible with an FBI algebra is bounded by a multiple exponential function. A more precise upper bound is given by the following lemma.

▶ **Lemma 10.** *For any TRS $\mathcal{R}$ compatible with an FBI algebra $\mathcal{A}$ having base $b$ and maximal depth $d - 1$,* $\mathrm{dh}_\mathcal{R}(n) \in \exp_b^{d\,n}(\mathcal{O}(n))$.

## 4 Experimental Results

We implemented FBIs in the termination tool T$_{\text{T}}$T$_2$ [2] version 1.15. For experiments[1] we considered the 1463 TRSs in the Standard TRS category of the Termination Problems Data Base (TPDB 8.0.7)[2] and examples from the dedicated literature. If a TRS could not be handled within 60 seconds, the execution of T$_{\text{T}}$T$_2$ was aborted.

Table 1 compares the power of FBIs (of depth at most 2) with linear polynomial interpretations when used in direct termination proofs (orient all rules by a single interpretation). For numbers in parentheses T$_{\text{T}}$T$_2$ was not successful. The expressions in brackets indicate which heuristics have been used. FBIs as well as linear interpretations use two bits to encode coefficients and seven bits for arithmetic evaluations.

[1] Details available from `http://cl-informatik.uibk.ac.at/ttt2/ordinals`
[2] Available from `http://termcomp.uibk.ac.at`.

Our experiments show the need for the heuristic limiting the depth of the FBIs (setting [d] in Table 1). We have also experimented with other heuristics [d+] but they are much less effective, i.e., they either slightly decrease the execution time or increase the number of systems shown terminating but are not explained here. The systems where FBIs succeed but linear polynomials fail often require interpretation functions of non-linear shape.

## 5 Related Work

Lescanne proposed elementary functions for proving (AC-)termination but his implementation is limited to checking the orientation of rules for *given* interpretations [3]. Lucas has achieved partial progress by considering so-called linear elementary interpretations (LEIs) of the shape $A(\overline{x}) + B(\overline{x})^{C(\overline{x})}$ where $A(\overline{x})$, $B(\overline{x})$, and $C(\overline{x})$ are linear polynomials [4]. He proposes an approach based on rewriting, constraint logic programming (CLP), and constraint satisfaction problems (CSPs) to also *find* suitable interpretation functions but leaves an actual implementation of his method as future work.

## 6 Conclusion

Our findings are related to Problem #28 in the RTA List of Open Problems,[3] which asks to "develop effective methods to decide whether a system decreases with respect to some exponential interpretation". In addition our contribution admits the search for suitable interpretations.

Generalizing elementary interpretations to a non-fixed base is an obvious choice for future work. However, we anticipate that suitable approximations will neither give further deep insights nor significantly improve termination proving power and hence we propose a different line of research, viz., the study how to employ them for AC termination.

Since non-linear polynomials give rise to an exponential size SMT encoding, such interpretations are hardly used within termination tools. We anticipate that suitable approximations could improve the performance of these implementations.

### Acknowledgments

We thank the program committee for useful comments.

## References

1 F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, Cambridge, 1998.
2 M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In *Proc. 20th RTA*, volume 5595 of *LNCS*, pages 295–304, 2009.
3 P. Lescanne. Termination of rewrite systems by elementary interpretations. *Formal Aspects of Computing*, 7(1):77–90, 1995.
4 S. Lucas. Automatic proofs of termination with elementary interpretations. In *Proc. 9th PROLE*, volume 258 of *ENTCS*, pages 41–61, 2009.

[3] `http://www.win.tue.nl/rtaloop/`