13th International Workshop on Termination (WST 2013)

Johannes Waldmann (Editor)

WST 2013, August 29–31, 2013, Bertinoro, Italy

Preface

This volume contains the informal proceedings of the *13th International Workshop on Termination*, to be held August 29–31, 2013, at the Centro Residenziale Universitario di Bertinoro, Italy.

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.

This year, for the first time, WST takes place jointly with the *3rd International Workshop on Foundational and Practical Aspects of Resource Analysis* (FOPARA).

Previously, WST was held at St. Andrews (1993), La Bresse (1995), Ede (1997), Dagstuhl (1999), Utrecht (2001), Valencia (2003), Aachen (2004), Seattle (2006), Paris (2007), Leipzig (2009), Edinburgh (2010), and Obergurgl (2012).

The 13th Workshop on Termination consists of 16 regular submissions, contained in this volume, and an invited talk by Byron Cook on *Beyond Termination*.

I would like to take this opportunity to thank all those that helped to prepare and run the workshop: the participants, the members of the program committee, and especially the local organisers.

Leipzig, August 27, 2013 Johannes Waldmann.

Program Committee

Evelyne Contejean	LRI, CNRS, Univ Paris-Sud, Orsay
Carsten Fuhs	University College London
Alfons Geser	HTWK Leipzig
Jürgen Giesl	RWTH Aachen
Sergio Greco	University of Calabria
Nao Hirokawa	Japan Advanced Institute of Science and Technology
Dieter Hofbauer	ASW BA Saarland
Georg Moser	Universität Innsbruck
Albert Rubio	Universitat Politecnica de Catalunya
Peter Schneider-Kamp	Syddansk Universitet
Johannes Waldmann (chair)	HTWK Leipzig
Florian Zuleger	TU Wien

Local Organisation

Ugo Dal Lago	Università di Bologna
Monica Michelacci	CEU Bertinoro
Roberta Partisani	CEU Bertinoro

Table of Contents

Small Polynomial Path Orders in TCT*

Martin Avanzini[1], Georg Moser[1], and Michael Schaper[1]

1 **Institute of Computer Science,**
University of Innsbruck, Austria
{martin.avanzini,georg.moser,michael.schaper}@uibk.ac.at

Abstract

1998 ACM Subject Classification F.2.2, F.4.1, F.4.2, D.2.4, D.2.8

Keywords and phrases Runtime Complexity, Polynomial Time Functions, Rewriting

1 Introduction

In [2] we propose the *small polynomial path order* (sPOP* for short). This order provides a characterisation of the class of *polytime computable function* via term rewrite systems (TRSs for short). Any polytime computable function is expressible as a constructor TRS which is compatible with (an instance of) sPOP*. On the other hand, any function defined by a constructor TRS compatible with sPOP* is polytime computable. This order has also ramifications in the *automated complexity analysis* of rewrite systems. The *innermost runtime complexity* of any constructor TRS $\mathcal{R}$ compatible with sPOP* lies in $O(n^d)$. Here $d \in \mathbb{N}$ refers to the maximal *depth of recursion* of defined symbols f in $\mathcal{R}$.

This work deals with the implementation of sPOP* in the *Tyrolean complexity tool*[1] (TCT for short). The order has been extended to relative rewriting, and takes also *usable arguments* [6] into account. As by-product, we obtain a form of *reduction pair* from sPOP*. Such reduction pairs can be used in the *dependency pair* analysis of Hirokawa and the second author [5] and Noschinski et al. [7]. For details and proofs we refer the reader to [1].

2 Small Polynomial Path Orders

We assume familiarity with rewriting [3]. Let $\mathcal{R}$ be a TRS over a signature $\mathcal{F}$, with *defined symbols* in $\mathcal{D}$. *Constructors* are denoted by $\mathcal{C} := \mathcal{F} \setminus \mathcal{D}$. Further, let $\mathcal{K} \subseteq \mathcal{D}$ denote a set of *recursive symbols*, and let $\gtrsim$ denote a (quasi)-precedence on $\mathcal{F}$. We denote by $>$ and $\sim$ the proper order and equivalence underlying $\gtrsim$. We call the precedence $\gtrsim$ *admissible* for sPOP* if it retains the partitioning of $\mathcal{F}$ in the following sense. If $f \sim g$ then $f \in \mathcal{C}$ implies $g \in \mathcal{C}$, likewise, $f \in \mathcal{K}$ implies $g \in \mathcal{K}$. Small polynomial path orders embody the principle of *predicative recursion* [4] on compatible TRSs. To this end, arguments of every function symbol are partitioned into normal and safe ones. Notationally we write $f(t_1, \dots, t_k; t_{k+1}, \dots, t_{k+l})$ with *normal* arguments $t_1, \dots, t_k$ separated from *safe* arguments $t_{k+1}, \dots, t_{k+l}$ by a semicolon. For constructors, we fix that all argument positions are safe. We define the equivalence $\approx_\mathsf{s}$ on terms respecting this separation as follows: $s \approx_\mathsf{s} t$ holds if $s = t$ or $s = f(s_1, \dots, s_k; s_{k+1}, \dots, s_{k+l})$ and $t = g(t_1, \dots, t_k; t_{k+1}, \dots, t_{k+l})$ where $f \sim g$ and $s_i \approx_\mathsf{s} t_i$ holds for all $i = 1, \dots, k+l$. We write $s \trianglerighteq\!\!\!\!/_{\approx} t$ if t is a subterm (modulo $\approx_\mathsf{s}$) of a normal argument of s.

* This work is partially supported by FWF (Austrian Science Fund) project I-608-N18.

[1] TCT is open source and available from http://cl-informatik.uibk.ac.at/software/tct.

The following definition introduces small polynomial path orders, also accounting for parameter substitution [2]. We denote by $\mathcal{T}(\mathcal{F}^{<f}, \mathcal{V})$ the set of terms built from variables and function symbols $\mathcal{F}^{<f} := \{g \in \mathcal{F} \mid f > g\}$.

▶ **Definition 2.1.** Let $s = f(s_1, \dots, s_k; s_{k+1}, \dots, s_{k+l})$. Then $s >_{\mathsf{spop}^*_{\mathsf{ps}}} t$ if either

1) $s_i \gtrsim_{\mathsf{spop}^*_{\mathsf{ps}}} t$ for some argument s_i of s.
2) $f \in \mathcal{D}$, $t = g(t_1, \dots, t_m; t_{m+1}, \dots, t_{m+n})$ with $f > g$ and the following conditions hold: (i) $s \not\rhd\!\!\!/\approx t_j$ for all normal arguments t_j of t, (ii) $s >_{\mathsf{spop}^*_{\mathsf{ps}}} t_j$ for all safe arguments t_j of t, and (iii) $t_j \notin \mathcal{T}(\mathcal{F}^{<f}, \mathcal{V})$ for at most one $j \in \{1, \dots, k+l\}$.
3) $f \in \mathcal{K}$, $t = g(t_1, \dots, t_k; t_{k+1}, \dots, t_{k+l})$ with $f \sim g$ and the following conditions hold: (i) $\langle s_1, \dots, s_k\rangle >^{\mathsf{prod}}_{\mathsf{spop}^*_{\mathsf{ps}}} \langle t_1, \dots, t_k\rangle$, (ii) $s >_{\mathsf{spop}^*_{\mathsf{ps}}} t_j$ for all safe arguments t_j ($j = k+1, \dots, k+l$), and (iii) $t_j \in \mathcal{T}(\mathcal{F}^{<f}, \mathcal{V})$ for all $j = 1, \dots, k+l$.

Here $\gtrsim_{\mathsf{spop}^\star}$ denotes the extension of $>_{\mathsf{spop}^*_{\mathsf{ps}}}$ by safe equivalence $\approx_{\mathsf{s}}$. Further, $>^{\mathsf{prod}}_{\mathsf{spop}^*_{\mathsf{ps}}}$ denotes the product extension of $>_{\mathsf{spop}^*_{\mathsf{ps}}}$: $\langle s_1, \dots, s_n\rangle >^{\mathsf{prod}}_{\mathsf{spop}^*_{\mathsf{ps}}} \langle t_1, \dots, t_n\rangle$ if $s_i \gtrsim_{\mathsf{spop}^\star} t_i$ for all $i = 1, \dots, n$, and $s_{i_0} >_{\mathsf{spop}^*_{\mathsf{ps}}} t_{i_0}$ for some $i_0 \in \{1, \dots, n\}$.

The *depth of recursion* $\mathsf{rd}_{\gtrsim,\mathcal{K}}(f)$ of $f \in \mathcal{F}$ is recursively defined by $\mathsf{rd}_{\gtrsim,\mathcal{K}}(f) := 1 + d$ if $f \in \mathcal{K}$ and $\mathsf{rd}_{\gtrsim,\mathcal{K}}(f) := d$ if $f \notin \mathcal{K}$, where $d = \max\{0\} \cup \{\mathsf{rd}_{\gtrsim,\mathcal{K}}(g) \mid f > g\}$.

▶ Proposition 2.2 ([2]). Let $\mathcal{R}$ be a constructor TRS compatible with an instance $>_{\mathsf{spop}^*_{\mathsf{ps}}}$ based on an admissible precedence $\gtrsim$ with recursive symbols $\mathcal{K}$. Then the innermost runtime complexity of $\mathcal{R}$ lies in $O(n^d)$, where $d = \max\{0\} \cup \{\mathsf{rd}_{\gtrsim,\mathcal{K}}(f) \mid f \in \mathcal{D}\}$.

3 Polynomial Path Orders as Complexity Processors

Our tool TCT operates internally on *complexity problems* $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle$, where $\mathcal{S}, \mathcal{W}, \mathcal{Q}$ are TRSs and $\mathcal{T}$ denotes a set of ground terms. The set $\mathcal{T}$ is called the set of *starting terms* of $\mathcal{P}$. Throughout the following, this complexity problem is kept fixed. The *complexity (function)* $\mathsf{cp}_{\mathcal{P}} : \mathbb{N} \to \mathbb{N}$ of $\mathcal{P}$ is defined as the partial function

$$\mathsf{cp}_{\mathcal{P}}(n) := \max\{\mathsf{dh}(t, \xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}}) \mid \exists t \in \mathcal{T} \text{ and } |t| \leqslant n\} \,.$$

Here $\xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}} := \xrightarrow{\mathcal{Q}}^*_{\mathcal{W}} \cdot \xrightarrow{\mathcal{Q}}_{\mathcal{S}} \cdot \xrightarrow{\mathcal{Q}}^*_{\mathcal{W}}$ denotes the *$\mathcal{Q}$-restricted rewrite relation of $\mathcal{S}$ relative to $\mathcal{W}$*, where $\xrightarrow{\mathcal{Q}}_{\mathcal{R}}$ is the restriction of $\to_{\mathcal{R}}$ where all proper subterms of the redex are in $\mathcal{Q}$ normal form. We call the complexity problem $\mathcal{P}$ a *runtime complexity problem* if all terms in $\mathcal{T}$ are *basic*, i.e., of the form $f(t_1, \dots, t_k)$ for $f \in \mathcal{D}$ and constructor terms $t_1, \dots, t_k$. It is called an *innermost complexity problem* if all normal forms of $\mathcal{Q}$ are normal forms of $\mathcal{S} \cup \mathcal{W}$.

A *(complexity) judgement* is a statement $\vdash \mathcal{P} : f$ where $\mathcal{P}$ is a complexity problem and $f : \mathbb{N} \to \mathbb{N}$. This judgement is *valid* if $\mathsf{cp}_{\mathcal{P}}$ is defined on all inputs, and $\mathsf{cp}_{\mathcal{P}} \in \mathcal{O}(f)$. A *complexity processor* is an inference rule

$$\frac{\vdash \mathcal{P}_1 : f_1 \quad \cdots \quad \vdash \mathcal{P}_n : f_n}{\vdash \mathcal{P} : f} \,.$$

This processor is *sound* if $\vdash \mathcal{P} : f$ is valid whenever the judgements $\vdash \mathcal{P}_1 : f_1, \dots, \vdash \mathcal{P}_n : f_n$ are valid. We follow the usual convention and annotate side conditions as premises to inference rules. An inference of $\vdash \mathcal{P} : f$ using sound processors is called a *complexity proof*. If this inference admits no assumptions, then the judgement $\vdash \mathcal{P} : f$ is valid.

In the following, we propose a complexity processors based on sPOP* that operates on innermost runtime complexity problems. In essence, this processor requires that $\mathcal{W} \subseteq \gtrsim_{\mathsf{spop}^*_{\mathsf{ps}}}$

and $\mathcal{S} \subseteq >_{\mathsf{spop}^*_{\mathsf{ps}}}$ holds, and that $\mathcal{W}$ and $\mathcal{S}$ are constructor TRSs. If these requirements are met, then the complexity of $\mathcal{P}$ lies in $\mathcal{O}(n^d)$ for $d \in \mathbb{N}$ the maximal depth of recursion as in Proposition 2.2. To weaken monotonicity requirements, we integrate *argument filterings* into the order. The argument filtering is constrained, so that in derivations of starting terms, no redex is removed. Compare [6], where *μ-monotone orders* are used in a similar spirit.

An argument filtering (for a signature $\mathcal{F}$) is a mapping π that assigns to every k-ary function symbol $f \in \mathcal{F}$ an argument position $i \in \{1, \dots, k\}$ or a (possibly empty) list $[i_1, \dots, i_l]$ of argument positions with $1 \leqslant i_1 < \cdots < i_l \leqslant k$. If $\pi(f)$ is a list we say that π is *non-collapsing* on f. Below π always denotes an argument filtering. For each $f \in \mathcal{F}$, let f_π denote a fresh function symbol associated with f. We define $\mathcal{F}_\pi := \{f_\pi \mid f \in \mathcal{F} \text{ and } \pi(f) = [i_1, \dots, i_l]\}$. The sets $\mathcal{D}_\pi$ and $\mathcal{C}_\pi$ denote the defined symbols and constructors in $\mathcal{F}_\pi$, as given by the restriction of $\mathcal{F}_\pi$ to symbols f_π associated with $f \in \mathcal{D}$ and $f \in \mathcal{C}$ respectively. We denote by π also its extension to terms: $\pi(t) := t$ if t is a variable, and for $t = f(t_1, \dots, t_k)$, $\pi(t) := \pi(t_i)$ if $\pi(f) = i$ and $f(\pi(t_{i_1}), \dots, \pi(t_{i_l}))$ if $\pi(f) = [i_1, \dots, i_l]$. For an order $\succ$ on terms over $\mathcal{F}_\pi$, we define $s \succ^\pi t$ if $\pi(s) \succ \pi(t)$ holds.

A map $\mu \colon \mathcal{F} \to \mathcal{P}(\mathbb{N})$ with $\mu(f) \subseteq \{1, \dots, k\}$ for every k-ary $f \in \mathcal{F}$ is called a *replacement map* on $\mathcal{F}$. The set $\mathcal{P}\mathrm{os}_\mu(t)$ of *μ-replacing positions* in a term t is given by $\mathcal{P}\mathrm{os}_\mu(t) := \varnothing$ if t is a variable, and $\mathcal{P}\mathrm{os}_\mu(t) := \{\epsilon\} \cup \{i{\cdot}p \mid i \in \mu(f) \text{ and } p \in \mathcal{P}\mathrm{os}_\mu(t_i)\}$ if $t = f(t_1, \dots, t_k)$. For a binary relation $\to$ on terms we denote by $\mathcal{T}_\mu(\to)$ the set of terms t where sub-terms at non-μ-replacing positions are in normal form: $t \in \mathcal{T}_\mu(\to)$ if for all positions p in t, if $p \notin \mathcal{P}\mathrm{os}_\mu(t)$ then $t|_p \to u$ does not hold for any term u. Let $\mathcal{R}$ denote a set of rewrite rules. A replacement map μ is called a *usable replacement map* for $\mathcal{R}$ in $\mathcal{P}$, if $\to^*_{\mathcal{S}\cup\mathcal{W}}(\mathcal{T}) \subseteq \mathcal{T}_\mu(\xrightarrow[\mathcal{Q}]{}_{\mathcal{R}})$. For a usable replacement map μ and argument filtering π, we say that π *agrees with* μ if for all function symbols f in the domain of μ, either (i) $\pi(f) = i$ and $\mu(f) \subseteq \{i\}$ or otherwise (ii) $\mu(f) \subseteq \pi(f)$ holds.

▶ **Theorem 3.1.** *Let $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle$ be an innermost complexity problem, where $\mathcal{S}$ and $\mathcal{W}$ are constructor TRSs. Let π denote an argument filtering on the symbols in $\mathcal{P}$ that agrees with a usable replacement map for $\mathcal{S}$ in $\mathcal{P}$, and that is non-collapsing on defined symbols of $\mathcal{S}$. Let $\mathcal{K}_\pi \subseteq \mathcal{D}_\pi$ denote a set of recursive function symbols, and $\gtrsim$ an admissible precedence on $\mathcal{F}_\pi$. The following processor is sound, for $d := \max\{0\} \cup \{\mathsf{rd}_{\gtrsim,\mathcal{K}_\pi}(f_\pi) \mid f_\pi \in \mathcal{F}_\pi\}$.*

$$\frac{\mathcal{S} \subseteq >^\pi_{\mathsf{spop}^*_{\mathsf{ps}}} \quad \mathcal{W} \subseteq \gtrsim^\pi_{\mathsf{spop}^*_{\mathsf{ps}}}}{\vdash \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle \colon n^d} \ .$$

We remark that the restriction that π is non-collapsing on defined symbols of $\mathcal{S}$ is essential, compare also [1]. In TCT, Theorem 3.1 is usually applied in combination with the *relative decomposition processor* [1], This processor allows the iterated combination of different techniques, by translating the judgement $\vdash \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle \colon f$ into the two judgements $\vdash \langle \mathcal{S}_1/\mathcal{S}_2 \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle \colon f$ and $\vdash \langle \mathcal{S}_2/\mathcal{S}_1 \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle \colon f$, where $\mathcal{S}_1 \cup \mathcal{S}_2 = \mathcal{S}$. Theorem 3.1 is tight, in the sense that for any $d \in \mathbb{N}$ one can find a complexity problem $\mathcal{P}$ that satisfies the pre-conditions, and whose complexity function lies in $\Omega(n^d)$ [2]. The next example illustrates the application of Theorem 3.1.

▶ **Example 3.2.** Consider the innermost complexity problem $\mathcal{P}^\sharp_{\mathsf{log}} = \langle \mathcal{S}^\sharp_{\mathsf{log}}/\mathcal{W}_{\mathsf{log}}, \mathcal{S}^\sharp_{\mathsf{log}} \cup \mathcal{W}_{\mathsf{log}}, \mathcal{T}^\sharp_{\mathsf{log}}\rangle$ where the TRS $\mathcal{S}^\sharp_{\mathsf{log}}$ consisting of the rewrite rules

$$\mathsf{half}^\sharp(\mathsf{s}(\mathsf{s}(x))) \to \mathsf{half}^\sharp(x) \qquad \mathsf{log}^\sharp(\mathsf{s}(\mathsf{s}(x))) \to \mathsf{log}^\sharp(\mathsf{s}(\mathsf{half}(x))) \ ,$$

the TRS $\mathcal{W}_{\mathsf{log}}$ consists of the rules

$$\mathsf{half}(0) \to 0 \qquad \mathsf{half}(\mathsf{s}(\mathsf{s}(x))) \to \mathsf{s}(\mathsf{half}(x)) \ ,$$

and $\mathcal{T}^\sharp$ consists of the basic terms $f(\mathsf{s}^n(0))$ for $n \in \mathbb{N}$ and $f \in \{\mathsf{half}^\sharp, \mathsf{log}^\sharp\}$. Observe that the rules in $\mathcal{S}^\sharp_{\mathsf{log}}$ can only be applied on root positions in derivations starting from $\mathcal{T}^\sharp_{\mathsf{log}}$. It follows that the map $\mu_\varnothing$, which maps any function symbol f in $\mathcal{P}^\sharp_{\mathsf{log}}$ to $\varnothing$, is a usable replacement map for $\mathcal{S}^\sharp_{\mathsf{log}}$ in $\mathcal{P}^\sharp_{\mathsf{log}}$. Consider the argument filtering π with $\pi(\mathsf{half}) = 1$ and $\pi(f) = [1]$ for $f \neq \mathsf{half}$. Note that π trivially agrees with $\mu_\varnothing$. Using $\mathcal{K}_\pi := \{\mathsf{half}^\sharp, \mathsf{log}^\sharp\}$ and the empty precedence one can show $\mathcal{S}^\sharp_{\mathsf{log}} \subseteq >^\pi_{\mathsf{spop}^*_{\mathsf{ps}}}$ and $\mathcal{W}_{\mathsf{log}} \subseteq \gtrsim^\pi_{\mathsf{spop}^*_{\mathsf{ps}}}$. Trivially $\mathsf{rd}_{\gtrsim,\mathcal{K}_\pi}(\mathsf{s}_\pi) = \mathsf{rd}_{\gtrsim,\mathcal{K}_\pi}(0_\pi) = 0$, as neither $\mathsf{half}^\sharp_\pi > \mathsf{log}^\sharp_\pi$ nor $\mathsf{log}^\sharp_\pi > \mathsf{half}^\sharp_\pi$ holds, we see that $\mathsf{rd}_{\gtrsim,\mathcal{K}_\pi}(\mathsf{half}^\sharp_\pi) = \mathsf{rd}_{\gtrsim,\mathcal{K}_\pi}(\mathsf{log}^\sharp_\pi) = 1$. By Theorem 3.1, the complexity of $\mathcal{P}^\sharp_{\mathsf{log}}$ is bounded by a linear function.

4 Polynomial Path Orders and Dependency Pairs

In TCT, a *dependency pair* problem (*DP* problem for short) is a complexity problem whose strict and weak component contains also *dependency pairs*. Unlike for termination analysis, we allow *compound symbols* in right hand sides of dependency pairs. The purpose of these symbols is to group function calls. The example considered above is a DP problem that was generated by TCT on AG01/#3.7 from the *termination problem data base*[2] (*TPDB* for short). For each k-ary $f \in \mathcal{D}$, let $f^\sharp$ denote a fresh function symbol also of arity k, the *dependency pair symbol* (of f). The least extension of $\mathcal{F}$ to all dependency pair symbols is denoted by $\mathcal{F}^\sharp$. We define $t^\sharp := f^\sharp(t_1, \dots, t_k)$ if $t = f(t_1, \dots, t_k)$ and $f \in \mathcal{D}$, and $t^\sharp := t$ otherwise. For a set of terms T, we denote by $T^\sharp$ the set of *marked terms* $T^\sharp := \{t^\sharp \mid t \in T\}$. Consider the infinite signature $\mathcal{C}\mathsf{om}$ that contains for each $i \in \mathbb{N}$ a fresh constructor symbol $\mathsf{c}_i \in \mathcal{C}\mathsf{om}$ of arity i. Symbols in $\mathcal{C}\mathsf{om}$ are called *compound symbols*. We denote by $\textsc{com}(t)$ the term t, and overload this notation to sequences of terms such that $\textsc{com}(t_1, \dots, t_k) = \mathsf{c}_k(t_1, \dots, t_k)$ for $k \neq 1$. A *dependency pair* (*DP* for short) is a rewrite rule $l^\sharp \to \textsc{com}(r^\sharp_1, \dots, r^\sharp_k)$ where $l, r_1, \dots, r_k \in \mathcal{T}(\mathcal{F}, \mathcal{V})$. Let $\mathcal{S}$ and $\mathcal{W}$ be two TRSs over $\mathcal{T}(\mathcal{F}, \mathcal{V})$, and let $\mathcal{S}^\sharp$ and $\mathcal{W}^\sharp$ be two sets of dependency pairs. A *dependency pair complexity problem*, or simply *DP problem*, is a runtime complexity problem $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp \cup \mathcal{S} / \mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle$ over marked basic terms $\mathcal{T}^\sharp$. We keep the convention that $\mathcal{S}^\sharp$ and $\mathcal{W}^\sharp$ denote dependency pairs. Our notion of a DP problem is general enough to capture images of the transformations proposed in the literature [5, 7] for polynomial complexity analysis, compare [1]. In the following, we suppose $\mathcal{S} = \varnothing$, i.e., the complexity function of $\mathcal{P}^\sharp$ accounts for dependency pairs only. We emphasise that for innermost runtime complexity analysis, TCT always constructs a DP problem of this shape, by either applying the *weightgap condition* [5] or using *dependency tuples* [7] only.

As a consequence of the following simple observation, the argument filtering employed in Theorem 3.1 has to fulfil, besides the non-collapsing condition on defined symbols of $\mathcal{S}^\sharp$, only mild conditions on compound symbols.

▶ **Lemma 4.1.** *Let $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp / \mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle$ be a DP problem. Suppose μ denotes a usable replacement map for dependency pairs $\mathcal{S}^\sharp$ in $\mathcal{P}^\sharp$. Then $\mu_{\textsc{com}}$ is a usable replacement map for $\mathcal{S}^\sharp$ in $\mathcal{P}^\sharp$. Here $\mu_{\textsc{com}}$ denotes the restriction of μ to compound symbols in the following sense: $\mu_{\textsc{com}}(\mathsf{c}_n) := \mu(\mathsf{c}_n)$ for all $\mathsf{c}_n \in \mathcal{C}\mathsf{om}$, and otherwise $\mu_{\textsc{com}}(f) := \varnothing$ for $f \in \mathcal{F}^\sharp$.*

For DP problems, one can remove the non-collapsing condition on the employed argument filtering. The inferred complexity bound is less fine grained however.

▶ **Theorem 4.2.** *Let $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp / \mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle$ be an innermost DP problem, where $\mathcal{S}^\sharp$, $\mathcal{W}^\sharp$ and $\mathcal{W}$ are constructor TRSs. Let μ denote a usable replacement map for $\mathcal{S}^\sharp \cup \mathcal{W}^\sharp$ in $\mathcal{P}^\sharp$, Let*

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

π denote an argument filtering on the symbols in $\mathcal{P}$ that agrees with a usable replacement map for all dependency pairs in $\mathcal{P}^\sharp$. Let $\mathcal{K}_\pi \subseteq \mathcal{D}^\sharp_\pi$ denote a set of recursive function symbols, and $\gtrsim$ an admissible precedence where $c_\pi \sim d_\pi$ only holds for non-compound symbols $c, d \notin \mathsf{Com}$. The following processor is sound, for $d := \max\{0\} \cup \{\mathsf{rd}_{\gtrsim,\mathcal{K}_\pi}(f_\pi) \mid f_\pi \in \mathcal{F}^\sharp_\pi\}$.

$$\frac{\mathcal{S}^\sharp \subseteq >^\pi_{\mathsf{spop}^*_{\mathsf{ps}}} \quad \mathcal{W}^\sharp \cup \mathcal{W} \subseteq \gtrsim^\pi_{\mathsf{spop}^*_{\mathsf{ps}}}}{\vdash \langle \mathcal{S}^\sharp / \mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle : n^{\max(1,2\cdot d)}} \ .$$

We remark that the pre-conditions of the theorem are essential, and the estimated complexity is asymptotically optimal in general [1].

5 Conclusion

In this work we have outlined the implementation of sPOP* in TCT. We conclude with an empirical evaluation of this method. In Table 1 we contrast sPOP* to *matrix interpretations* (MI for short). Here the subscript DP denotes that the input is first transformed into a DP problem and syntactically simplified, compare [1, Section 14.5]. As testbed we used the 757 well-formed constructor TRSs from the TPDB 8.0.[3]

bound	sPOP*	sPOP$^\star_{\mathsf{DP}}$	MI$_{\mathsf{DP}}$
$\mathcal{O}(1)$	$4_{\backslash 0.17}$	$20_{\backslash 0.28}$	$20_{\backslash 0.27}$
$\mathcal{O}(n^1)$	$20_{\backslash 0.17}$	$72_{\backslash 0.31}$	$98_{\backslash 0.48}$
$\mathcal{O}(n^2)$	$23_{\backslash 0.19}$	$11_{\backslash 0.44}$	$17_{\backslash 4.67}$
$\mathcal{O}(n^3)$	$6_{\backslash 0.23}$	$3_{\backslash 0.60}$	$8_{\backslash 14.7}$
total	$54_{\backslash 0.19}$	$106_{\backslash 0.32}$	$143_{\backslash 1.55}$
maybe	$703_{\backslash 0.34}$	$652_{\backslash 1.20}$	$613_{\backslash 18.3}$

Table 1 Number of oriented problems and average execution time (secs.)

Comparing sPOP* and sPOP$^\star_{\mathsf{DP}}$ we see a significant increase in precision and power. This can be attributed to the relaxed conditions on the employed argument filtering. On the testbed, sPOP$^\star_{\mathsf{DP}}$ cannot cope in power with MI$_{\mathsf{DP}}$, but the average execution time of sPOP$^\star_{\mathsf{DP}}$ is significantly lower. Worthy of note, sPOP$^\star_{\mathsf{DP}}$ and MI$_{\mathsf{DP}}$ are incomparable. Their combination can handle 149 examples.

References

1 M. Avanzini. *Verifying Polytime Computability Automatically.* PhD thesis, University of Innsbruck, Institute for Computer Science, 2013. Submitted. Available at `http://cl-informatik.uibk.ac.at/~zini/publications/`.

2 M. Avanzini, N. Eguchi, and G. Moser. A New Order-theoretic Characterisation of the Polytime Computable Functions. In *Proc. of 10th APLAS*, volume 7705 of *LNCS*, pages 280–295, 2012.

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

4 S. Bellantoni and S. Cook. A new Recursion-Theoretic Characterization of the Polytime Functions. *Computational Complexity*, 2(2):97–110, 1992.

5 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, 2008.

6 N. Hirokawa and G. Moser. Automated Complexity Analysis Based on the Dependency Pair Method. 2012. To appear.

7 L. Noschinski, F. Emmes, and J. Giesl. A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems. In *Proc. of 23rd CADE*, volume 6803 of *LNAI*, pages 422–438. Springer, 2011.

[3] See `http://cl-informatik.uibk.ac.at/software/tct/experiments/wst2013` for full experimental evidence and explanation on the setup.

SAT compilation for Termination Proofs via Semantic Labelling

Alexander Bau[1], Jörg Endrullis[2], and Johannes Waldmann[1]

1 **HTWK Leipzig, Fakultät IMN, 04277 Leipzig, Germany**
2 **Vrije Universiteit Amsterdam, The Netherlands**

Abstract

For the termination method of rule removal by semantic labelling, matrix interpretations and then unlabelling, we give a purely functional specification and apply our CO4 constraint compiler to automatically generate a propositional encoding. This extends a "manual" SAT encoding of this method in Jambox (Endrullis, 2008): we allow sequences of interpretations during labelling.

1 Introduction

Finding parameters for termination proof methods is a constraint satisfaction problem: given a rewrite system, we are looking for a precedence for a path order, or for coefficients of interpretations, etc.

For (finite domain) constraint programming, SAT (propositional satisfiably) takes the role of an assembly language: it gives direct access to the machine (i.e., powerful and highly optimized SAT solver), but it is cumbersome and error-prone for actual programming. Instead, we want to use a high-level specification language, to increase expressiveness and safety.

Two of us (Bau, Waldmann) are developing the CO4 language and compiler [1] that translates Haskell specifications to SAT formulas. We have previously applied CO4 for finding loops in string rewriting. With respect to a manual SAT encoding of TTT2 [5], our compiled formula is larger by a factor of < 10, with a similar factor for run-times of the SAT solver, but the source code of the constraint system is just 1/3 in size.

In this note, we report on an application of CO4 in termination of string rewriting, using the method of semantic labelling, interpretations, and unlabelling. This is well-known [6], and it had been implemented in several termination tools already: Teparla and Torpa used semantic labelling in $\approx$ 2006, by a stochastic search for models.

Jambox [2] $\approx$ 2007 used a "manual" SAT encoding for the following constraint: given a rewrite system R, there is a model M for R and an interpretation I for the M-labelled system R_M and a non-empty $S \subseteq R$ such that each rule in S_M is strictly I-decreasing. We then have relative termination $\mathrm{SN}(S/R)$, that is, we can remove S. Correctness of this method had been formalized for CeTA [4].

We now extend Jambox' implementation by allowing for a sequence of interpretations for the labelled system. This is a conceptually simple modification: instead of one order, use a lexicographic product of several orders. Using the CO4 language, this modification can be expressed directly in the source code (see function `lexi`), and all extra encoding is done by the compiler.

2 Semantic Labelling and Unlabelling

An algebra A is a *model* for a rewrite system R over signature Σ if for each $s \to_R t$, we have $A(s) = A(t)$. If we have a model A for R, we construct a labelled version R_A of R where each function symbol is labelled by the value(s) of its argument(s).

▶ **Example 1.** We consider the string rewriting system $R = \{aa \to aba\}$ over the signature $\Sigma = \{a, b\}$. The algebra A with domain $D = \{0, 1\}$, and interpretation $a_A = \{(0, 1), (1, 1)\}, b_A = \{(0, 0), (1, 0)\}$, is a model for R, and R_A is $\{a_1a_0 \to a_0b_1a_0, a_1a_1 \to a_0b_1a_1\}$.

Termination of R and R_A are equivalent (under some conditions that are true for string rewriting). The point of the method is that termination of R_A may be easier to show that termination of R since we increase the signature, and have more room to maneuver.

In the example, we can see that the number of occurrences of a_1 decreases in each rule application. This can be verified by the (linear, additive) interpretation $\{(a_0, 0), (a_1, 1), (b_1, 0)\}$ for R_A. There is no such interpretation for R.

3 SAT compilation with CO4

CO4 [1] is a high-level declarative language for describing constraint systems. The language is a subset of the purely functional programming language Haskell [3] that includes user-defined algebraic data types and recursive functions defined by pattern matching, as well as higher-order polymorphic types.

Source programs operate on algebraic data (like Booleans, List, Trees) which we call *concrete values*. CO4 compilation creates programs that operate on *abstract values*. An abstract value $\in \mathbb{A}$ represents a set of concrete values of the same type. An abstract value is a tree, where each node contains a sequence of propositional logic formulas. Given a truth assignment σ, the sequence of truth values of these formulas under σ gives a binary number that denotes a constructor. Doing this for each node, a concrete value $\texttt{decode}(a, \sigma)$ is determined.

A high level, parametric constraint system `constraint :: P -> U -> Bool` written in CO4 represents a predicate on `U` depending on a parameter $p \in$ `P`. p is not known a-priori. The compilation and evaluation of a CO4 program is a staged process:

1. The original program, operating on *concrete values* $\mathbb{C}$, is compiled into an abstract program that operates on *abstract values* $\mathbb{A}$.
2. Given an parameter p, the abstract program is evaluated, resulting in an abstract value a (representing a Boolean).
3. The formula $a =$ `True` is given to an external SAT solver. It tries to find a satisfying assignment σ.
4. A value $u \in$ `U` is reconstructed by σ, so that `constraint` p u `= True`.

Values that depend on the u parameter of the top-level constraint are not known during abstract evaluation. Abstract evaluation of case-distinctions on those values has to evaluate (abstractly) all branches and then merge the results into a single abstract value.

Natural numbers may be defined as list of Booleans `type Nat = [Bool]` in CO4. For naturals with bit-width k this encoding ends up in abstract values with depth k. A user-defined function on naturals would require k pattern matches to inspect a number: this leads to long runtimes of the compiled program and the resulting formulas would be large. To avoid those issues, CO4 provides built-in naturals, where a number of bit-width k is encoded by an abstract value with k flags and no arguments. CO4 provides arithmetic and relational operations on naturals as well.

4 Implementation

Complete source code of the termination method is available as part of `https://github.com/jwaldmann/matchbox` (file `MB/Label/SLPO.standalone.hs`).

Here, we just indicate the main types and functions. For string rewriting, we have

```
type Symbol = [Bool]; type Word = [Symbol]
data Mode = Strict | Weak ; data Rule = Mode Word Word ; type SRS = [Rule]
```

where lists are built-in (with their usual Haskell definition). A symbol is represented as a binary string. The length of that string is known at run-time (when the implementation sees the signature of the rewrite system) but not at compile-time.

We often access information that belongs to a symbol. We use binary trees where the symbol encodes a path from the root to a leaf

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
get :: Tree a -> Symbol -> a
get t p = case p of
    []   -> case t of Leaf x -> x
    x:p' -> case t of Branch l r ->
         get (case x of False -> l ; True -> r) p'
```

Symbols will be labelled according to an unknown model, but we know the size of the model. We represent a labelled symbol as the concatenation of a list of known booleans (the original symbol) with a list (of known length) of unknown booleans (the label).

Note that the pattern matches on `p` and on `t` can be resolved at run-time (when generating the SAT formula), but the pattern match on `x` has to be encoded (since `x` depends on the model, which is determined by the solver).

We also represent finite algebras (of size 2^n) the same way: the domain is the set of bit strings of length n, and for each symbol, we have a function from domain to domain:

```
type Model = Tree Func; type Func = Tree Value; type Value = [Bool]
```

We compose functions (and give the most general type):

```
timesF :: Tree a -> Func -> Tree a
timesF f g = case g of
    Leaf w -> Leaf (get f w)
    Branch l r -> Branch (timesF f l) (timesF f r)
```

Given an algebra, we compute the labelled version of rewrite system, as a list of systems (for each original rule, one sub-system containing all its labelled versions). Also, each side of each rule is annotated with its value in the model. These values will be constrained to be equal.

```
labelled :: SRS -> Model -> [ [ ((Value,Value), Rule) ] ]
labelled srs mod =
    let ks = keys ( leftmost mod )
        labelRule u k = case u of
            (l,r) -> case labelledW mod l k of
                ( ltop, l' ) -> case labelledW mod r k of
                    ( rtop, r' ) -> ((ltop,rtop),(l',r'))
    in  map ( \ u ->  map ( \ k -> labelRule u k ) ks    ) srs
```

We omit details on interpretations, and come back to the lexicographic comparison mentioned in the introduction:

```
data Comp = Greater | GreaterEquals | None
comp :: Interpretation -> Rule -> Comp
comps :: [ Interpretation ] -> Rule -> Comp
comps ints u = lexi (map ( \ i -> comp i u) ints )
lexi :: [Comp] -> Comp
lexi cs = case cs of
  [] -> GreaterEquals
  c : cs' -> case c of
    Greater -> Greater; GreaterEquals -> lexi cs'; None -> None
```

This shows the expressiveness of the CO4 language. — The main constraint is

```
data Label = Label Model [ Interpretation ] [ Bool ]
constraint :: SRS -> Label -> Bool
```

where `constraint srs (Label mod ints flags)` is True iff `mod` is a model for `srs` and the lexicographic combination of `ints` is compatible with the labelled system, and removes the flagged rules (of the original system) completely, and at least one rule is flagged.

5 Performance

We describe work-in-progress, so we don't have complete performance data on a larger set of problems. We give an example (that shows the power of the method) and a comparison (that shows the quality of the SAT compilation).

▶ **Example 2.** Our implementation produces this termination proof for $a^2b^2 \to b^3a^3$:

```
matchbox ~/tpdb/tpdb-4.0/SRS/Zantema/z001.srs  -l2,2 --dim 1 --bits 4 --nat
   # 2 bits for model values, 2 interpretations for labelled system
CNF finished (#variables: 21483, #clauses: 82956)
Solver finished in 4.152 seconds (result: True)

model:  3b0 2b1 3b2 1b3
        2a0 0a1 0a2 0a3

labelled system: a0 a1 b3 b0 -> b1 b3 b2 a0 a2 a0
                 a0 a1 b3 b2 -> b1 b3 b0 a2 a0 a2
                 a0 a3 b2 b1 -> b1 b3 b0 a2 a0 a1
                 a0 a2 b1 b3 -> b1 b3 b0 a2 a0 a3

natural matrix interpretation 1    natural matrix interpretation 2
    [(b0,x -> [[1]] * x +  [[0]])      [(b0,x -> [[1]] * x +  [[1]])
    ,(b2,x -> [[3]] * x +  [[1]])      ,(b2,x -> [[5]] * x +  [[9]])
    ,(b1,x -> [[1]] * x +  [[4]])      ,(b1,x -> [[1]] * x +  [[0]])
    ,(b3,x -> [[1]] * x +  [[1]])      ,(b3,x -> [[1]] * x +  [[4]])
    ,(a0,x -> [[1]] * x +  [[0]])      ,(a0,x -> [[1]] * x +  [[0]])
    ,(a2,x -> [[1]] * x +  [[1]])      ,(a2,x -> [[3]] * x +  [[0]])
    ,(a1,x -> [[3]] * x +  [[7]])      ,(a1,x -> [[1]] * x +  [[0]])
    ,(a3,x -> [[1]] * x +  [[0]])]     ,(a3,x -> [[1]] * x +  [[1]])]
```

▶ **Example 3.** We can mimick Jambox' behaviour by restricting the number of interpretations for the labelled system to 1. With a model of size 2^3, we get a termination proof (details omitted)

```
matchbox  ~/tpdb/tpdb-4.0/SRS/Zantema/z001.srs  -l3,1 -d1 -b3 --nat
   CNF finished (#variables: 23876, #clauses: 91447)
   Solver finished in 2.282 seconds (result: True)
```

Jambox' formula (for the same parameters) has 33492 variables and 232683 clauses. That means that our CO4 compiler produced a SAT encoding that is comparable to a manual encoding.

6 Extensions

Our implementation additionally allows for each removal step for the labelled system to use arctic or natural matrix interpretation, or lexicographic path order with argument filtering, and also to reverse rules or not—where all these options are encoded in the program, and thus chosen by the solver.

▶ **Example 4.** This gives one of the shorter termination proofs for Zantema's system:

```
matchbox  ~/tpdb/tpdb-4.0/SRS/Zantema/z001.srs  -l2,1 -d1 -b1 --lpo
    CNF finished (#variables: 4752, #clauses: 17075)
    Solver finished in 1.811 seconds (result: True)
model:    2b0 3b1 3b2 0b3
          1a0 2a1 1a2 1a3
labelled system: a1 a3 b2 b0 -> b0 b3 b1 a2 a1 a0
                 a1 a0 b3 b2 -> b0 b3 b1 a2 a1 a2
                 a1 a0 b3 b1 -> b0 b3 b2 a1 a2 a1
                 a1 a2 b0 b3 -> b0 b3 b1 a2 a1 a3
LPO:  delete symbols: a1 a3
      precedence: a0 = b2 > a2 = b3 > b0 > b1
```

Here we compare (with respect to the path order) after applying the morphism (argument filter) that deletes some symbols.

References

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

2 Jörg Endrullis. Jambox, 2009. Available at `http://joerg.endrullis.de`.

3 Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

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

5 Harald Zankl, Christian Sternagel, Dieter Hofbauer, and Aart Middeldorp. Finding and certifying loops. In Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorný, and Bernhard Rumpe, editors, *SOFSEM*, volume 5901 of *Lecture Notes in Computer Science*, pages 755–766. Springer, 2010.

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

Analyzing Runtime and Size Complexity of Integer Programs (Abstract)

Marc Brockschmidt[1], Fabian Emmes[1], Stephan Falke[2], Carsten Fuhs[3], and Jürgen Giesl[1]

1 **RWTH Aachen University, Germany**
2 **Karlsruhe Institute of Technology, Germany**
3 **University College London, UK**

Abstract

We developed a modular approach to automatic complexity analysis. Based on a novel alternation between finding symbolic time bounds for program parts and using these to infer size bounds on program variables, we can restrict each analysis step to a small program part while maintaining a high level of precision. Extensive experiments with the implementation of our method demonstrate its performance and power in comparison with other tools. In particular, our method finds bounds for many programs whose complexity could not be analyzed by automatic tools before.

1998 ACM Subject Classification D.2.4 Software/Program Verification, D.2.8 Metrics, F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Symbolic Complexity Bounds, Termination Analysis, Runtime Complexity, Size Complexity, Integer Programs

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

Methods for automatically proving termination of imperative programs have received increased attention recently. But in many cases, termination is not sufficient. Instead, the program should terminate in reasonable (typically, (pseudo-)polynomial) time. We build upon the well-known observation that (e.g., polynomial) rank functions used for termination proofs implicitly also provide a runtime complexity bound. However, this only holds for proofs using a *single* rank function. In practice, larger programs are usually handled by a disjunctive or lexicographic combination of simple rank functions. Deriving a complexity bound in this setting is much harder, as the two examples below illustrate.

For both programs, the lexicographic rank function $\langle f_1, f_2 \rangle$ proves termination, where f_1 measures states by $\mathtt{i}$ and f_2 is $\mathtt{x}$. However, the program on the left has linear runtime, while the program on the right has quadratic complexity. The difference between the programs is in the *size* of $\mathtt{x}$ after the first loop. To handle such effects, our method derives *runtime complexity* bounds for parts of the program and uses them to deduce *size complexity* bounds for program variables at certain locations. We measure the *size* of integers by their absolute values. These size bounds allow to derive more runtime complexity bounds, and the process continues until all loops and variables have been handled.

while $\mathtt{i} > 0$ **do**	**while** $\mathtt{i} > 0$ **do**
$\mathtt{i} = \mathtt{i} - 1$	$\mathtt{i} = \mathtt{i} - 1$
	$\mathtt{x} = \mathtt{x} + \mathtt{i}$
done	**done**
while $\mathtt{x} > 0$ **do**	**while** $\mathtt{x} > 0$ **do**
$\mathtt{x} = \mathtt{x} - 1$	$\mathtt{x} = \mathtt{x} - 1$
done	**done**

For the example on the right, our method first proves that the first loop is executed linearly often using the rank function $\mathtt{i}$. Then, it deduces that $\mathtt{i}$ is bounded by its initial value $\mathtt{i}_0$ in all loop iterations. Combining these observations, our approach infers that $\mathtt{x}$ is incremented by a value bounded by $\mathtt{i}_0$ at most $\mathtt{i}_0$ times, yielding that $\mathtt{x}$ is bounded by $\mathtt{x}_0 + \mathtt{i}_0^2$. Finally, our method detects that the second loop is executed $\mathtt{x}$ times, and combines this with our bound $\mathtt{x}_0 + \mathtt{i}_0^2$ on the value of $\mathtt{x}$ when entering the second loop. This allows us to conclude that the program's runtime is bounded by $\mathtt{i}_0 + \mathtt{i}_0^2 + \mathtt{x}_0$.

Our main contribution is a novel approach which *alternates* between finding *runtime bounds* and finding *size bounds* for sequential imperative programs operating on integer data with (potentially non-linear) arithmetic and (unbounded) non-determinism. We apply this general approach to obtain two main results:

1. A novel method to deduce (often non-linear) *size bounds* on program variables by combining bounds for local variable changes with runtime bounds.
2. A new *modular* method to compute symbolic *runtime bounds* for isolated program parts. These runtime bounds are based on size bounds for variables that may have been modified in the preceding parts of the program. In this way, we only need to consider small program parts at a time, allowing our approach to *scale* to larger programs.

Several methods to determine symbolic runtime complexity bounds have been developed in recent years, e.g., [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. To the best of our knowledge, our approach is the first that implements a combined, alternating size and runtime analysis for imperative programs. To evaluate our method, we have created a simple prototype `KoAT`.

We compared this prototype with the complexity analyzers `PUBS` [1, 2] and `Rank` [3] on 682 examples from the literature on termination and complexity analysis of integer programs. The table on the side illustrates how often the tools could infer a runtime bound for the example set. Here 1, $\log n$, n, $n \log n$, n^2, n^3, and $n^{>3}$ represent their corresponding asymptotic classes and EXP is the class of exponential functions. The column "Time" gives the average runtime on those examples where the respective tool was successful. The table shows that on this collection, our approach was substantially more powerful than the two other previous tools and still efficient.

	1	$\log n$	n	$n \log n$	n^2	n^3	$n^{>3}$	EXP	Time
`KoAT`	102	0	151	0	58	3	3	0	1.7 s
`PUBS`	85	4	104	1	13	4	0	6	.3 s
`Rank`	56	0	19	0	8	1	0	0	.5 s

References

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

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

3 C. Alias, A. Darte, P. Feautrier, L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS '10*, pages 117–133, 2010.

4 M. Avanzini and G. Moser. A combination framework for complexity. In *RTA '13*, pages 55–70, 2013.

5 R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács. `ABC`: Algebraic bound computation for loops. In *LPAR-16*, pages 103–118, 2010.

6 J. Giesl, T. Ströder, P. Schneider-Kamp, F. Emmes, and C. Fuhs. Symbolic evaluation graphs and term rewriting: A general methodology for analyzing logic programs. In *PPDP '12*, pages 1–12, 2012.

7 S. Gulwani, K. K. Mehra, and T. M. Chilimbi. `SPEED`: Precise and efficient static estimation of program computational complexity. In *POPL '09*, pages 127–139, 2009.

8 J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *TOPLAS*, 34(3), 2012.

9 J. A. Navas, E. Mera, P. López-García, and M. V. Hermenegildo. User-definable resource bounds analysis for logic programs. In *ICLP '07*, pages 348–363, 2007.

10 L. Noschinski, F. Emmes, and J. Giesl. Analyzing innermost runtime complexity of term rewriting by dependency pairs. *JAR*, 51(1):27–56, 2013.

11 F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS '11*, pages 280–297, 2011.

Cooperation For Better Termination Proving

Marc Brockschmidt[1], Byron Cook[2], and Carsten Fuhs[3]

1 **RWTH Aachen University, Germany**
`brockschmidt@cs.rwth-aachen.de`
2 **Microsoft Research and University College London, United Kingdom**
`bycook@microsoft.com`
3 **University College London, United Kingdom**
`c.fuhs@cs.ucl.ac.uk`

Abstract

One of the difficulties of proving program termination is managing the subtle interplay between the finding of a termination argument and the finding of the argument's supporting invariant. In this extended abstract we propose a new mechanism that facilitates better cooperation between these two types of reasoning. In an experimental evaluation we find that our new method leads to dramatic performance improvements.

Keywords and phrases Termination analysis, safety proving, rank functions

1 Introduction

When proving program termination we are simultaneously solving two problems: the search for a termination argument, and the search for a supporting invariant. Consider the example to the right. To prove termination of this program we are looking to find both a termination argument (*i.e.*, "x decreases until 0") and a supporting invariant (*i.e.*, $y > 0$). The two are interrelated: Without $y > 0$, we cannot prove the validity of the argument "x decreases until 0"; and without "x decreases towards 0", how would we know that we need to prove $y > 0$?

```
y := 1;
while x > 0 do
    x := x − y;
    y := y + 1;
done
```

Several program termination proving tools (*e.g.* [9], [10], [11], [15], [17]) address this problem using a strategy that oscillates between calls to an off-the-shelf safety prover (*e.g.* [1], [3], [8], [12], [14], *etc.*) and calls to a rank function synthesis tool (*e.g.* [2], [5], [6], [16], *etc.*). In this setting a candidate termination argument is iteratively constructed. The safety prover proves or disproves the validity of the current argument via the search for invariants. Refinement of the current termination argument is performed using the output of a rank function synthesis tool when applied to counterexamples found by the safety prover.

A difficulty with this approach is that currently, the underlying tools do not share enough information about the overall state of the termination proof. For example, the rank function synthesis tool is only applied to the single path through the program described by the counterexample found by the safety prover, while the context of this single path is not considered at all. Meanwhile, the safety prover is unaware of things such as which paths in the program have already been deemed terminating and how those paths might contribute to other potentially infinite executions. The result is lost performance, as the underlying tools often make choices inappropriate to the common goal of fast termination proving.

Here we introduce a technique that facilitates cooperation between the underlying tools in a termination prover, thus allowing for decisions more appropriate to the common good of proving program termination. The idea is to use a single representation of the state of the termination proof search—called a *cooperation graph*—that both tools operate over. Nodes in the graph are marked as either termination-nodes or safety-nodes to indicate their role

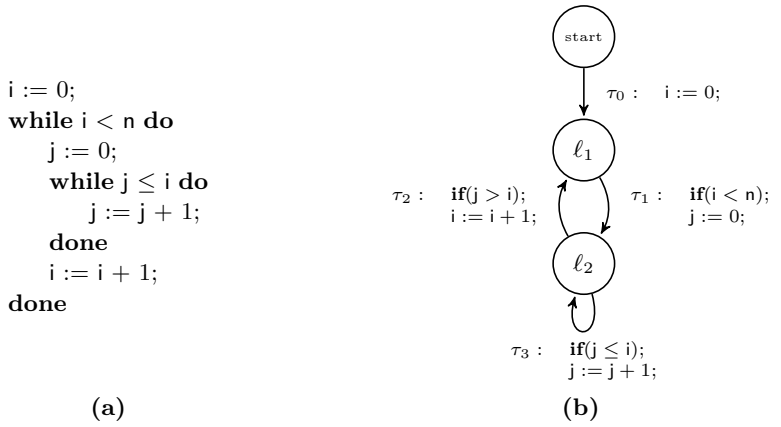


Figure 1 Textual and control-flow graph representation of skeleton bubble sort routine

in the state of the proof. With this additional information exposed, we can now represent the progress of the termination proof search by modifying the termination subgraph. This has practical advantages: the safety prover can be encouraged not to explore parts of the program that have already been proven terminating, and the rank function synthesis can make use of the full program structure in order to find better termination arguments.

Our approach results in significant performance improvements over earlier methods and our implementation succeeds on numerous programs where previous tools fail. In cases where previous tools do succeed, our implementation boosts performance by orders of magnitude.

Limitations. While in theory our approach works in a general setting, our implementation focuses on sequential arithmetic programs (so these programs do not use the heap or bitvectors). In some cases we have soundly abstracted C programs with heap to arithmetic programs (*e.g.* using a technique due to Magill *et al.* [13]); in other cases, as is standard in many tools (*e.g.* SLAM [3]), we essentially ignored bitvectors and the heap.

The full version of the present short paper has been published in [7].

2 Illustrating Example

We illustrate our approach using the example in Fig. 1, which displays a bubble-sort like program (the manipulation of the data has been abstracted away). In our setting we use a graph—called a *cooperation graph*—to facilitate sharing of information between a safety prover and a rank function synthesis procedure. See Fig. 2 for the cooperation graph at the start of the proof search. We start with the control-flow graph of the original program from Fig. 1, which we keep for reasoning about safety (*i.e.*, (un)reachability from the initial program states). Intuitively, this part of the graph is for the finite prefix of a run from the initial location to a loop with a (hypothetical) infinite suffix of the run. For this infinite suffix, we have duplicated the loops of the original program (in the form of the *strongly-connected components (SCCs)* of the graph with locations ℓ_1^t and ℓ_2^t). We connect the two parts of the graph with non-deterministic transitions from one copy of the program to the other (*i.e.*, τ_4 and τ_5). Technically, the cooperation graph contains a superset of the transitions in the initial program, yet if we can prove that there is no infinite run from the initial location where ℓ_1^t or ℓ_2^t occur infinitely often, this implies termination of the original program as well.

After duplication, we also apply a few known tricks: In the new copy of the program, we follow the approach of Biere *et al.* [4] by adding nodes (*i.e.*, ℓ_1^d and ℓ_2^d) and transitions to take

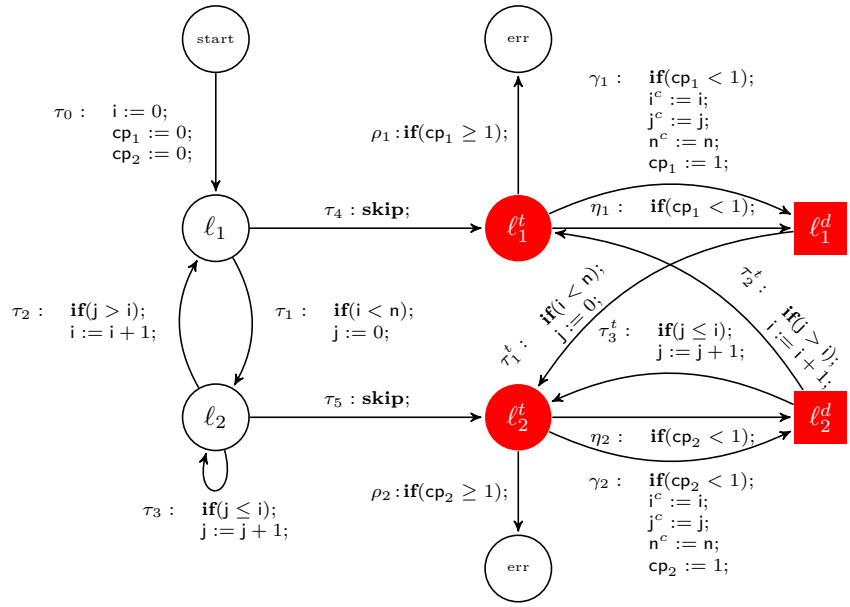


Figure 2 Cooperation graph derived from Fig. 1

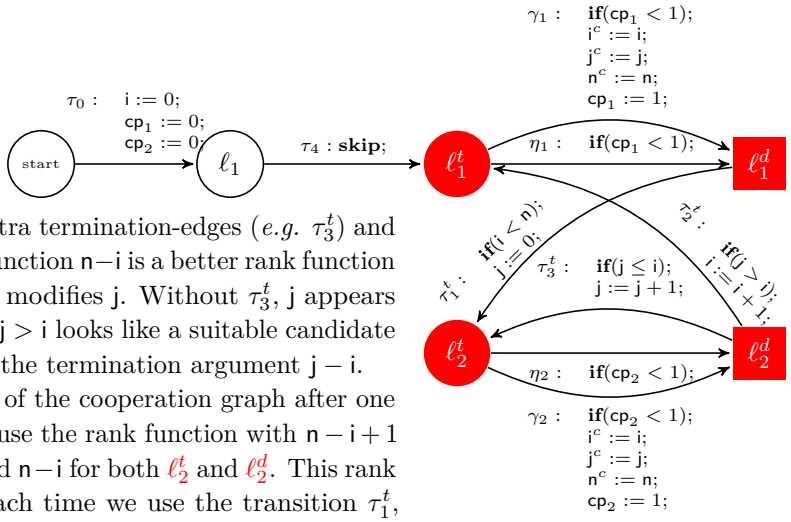
a snapshot of variable values (*i.e.*, γ_1 and γ_2). The current values of variables $\mathsf{i}, \mathsf{j}, \mathsf{n}$ are stored in copies $\mathsf{i}^c, \mathsf{j}^c, \mathsf{n}^c$ and the flag cp_k is set to indicate that a snapshot was taken at location ℓ_k. Furthermore, new transitions to an error location "err" have been added that can be strengthened later by partial termination arguments *à la* Cook *et al.* [9]. Proving this error location unreachable then implies a termination proof for the input program. In the resulting graph, reasoning about termination is performed on the right-hand side (the *termination subgraph*) by a procedure built around an efficient rank function synthesis. We search for supporting invariants on the left-hand side (the *safety subgraph*) via a safety prover.

Via this duplication to the termination and safety subgraphs, we can easily restrict certain operations to either subgraph, yet still maintain a connection between them. The safety subgraph describes an over-approximation of all reachable states, while the termination subgraph is an over-approximation of those states whose termination has not been proven yet. This allows us to perform operations in the one half that may not make sense (or may be unsound) in the other: when we prove that transitions in the termination subgraph can only be used finitely often, we can simply remove them, as they cannot contribute to infinite executions. This is only sound because the safety subgraph remains unchanged in this simplification, which keeps the set of reachable states unchanged and hence allows reasoning about safety/invariants. These iterative program simplifications encode the progress of the termination proof search and are directly available to the safety prover.

The graph structure guides the safety prover to unproven parts of the program yielding relevant counterexamples and allowing the rank function synthesis to produce better termination arguments. If these do not allow a program simplification, they still guide the invariant generation by the safety prover for nodes in the safety subgraph. The invariants in turn support reasoning about the validity of termination arguments in the termination subgraph.

Termination proof sketch. In our example, we begin searching for a path from the "start" location to the error location "err". We might, for example, choose the path $\langle \tau_0, \tau_4, \gamma_1, \tau_1^t, \eta_2, \tau_3^t, \eta_2, \tau_2^t, \rho_1 \rangle$ where τ_0 is drawn from the safety subgraph and the other transitions come from the termination subgraph. Here, $\langle \gamma_1, \tau_1^t, \eta_2, \tau_3^t, \eta_2, \tau_2^t \rangle$ form a cy-

cle in the execution, returning back to location ℓ_1^t. In our approach we do not simply use this command sequence directly to search for a new termination argument (as is done in previous tools). Instead, we additionally consider all transitions from the termination subgraph that enter and exit nodes in the strongly connected component containing the found cycle of termination-transitions in the counterexample. In this case, because the graph is so small, this includes the entire termination subgraph:



This graph has extra termination-edges (*e.g.* τ_3^t) and shows that the rank function $\mathsf{n}-\mathsf{i}$ is a better rank function than $\mathsf{j}-\mathsf{i}$ because τ_3^t modifies j. Without τ_3^t, j appears as a constant so that $\mathsf{j} > \mathsf{i}$ looks like a suitable candidate invariant supporting the termination argument $\mathsf{j}-\mathsf{i}$.

Fig. 3 is the state of the cooperation graph after one counterexample. We use the rank function with $\mathsf{n}-\mathsf{i}+1$ for both ℓ_1^t and ℓ_1^d, and $\mathsf{n}-\mathsf{i}$ for both ℓ_2^t and ℓ_2^d. This rank function decreases each time we use the transition τ_1^t, and the condition $\mathsf{i} < \mathsf{n}$ implies that the rank function is bounded from below. So τ_1^t can only be used finitely often and we can remove it from the termination subgraph. This also allows to remove ℓ_1^t, ℓ_1^d and all transitions connected to the two, as they are not on a non-trivial SCC anymore and thus cannot occur infinitely often in an execution. Removing the corresponding node ℓ_1 from the safety subgraph is unsound, as this would make the inner loop unreachable, without any termination proof for it.

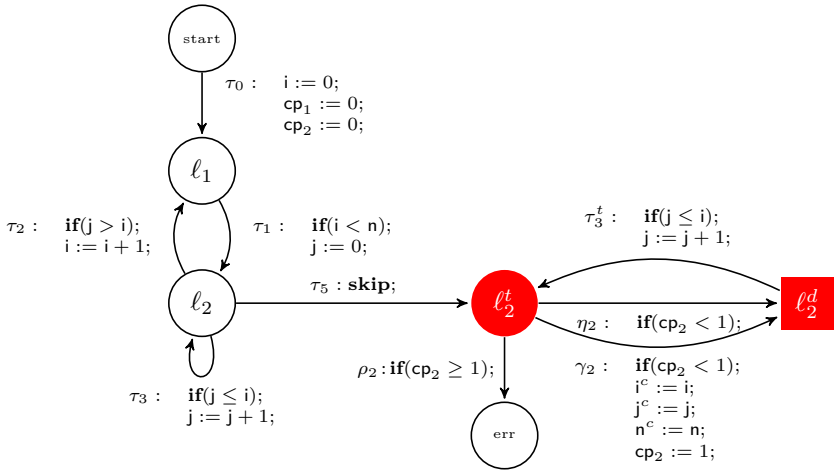


Figure 3 Cooperation graph after safety and termination analysis on the graph from Fig. 2. Due to termination analysis, the transition τ_2^t has been removed. Afterward, ℓ_1^t was not part of a non-trivial SCC anymore, so it, its duplicate ℓ_1^d, and the connecting transitions were removed.

In the next iteration, starting on Fig. 3, all possible cycles in the termination subgraph use the transition τ_3^t. We prove this transition well founded via the rank function $\mathsf{i}-\mathsf{j}$ for the locations ℓ_2^t and ℓ_2^d, allowing us to remove τ_3^t and then, ℓ_2^t, ℓ_2^d and all connected transitions. This yields a cooperation graph with an empty termination subgraph (so we are left with

what is essentially the original graph from Fig. 1). Thus we have proved termination.

3 Conclusion

One of the difficulties for reliable and scalable program termination provers is orchestrating the interplay between the reasoning about progress and the search for supporting invariants. We have developed a new method that facilitates cooperation between these two types of reasoning. Our representation gives the underlying tools the whole picture of the current proof state, allowing both types of reasoning to contribute towards the greater goal and also to share their intermediate findings. Our experiments (which we cannot present here for space reasons; details on experiments and benchmarks are available at `http://verify.rwth-aachen.de/brockschmidt/Cooperating-T2/` and in [7]) indicate dramatic performance gains.

The full version of this short paper has been published at [7], and our implementation in T2 is available for download at `http://research.microsoft.com/en-us/projects/t2/`.

References

1 Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Whale: an interpolation-based algorithm for inter-procedural verification. In *Proc. VMCAI '12*.

2 Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Proc. SAS '10*.

3 Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *Proc. CAV '01*.

4 Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. In *Proc. FMICS '02*.

5 Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In *Proc. CAV '05*.

6 Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. The polyranking principle. In *Proc. ICALP '05*.

7 Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In *Proc. CAV '13*.

8 Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Proc. TACAS '05*.

9 Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proc. PLDI '06*.

10 Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. lexicographic termination proving. In *Proc. TACAS '13*.

11 Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *Proc. PLDI '12*.

12 Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *Proc. SPIN '03*.

13 Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proc. POPL '10*.

14 Ken McMillan. Lazy abstraction with interpolants. In *Proc. CAV '06*.

15 Andreas Podelski and Andrey Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *Proc. PADL '07*.

16 Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Proc. VMCAI '04*.

17 Aliaksei Tsitovich, Natasha Sharygina, Christoph M. Wintersteiger, and Daniel Kroening. Loop summarization and termination analysis. In *Proc. TACAS '11*.

Towards decidable classes of logic programs with function symbols

Marco Calautti, Sergio Greco, Cristian Molinaro, Irina Trubitsyna

DIMES, Università della Calabria
87036 Rende (CS), Italy
{calautti,greco,cmolinaro,trubitsyna}@dimes.unical.it

Abstract

Function symbols are widely acknowledged as an important feature in logic programming, but unfortunately, common inference tasks become undecidable in their presence. To cope with this issue, recent research has focused on identifying decidable classes of programs allowing only a restricted use of function symbols while ensuring decidability of common inference tasks. In this paper, we give an overview of current *termination criteria*. We also present a technique which can be used in conjunction with current termination criteria to enlarge the class of programs recognized as terminating.

1998 ACM Subject Classification F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

Keywords and phrases Logic programming with function symbols, bottom-up evaluation, stable model semantics

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

1 Introduction and Preliminaries

In recent years, there has been a great deal of interest in enhancing answer set solvers by supporting function symbols. Function symbols often make modeling easier and the resulting encodings more readable and concise, but unfortunately, common inference tasks become undecidable in their presence. The class of *finitely ground programs*, proposed in [1], guarantees decidability of common inference tasks. In particular, finitely ground programs are terminating, i.e. the bottom-up evaluation of these programs gives a finite number of finite stable models. Since membership in the class is semi-decidable, research has focused on identifying sufficient conditions for a program to be finitely ground, leading to different criteria, called *termination criteria*. Efforts in this direction are *ω-restricted programs* [9], *λ-restricted programs* [2], *finite domain programs* [1], *argument-restricted programs* [8], *safe programs* [7], *Γ-acyclic programs* [7], and *bounded programs* [5]. In this paper, we give an overview of recent research on this topic. Specifically, we present some recently proposed decidable termination criteria, able to recognize the termination of disjunctive logic programs, and an orthogonal technique that can be used in conjunction with them to enlarge the class of programs recognized as finitely-ground [6].

We assume the reader is familiar with logic programs with function symbols under the *stable model semantics* [3] (see [5] for a brief overview). Given a program $\mathcal{P}$ we denote by $arg(\mathcal{P})$ the set of all arguments of $\mathcal{P}$, i.e., expressions of the form $p[i]$ where p is a predicate symbol of arity n appearing in $\mathcal{P}$ and $1 \leq i \leq n$. We use $body(r)$ and $head(r)$ to denote the body and the head of a rule r in $\mathcal{P}$; $body^+(r)$ denotes the conjunction of all positive literals in $body(r)$. For any rule r, $ground(r)$ denotes the set of rules obtained by replacing

variables with ground terms which can be constructed using constants and function symbols occurring in $\mathcal{P}$. An argument $q[i] \in arg(\mathcal{P})$ is said to be *limited* if it takes values from a finite domain, that is, if for every (stable) model M of $\mathcal{P}$ the projection of Q over the i-th arguments is a finite set, where Q is the set of q-atoms in M. We consider programs where rules are range restricted, that is all variables occurring in a rule r also occur in $body^+(r)$ and distinguish *base predicate symbols*, defined only by facts (i.e., ground rules with empty body) from *derived predicate symbols*, defined by arbitrary rules. For ease of presentation, we sometimes consider only positive programs as the techniques described can be easily extended to programs with negative body literals and disjunction in head.

2 Basic Termination Criteria

In this section, we describe the most general "basic" termination criterion proposed in the literature, namely *argument-restricted programs* [8]. We shall not discuss other well-known basic termination criteria, such as *ω-restricted programs* [9], *λ-restricted programs* [2] and *finite domain programs* [1], as they have been generalized by argument-restricted programs. We named the aforementioned termination criteria "basic" as their definition does not rely on other termination criteria.

Termination criteria are used to determine sets of arguments which are limited. In the following we shall use the following notations. Given a program $\mathcal{P}$ and a criterion W, $W(\mathcal{P})$ denotes the set of arguments which are recognized as limited by criterion W, whereas $\mathcal{W}$ denotes the class of programs which are recognized as terminating by W, that is the class of programs such that $arg(\mathcal{P}) = W(\mathcal{P})$.

Argument-restricted programs [8]. The argument-restricted criterion tests the possibility to find for each argument a *finite* upper bound of the depth of terms that may occur in that argument during the program evaluation. This test is based on the notion of *argument ranking function* defined below. For any atom A of the form $p(t_1, ..., t_n)$, A^0 denotes the predicate symbol p, and A^i denotes term t_i, for $1 \leq i \leq n$.

▶ **Definition 1.** An *argument ranking* for a program $\mathcal{P}$ is a partial function ϕ from $arg(\mathcal{P})$ to non-negative integers such that, for every rule r of $\mathcal{P}$, every atom A occurring in the head of r, and every variable X occurring in an argument term A^i, if $\phi(A^0[i])$ is defined, then $body^+(r)$ contains an atom B such that X occurs in an argument term B^j, $\phi(B^0[j])$ is defined, and the following condition is satisfied

$$\phi(A^0[i]) - \phi(B^0[j]) \geq d(X, A^i) - d(X, B^j)$$

where $d(X,t) = 0$ if $t = X$; if $t = f(v_1, \ldots, v_k)$, then $d(X,t) = 1 + \max_{v_l \text{ contains } X} d(X, v_l)$.
The set of *restricted arguments* of $\mathcal{P}$ is $AR(\mathcal{P}) = \{p[i] \mid p[i] \in arg(\mathcal{P}) \wedge \exists \phi \text{ s.t. } \phi(p[i]) \text{ is defined}\}$.
A program $\mathcal{P}$ is said to be *argument restricted* iff $AR(\mathcal{P}) = arg(\mathcal{P})$. □

▶ **Example 2.** Consider the following logic program P_2:

$\mathtt{p(f(X))} \leftarrow \mathtt{q(X)}.$
$\mathtt{q(X)} \leftarrow \mathtt{p(f(X))}.$

The program is recognized to be argument-restricted. In particular, the argument-restricted function ϕ can be defined as follows: $\phi(\mathtt{p[1]}) = 1$ and $\phi(\mathtt{q[1]}) = 0$. □

3 Iterated Termination Criteria

In this section we present recently proposed criteria which, starting from a set of limited arguments defined through the application of a basic criterion, computes a possibly larger set of limited arguments.

Safe programs [7]. The first technique is obtained by introducing a fixpoint function, called *safe function*, which, iteratively, extends a given set of limited arguments. Its definition is based on the notion of activation graph.

The *activation graph* of a program $\mathcal{P}$, denoted $\Omega(\mathcal{P})$, is a directed graph whose nodes are the rules of $\mathcal{P}$, and there is an edge (r_i, r_j) in the graph iff r_i activates r_j, i.e. there exist two ground rules $r'_i \in ground(r_i)$, $r'_j \in ground(r_j)$ and a set of ground atoms D such that *(i)* $D \not\models r'_i$, *(ii)* $D \models r'_j$, and *(iii)* $D \cup head(r'_i) \not\models r'_j$. This intuitively means that if D does not satisfy r'_i, D satisfies r'_j, and $head(r'_i)$ is added to D to satisfy r'_i, this causes r'_j not to be satisfied anymore (and then to be "activated").

▶ **Definition 3.** Given a program $\mathcal{P}$ and a basic termination criterion W, the set of *W-safe arguments* $S\text{-}W(\mathcal{P})$ is computed by first setting $S\text{-}W(\mathcal{P}) = W(\mathcal{P})$ and next iteratively adding each argument $q[k]$ such that for all rules $r \in \mathcal{P}$ where q appears in the head (i) r does not depend on a cycle of $\Omega(\mathcal{P})$, or (ii) for every head atom $q(t_1, ..., t_n)$, every variable X appearing in t_k appears also in some safe argument in $body^+(r)$. A program $\mathcal{P}$ is said to be *W-safe* if $S\text{-}W(\mathcal{P}) = arg(\mathcal{P})$. □

The criterion obtained by combining basic criterion W with the safe function is denoted by $S\text{-}W$.

▶ **Example 4.** The following simple logic program $\mathcal{P}_4$ is not recognized as terminating by any basic termination criteria introduced so far.

$$\mathtt{p(X,X)} \leftarrow \mathtt{base(X)}.$$
$$\mathtt{p(f(X),g(X))} \leftarrow \mathtt{p(X,X)}.$$

However, this program is terminating and $\mathcal{P}_4 \in \mathcal{S}\text{-}\mathcal{W}$, for every basic criterion W, since the activation graph of $\mathcal{P}_4$ does not contain any cycle. □

Bounded Programs [5]. The definition of bounded programs relies on the notion of *labelled argument graph*. This graph, denoted $\mathcal{G}_L(\mathcal{P})$, is derived from the argument graph by labelling edges as follows: for each pair of nodes $p[i], q[j] \in arg(\mathcal{P})$ and for every rule $r \in \mathcal{P}$ such that *(i)* an atom $p(t_1, ..., t_n)$ appears in $head(r)$, *(ii)* an atom $q(u_1, ..., u_m)$ appears in $body^+(r)$, *(iii)* terms t_i and u_j have a common variable X, there is an edge $(q[j], p[i], \langle \alpha, r, h, k \rangle)$, where h and k are natural numbers denoting the positions of $p(t_1, ..., t_n)$ in $head(r)$ and $q(u_1, ..., u_m)$ in $body^+(r)$, respectively[1], whereas $\alpha = \epsilon$ if $t_i = u_j$, $\alpha = f$ if $u_j = X$ and $t_i = f(..., X, ...)$, $\alpha = \bar{f}$ if $u_j = f(..., X, ...)$ and $t_i = X$. For the sake of simplicity, without loss of generality, we assume that if a variable X appears in two terms occurring in the head and body of a rule respectively, then only one of the two terms is a complex term and that the nesting level of complex terms is at most one.

[1] We assume that literals in the head (resp. body) are ordered with the first one being associated with 1, the second one with 2, etc.

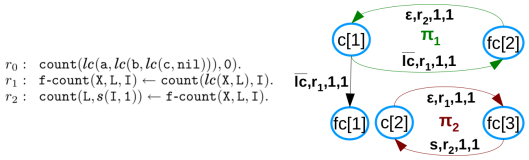


Figure 1 Rewriting of $\mathcal{P}6$ and corresponding labelled argument graph.

Given a path $\rho = (a_1, b_1, \langle \alpha_1, r_1, h_1, k_1 \rangle), \ldots, (a_m, b_m, \langle \alpha_m, r_m, h_m, k_m \rangle)$, we define $\lambda_1(\rho) = \alpha_1 \ldots \alpha_m$, $\lambda_2(\rho) = r_1, ..., r_m$, and $\lambda_3(\rho) = \langle r_1, h_1, k_1 \rangle \ldots \langle r_m, h_m, k_m \rangle$. Given a cycle π consisting of n labelled edges $e_1, ..., e_n$, we can derive n different cyclic paths starting from each of the e_i's—we use $\tau(\pi)$ to denote the set of such cyclic paths.

Given two cycles π_1 and π_2, we write $\pi_1 \approx \pi_2$ iff $\exists \rho_1 \in \tau(\pi_1)$ and $\exists \rho_2 \in \tau(\pi_2)$ such that $\lambda_3(\rho_1) = \lambda_3(\rho_2)$. Given a program $\mathcal{P}$, we say that a cycle π in $\mathcal{G}_L(\mathcal{P})$ is *active* iff $\exists \rho \in \tau(\pi)$ such that $\lambda_2(\rho) = r_1, ..., r_m$ and $(r_1, r_2), ..., (r_{m-1}, r_m), (r_m, r_1)$ is a cyclic path in the activation graph $\Omega(\mathcal{P})$.

Given a program $\mathcal{P}$ and a path ρ in $\mathcal{G}_L(\mathcal{P})$, we denote with $\hat{\lambda}_1(\rho)$ the string obtained from $\lambda_1(\rho)$ by iteratively eliminating pairs of the form $\gamma\bar{\gamma}$ from the string until the resulting string cannot be further reduced.

Given a program $\mathcal{P}$, a cycle π in $\mathcal{G}_L(\mathcal{P})$ can be classified as follows. We say that π is i) *balanced* if $\exists \rho \in \tau(\pi)$ s.t. $\hat{\lambda}_1(\rho)$ is empty, ii) *growing* if $\exists \rho \in \tau(\pi)$ s.t. $\hat{\lambda}_1(\rho)$ does not contain a symbol of the form $\bar{\gamma}$, iii) *failing* otherwise.

▶ **Definition 5.** *Given a program $\mathcal{P}$ and a basic termination criterion W, the set of* W-bounded arguments *B-$W(\mathcal{P})$ is computed by first setting B-$W(\mathcal{P}) = W(\mathcal{P})$ and next iteratively adding each argument $q[k]$ such that for each basic cycle π in $\mathcal{G}_L(\mathcal{P})$ on which $q[k]$ depends, at least one of the following conditions holds:*

1. *π is not active or is not growing;*
2. *π contains an edge $(s[j], p[i], \langle f, r, l_1, l_2 \rangle)$ and, letting $p(t_1, ..., t_n)$ be the l_1-th atom in the head of r, for every variable X in t_i, there is an atom $b(u_1, ..., u_m)$ in $body^+(r)$ s.t. X appears in a term u_h and $b[h]$ is W-bounded;*
3. *there is a basic cycle π' in $\mathcal{G}_L(\mathcal{P})$ s.t. $\pi' \approx \pi$, π' is not balanced, and π' passes only through W-bounded arguments.*

A program $\mathcal{P}$ is said to be W-bounded *if B-$W(\mathcal{P}) = arg(\mathcal{P})$.* □

The criterion obtained by combining basic criterion W with the bounded function is denoted by B-W. The class of W-bounded programs is denoted by $\mathcal{BW}$. A relevant aspect that distinguishes this technique from other works is that this technique analyzes how groups of arguments are each other related—this is illustrated in the following example.

▶ **Example 6.** Consider the following logic program $\mathcal{P}_6$:

r_0 : count([a, b, c], 0).
r_1 : count(L, I + 1) ← count([X|L], I).

The bottom-up evaluation of $\mathcal{P}_6$ terminates yielding the set of atoms count([a, b, c], 0), count([b, c], 1), count([c], 2), and count([ ], 3). The query goal count([ ], L) can be used to retrieve the length L of list [a, b, c].[2] □

[2] Notice that $\mathcal{P}_6$ has been written so as to count the number of elements in a list when evaluated in a

To comply with the syntactic restrictions required by the bounded technique, Figure 1 shows a rewriting of $\mathcal{P}_6$ and the corresponding labelled argument graph. where lc and s denote the list constructor and the sum operators respectively. Basically, considering the argument-restricted technique as the basic criterion W, after having established that argument $\texttt{count}[1]$ is limited, that is $\texttt{count}[1] \in B\text{-}AR(\mathcal{P}_6)$, by analyzing the two cycles involving arguments $\texttt{count}[1]$ and $\texttt{count}[2]$, respectively and using Condition 3 of Definition 5 it is possible to detect that also argument $\texttt{count}[2]$ is limited, that is $\texttt{count}[2] \in B\text{-}AR(\mathcal{P}_6)$. Consequently, $\mathcal{P}_6$ is AR-bounded.

4 Rewriting technique

In this section we present a rewriting technique [6] that, used in conjunction with current termination criteria, allows us to detect more programs as finitely-ground. This technique takes a logic program $\mathcal{P}$ and transforms it into an adorned program $\mathcal{P}^\mu$ with the aim of applying termination criteria to $\mathcal{P}^\mu$ rather than $\mathcal{P}$. The transformation is sound in that if the adorned program satisfies a certain termination criterion, then the original program satisfies this criterion as well and, consequently, is finitely-ground. Importantly, as showed by the below example, applying termination criteria to adorned programs rather than the original ones strictly enlarges the class of programs recognized as finitely-ground. This technique is much more general than those used to deal with chase termination (see [4]).

▶ **Example 7.** Consider the following program $\mathcal{P}_7$, where $\texttt{base}$ is a base predicate symbol defined by facts not showed here.

$$r_0: \quad \texttt{p}(\texttt{X}, \texttt{f}(\texttt{X})) \leftarrow \texttt{base}(\texttt{X}).$$
$$r_1: \quad \texttt{p}(\texttt{X}, \texttt{f}(\texttt{X})) \leftarrow \texttt{p}(\texttt{Y}, \texttt{X}),\ \texttt{base}(\texttt{Y}).$$
$$r_2: \quad \texttt{p}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{p}(\texttt{f}(\texttt{X}), \texttt{f}(\texttt{Y})).$$

First, base predicate symbols are adorned with strings of ϵ's; thus, we get the adorned predicate symbol $\texttt{base}^\epsilon$. This is used to adorn the body of r_0 so as to get

$$\rho_0: \quad \texttt{p}^{\epsilon\texttt{f}_1}(\texttt{X}, \texttt{f}(\texttt{X})) \leftarrow \texttt{base}^\epsilon(\texttt{X}).$$

from which we derive the new adorned predicate symbol $\texttt{p}^{\epsilon\texttt{f}_1}$, and the adornment definition $\texttt{f}_1 = \texttt{f}(\epsilon)$. Next, $\texttt{p}^{\epsilon\texttt{f}_1}$ and $\texttt{base}^\epsilon$ are used to adorn the body of r_1 so as to get

$$\rho_1: \quad \texttt{p}^{\texttt{f}_1\texttt{f}_2}(\texttt{X}, \texttt{f}(\texttt{X})) \leftarrow \texttt{p}^{\epsilon\texttt{f}_1}(\texttt{Y}, \texttt{X}), \texttt{base}^\epsilon(\texttt{Y})$$

from which we derive the new adorned predicate symbol $\texttt{p}^{\texttt{f}_1\texttt{f}_2}$, and the adornment definition $\texttt{f}_2 = \texttt{f}(\texttt{f}_1)$. Intuitively, the body of ρ_1 is coherently adorned because $\texttt{Y}$ is always associated with the same adornment symbol ϵ. Using the new adorned predicate symbol $\texttt{p}^{\texttt{f}_1\texttt{f}_2}$, we can adorn rule r_2 and get

$$\rho_2: \quad \texttt{p}^{\epsilon\texttt{f}_1}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{p}^{\texttt{f}_1\texttt{f}_2}(\texttt{f}(\texttt{X}), \texttt{f}(\texttt{Y})).$$

At this point, we are not able to generate new adorned rules (using the adorned predicate symbols generated so far) with coherently adorned bodies and the transformation terminates. In fact, $\texttt{p}^{\texttt{f}_1\texttt{f}_2}(\texttt{Y}, \texttt{X}), \texttt{base}^\epsilon(\texttt{Y})$ is not coherently adorned because the same variable $\texttt{Y}$ is associated

bottom-up fashion, and therefore differs from the classical formulation relying on a top-down evaluation strategy. However, programs relying on a top-down evaluation strategy can be rewritten into programs whose bottom-up evaluation gives the same result.

with both $\mathtt{f}_1$ and ϵ; moreover, $\mathtt{p}^{\epsilon\mathtt{f}_1}(\mathtt{f}(\mathtt{X}), \mathtt{f}(\mathtt{Y}))$ is not coherently adorned because $\mathtt{f}(\mathtt{X})$ does not comply with the (simple) term structure described by ϵ.

To determine termination of the bottom-up evaluation of $\mathcal{P}_7$, we can apply current termination criteria to $\mathcal{P}_7^\mu = \{\rho_0, \rho_1, \rho_2\}$ rather than $\mathcal{P}_7$. □

It is worth noting that the rewriting technique ensures that if $\mathcal{P}_7^\mu$ is recognized as terminating, so is $\mathcal{P}_7$. Notice also that both $\mathcal{P}_7$ and $\mathcal{P}_7^\mu$ are recursive, but while some termination criteria (e.g., the argument-restricted and Γ-acyclicity criteria) detect $\mathcal{P}_7^\mu$ as terminating, none of the current termination criteria is able to realize that $\mathcal{P}_7$ terminates.

References

1 Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable functions in ASP: Theory and implementation. In *ICLP*, pages 407–424, 2008.

2 Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo : A new grounder for answer set programming. In *LPNMR*, pages 266–271, 2007.

3 Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080, 1988.

4 Sergio Greco, Cristian Molinaro, and Francesca Spezzano. *Incomplete Data and Data Dependencies in Relational Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

5 Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. Bounded programs: A new decidable class of logic programs with function symbols. In *IJCAI*, 2013.

6 Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. Logic programming with function symbols: Checking termination of bottom-up evaluation through program adornments. In *ICLP (to appear in TPLP journal)*, 2013.

7 Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. On the termination of logic programs with function symbols. In *ICLP (Technical Communications)*, pages 323–333, 2012.

8 Yuliya Lierler and Vladimir Lifschitz. One more decidable class of finitely ground programs. In *ICLP*, pages 489–493, 2009.

9 Tommi Syrjanen. Omega-restricted logic programs. In *LPNMR*, pages 267–279, 2001.

Automated nontermination proofs by safety proofs

Hong-Yi Chen[1], Byron Cook[1,2], Carsten Fuhs[1], Kaustubh Nimkar[1], and Peter O'Hearn[1]

1 **University College London**
Gower Street, London, United Kingdom
hongyichen00@gmail.com, c.fuhs@cs.ucl.ac.uk, k.nimkar@cs.ucl.ac.uk, p.ohearn@ucl.ac.uk

2 **Microsoft Research Cambridge**
21 Station Road, Cambridge, United Kingdom
bycook@microsoft.com

Abstract

We show how the problem of nontermination proving can be reduced to a question of under-approximation search guided by a safety prover. This reduction leads to new nontermination proving implementation strategies based on existing tools for safety proving. Our preliminary implementation has shown favorable results over existing tools.

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 Nontermination analysis, Safety analysis, Closed recurrence set

1 Introduction

The problem of proving program *non*termination represents an interesting complement to termination as, unlike safety, termination's falsification cannot be witnessed by a finite trace. While the problem of proving termination has now been extensively studied, the search for reliable and scalable methods for proving nontermination remains open.

In this extended abstract we present a new method of proving nontermination based on a reduction to safety proving that leverages the power of existing tools. An iterative algorithm is developed which uses counterexamples to a fixed safety property to refine an underapproximation of a program. With our approach, existing safety provers can now be employed to prove nontermination of programs that previous techniques could not handle. Not only does the new approach perform better, it also leads to nontermination proving tools supporting features previous tools could not handle reliably, *e.g.* heap, nonlinear commands, and nondeterminism.

Recall that *safety* of a program means that no undesired (or *unsafe*) program state can be reached from any initial state of the program. On source code level, unsafe states can be expressed by a statement "`assert`(φ)" for a Boolean expression φ. Then the program is unsafe *iff* there exists a run of the program from an initial state such that in this run, φ is violated at the position of this statement. Techniques for safety proving include counter-example based abstraction refinement as in SLAM [1] or interpolation as in IMPACT [9].

Gupta *et al.* [8] characterize nontermination of a program by the existence of a *recurrence set*. A program is nonterminating *iff* there exists a recurrence set for the program's transition relation. The existence of a recurrence set implies that the program does not terminate when from a reachable state in the recurrence set we can always choose the next transition to a state that also belongs to the recurrence set.

As opposed to their approach we search for an underapproximation of the original program that *never* terminates, regardless of the values introduced by nondeterministic assignments during the run. This property is characterized by a closed recurrence set for the transition relation of the underapproximation. For every state in the closed recurrence set, every possible transition leads us to a state that belongs to the closed recurrence set. As "never terminates" can be encoded as safety property, we can then iterate a safety prover together with a method of underapproximating based on counterexamples. We have to be careful, however, to find the right underapproximation in order to avoid unsoundness.

We describe our algorithm informally. It takes as input a program P and a loop L in P to be considered for nontermination. We then mark the L's exit location as an error location and invoke a safety checker. Any path that reaches the exit location is the counterexample to safety and it cannot contribute towards the nontermination of the loop. We then find an underapproximation of P that eliminates this path. Our algorithm either finds a precondition for P or a precondition after a nondeterministic assignment statement such that every state which fulfills this precondition reaches the error location when the counterexample path is followed. To eliminate the counterexample path we then negate this condition and add a restriction on the state space to get our underapproximating refinement. We continue this procedure as long as there is some counterexample to safety of our current underapproximation.

Note that sometimes our refinements are too weak and the search for a safe underapproximation may lead to divergence. In such cases we use suitable heuristics to strengthen our underapproximation which then avoids the problem of divergence.

Let P' be our final underapproximation that is safe. We refer to the loop L after refinements as L'. To prove nontermination we first need to ensure that the loop L' in P' is still reachable after the refinements. This can again be encoded as a safety problem, this time marking the loop header as an error location. If safety is violated, the counterexample path represents the path to L' ensuring the reachability of L'.

Note that our refinements also restrict the choices for nondeterministic assignment statements. We finally ensure that for every reachable state at the nondeterministic assignment inside L', we can still make a choice so that execution is never halted. When the check succeeds, we report nontermination. In the final underapproximation, the set of all reachable states at the loop header of L' forms a closed recurrence set for the loop's transition relation.

2 Example

We now describe our algorithm using a simple example. Consider the following program. In this program the command $\mathsf{i} := \texttt{nondet}()$ represents nondeterministic value introduction into the variable i (*e.g.* user input). The loop in this program is nonterminating when the program is invoked with appropriate inputs and when appropriate choices for the `nondet` assignment are made. We are interested in automatically detecting this nontermination.

```
if (k ≥ 0)
    skip;
else
    i := −1;
while (i ≥ 0) {
    i := nondet();
}
i := 2;
```

In order to find the desired underapproximation for our example, we first introduce an `assume` statement (where "`assume`(φ)" can be implemented by "`if` ($\neg\varphi$) `exit`") at the beginning with the initial precondition `true`. We also place `assume(true)` statements after each use of `nondet`. We then put an `assert(false)` statement at points where the loop under consideration exits (thus encoding the "never terminates" property). See Figure 1(a).

We then use a safety checker (here: for programs on integer data) to search for paths that violate this assertion. Any error path clearly cannot contribute towards the nontermination

```
assume(true);

if (k ≥ 0)
   skip;
else
   i := −1;

while (i ≥ 0) {
   i := nondet();
   assume(true);
}

assert(false);

i := 2;
```
(a)

```
assume(k ≥ 0);

if (k ≥ 0)
   skip;
else
   i := −1;

while (i ≥ 0) {
   i := nondet();
   assume(true);
}

assert(false);

i := 2;
```
(b)

```
assume(k ≥ 0 ∧ i ≥ 0);

if (k ≥ 0)
   skip;
else
   i := −1;

while (i ≥ 0) {
   i := nondet();
   assume(true);
}

assert(false);

i := 2;
```
(c)

```
assume(k ≥ 0 ∧ i ≥ 0);

if (k ≥ 0)
   skip;
else
   i := −1;

while (i ≥ 0) {
   i := nondet();
   assume(i ≥ 0);
}

assert(false);

i := 2;
```
(d)

```
assume(k ≥ 0 ∧ i ≥ 0);

if (k ≥ 0)
   skip;
else
   i := −1;

assert(false);

while (i ≥ 0) {
   i := nondet();
   assume(i ≥ 0);
}
```
(e)

```
assume(k ≥ 0 ∧ i ≥ 0);
assume(k ≥ 0);

skip;
while (i ≥ 0) {
   i := nondet();
   assume(i ≥ 0);
}
```
(f)

Figure 1 Original instrumented program (a) and its successive underapproximations (b), (c), (d). Reachability check for the loop (e), and nondeterminism-`assume` that must be checked for satisfiability (f).

of the loop. Initially, as a first counterexample to safety, we might get the path $\mathsf{k} < 0,\ \mathsf{i} := -1,\ \mathsf{i} < 0$, from a safety prover. We now want to determine from which states we can reach `assert(false)` and eliminate those states. Using a precondition computation similar to Calcagno *et al.* [4] we find the condition $\mathsf{k} < 0$. Note that our condition gives a set of states that actually reach the error location. To rule out the states $\mathsf{k} < 0$ we can add the negation (*e.g.* $\mathsf{k} \geq 0$) to the precondition `assume` statement. See Figure 1(b).

We then try to prove the `assert` statement unreachable for the program in Figure 1(b). Here we might get the path $\mathsf{k} \geq 0,\ \texttt{skip},\ \mathsf{i} < 0$, which again violates the assertion. For this path we would discover the precondition $\mathsf{k} \geq 0 \wedge \mathsf{i} < 0$, and to rule out these states we refine the precondition `assume` statement with "$\texttt{assume}(\mathsf{k} \geq 0 \wedge \mathsf{i} \geq 0)$". See Figure 1(c).

On this program our safety prover will again fail, perhaps resulting in the path $\mathsf{k} \geq 0,\ \texttt{skip},\ \mathsf{i} \geq 0,\ \mathsf{i} := \texttt{nondet}(),\ \mathsf{i} < 0$. In this case our algorithm stops computing the precondition at the command $\mathsf{i} := \texttt{nondet}()$. Here we would learn that at the nondeterministic command the result must be $\mathsf{i} < 0$ in order to violate the assertion, thus we would refine the assume statement just after the nondeterministic statement with the negation of $\mathsf{i} < 0$ and get "$\texttt{assume}(\mathsf{i} \geq 0)$". See Figure 1(d).

The program in Figure 1(d) cannot violate the assertion, and thus we have hopefully computed the desired underapproximation to the transition relation needed in order to prove nontermination. However, for soundness, it is essential to ensure that the loop in Figure 1(d) is still reachable, even after the successive restrictions to the state space. We encode this condition as a safety problem. See Figure 1(e). This time we add `assert(false)` before the loop and aim to prove that the assertion is violated. The existence of a path violating the assertion ensures that the loop in Figure 1(d) is reachable. In this case the assertion is reachable, and thus the loop is still reachable. The path violating the assertion is our desired path to the loop which we refer to as *stem*. Figure 1(f) shows the stem and the loop.

Finally we need to ensure that the `assume` statement in Figure 1(f) can always be satisfied with some choice of i by any reachable state from the restricted pre-state. This is necessary since our underapproximations may accidentally have eliminated not only the paths to the loop's exit location, but also all of the non-terminating paths inside the loop. We ensure this by calculating a location invariant *inv* before the `nondet` statement. We then check that the formula $\mathit{inv} \rightarrow \exists \mathsf{i}'.\mathsf{i}' \geq 0$ is valid. Even the weakest invariant `true` can be sufficient to easily prove the validity of the above formula. This ensures that for every reachable state at the nondeterministic assignment we can still make a choice so that execution is never halted. Once this check succeeds we report nontermination.

3 Experiments

In order to assess the impact of our approach, we have built a preliminary implementation within the tool T2 [2] [5] and evaluated it empirically comparing with the following tools:

- TNT [8]. Note that the original TNT tool was not available and thus we have reimplemented the underlying constraint-based algorithm with Z3 [6] as SMT backend.
- AProVE [7], using the Java Bytecode frontend. When proving nontermination of Java Bytecode programs, AProVE implements the SMT-based nontermination analysis by Brockschmidt *et al.* [3].
- Julia [13]: Julia implements an approach via a reduction from Java Bytecode to constraint logic programming described by Payet and Spoto [11].

As a benchmark set, we applied the tools on a set of 495 benchmarks from a variety of applications (*e.g.* Windows device drivers, the Apache web server, the PostgreSQL server,

integer approximations of numerical programs from a book on numerical recipes [12], integer approximations of benchmarks from LLBMC [10] and other tool evaluations).

We conducted three sets of experiments. The first set consists of all the 77 examples previously known to be nonterminating, the second set consists of all the 258 examples previously known to be terminating, and the third set consists of all the 160 examples for which no previous results are known and which are too large to render a manual analysis feasible. We used the first set of examples to assess the efficiency of the algorithm, the second set to demonstrate the algorithm's soundness, and the third set to check if our algorithm scales well on relatively large and complicated examples. The results demonstrate that our procedure is overwhelmingly the most successful tool and does not show erroneous behavior in our experiments.

4 Conclusion

In this paper we introduced a new method of proving nontermination. The idea is to split the reasoning in two parts: a safety prover is used to prove that a loop in an underapproximation of the original program *never* terminates; meanwhile failed safety proofs are used to calculate the underapproximation. Our implementation has shown that our approach leads to performance improvements against previous tools where they are applicable.

References

1 Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *Proc. CAV '01*, volume 2102 of *LNCS*, pages 260–264, 2001.

2 Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In *Proc. CAV '13*, volume 8044 of *LNCS*, pages 413–429, 2013.

3 Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. Automated detection of non-termination and `NullPointerException`s for Java Bytecode. In *Proc. FoVeOOS '11*, volume 7421 of *LNCS*, pages 123–141, 2012.

4 Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, 58(6):26, 2011.

5 Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. lexicographic termination proving. In *Proc. TACAS '13*, volume 7795 of *LNCS*, pages 47–61, 2013.

6 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS '08*, volume 4963 of *LNCS*, pages 337–340, 2008.

7 Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. IJCAR '06*, volume 4130 of *LNAI*, pages 281–286, 2006.

8 Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *Proc. POPL '08*, pages 147–158, 2008.

9 Ken McMillan. Lazy abstraction with interpolants. In *Proc. CAV '06*, volume 4144 of *LNCS*, pages 123–136, 2006.

10 Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *Proc. VSTTE '12*, LNCS, pages 146–161, 2012.

11 Étienne Payet and Fausto Spoto. Experiments with non-termination analysis for Java Bytecode. In *Proc. BYTECODE '09*, volume 253 of *ENTCS*, pages 83–96, 2009.

12 William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 1989.

13 Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for Java bytecode based on path-length. *ACM TOPLAS*, 32(3), 2010.

The Ordinal Path Ordering

Nachum Dershowitz

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

Abstract

We reformulate Okada's version of Takeuti's ordinal diagrams as inference rules in the style of the abstract path ordering.

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

Keywords and phrases Termination, ordinals, ordinal diagrams, path orderings

1 Purpose

Ordinals are used in proof-theoretical investigations to chraracterize the logical complexity of formal systems of analysis and of specific mathematical theorems. Ordinal diagrams, the original version of which is due to Takeuti [12], are one of the most powerful syntactic notations of ordinals that have been devised. They are related to the Friedman's [11] and Kříž's [6] gap version of Kruskal's famous Tree Theorem [7, 8], in that the well-orderedness of diagrams follows from the Gap Tree Theorem. Partially ordered versions of ordinal diagrams are only possible in restricted cases; see [10, 5, 4].

We reformulate ordinal diagrams in the style Okada [9] by ignoring forests (that is, unconnected trees), which can be compared as multisets of trees [2]. This re-articulation highlights the (as yet unexploited) similarity of the ordering of diagrams with the abstract path ordering [1], designed to prove termination of term rewriting systems.

2 Atomic Case

Given base sets Σ and Π, well-ordered by a *precedence* $\succcurlyeq$, with all of Π greater than all of Σ, we define a well-ordering $>_\infty$ over *unordered* trees T with leaves from Σ and (internal) nodes from Π, that is:

$$T ::= \Sigma \mid \Pi(T, \ldots, T)$$

Notation: s, t, s_k, t_ℓ are trees of T; α, β are nodes from Π; a, b are leaves from Σ; $u, v \in T \setminus \Sigma$, the non-leaf trees.

Stratified subtrees. By an α-subtree we mean an immediate subtree of some α node in the tree for which there are no smaller nodes en route from the root. Define the relation $\rhd_\alpha$ as follows:

$$\frac{\beta \geq \alpha \quad s \trianglerighteq_\alpha u}{\beta(\ldots, s, \ldots) \rhd_\alpha u}$$

where $>$ here is $\succcurlyeq$ (the ordering on nodes). As usual, we are using $\geq$ and $\trianglerighteq$ for the reflexive closures.

This relation is transitive.

The following three definitions are mutually recursive.

Minimal operator. *The smallest node in a tree (or trees) that is greater than α.*

$$\begin{aligned} \mu_\alpha(a) &= \infty \\ \mu_\alpha(\beta(s_1,\dots,s_n)) &= \begin{cases} \min\{\beta, \mu_\alpha(s_1,\dots,s_n)\} & \beta > \alpha \\ \mu_\alpha(s_1,\dots,s_n) & \alpha \geq \beta \end{cases} \\ \mu_\alpha(s_1,\dots,s_n) &= \min\{\mu_\alpha(s_1),\dots,\mu_\alpha(s_n)\} \end{aligned}$$

where $>$ is $\gtrdot$ and minima are taken with respect to $\gtrdot$, with ∞ greater than all node values.

Stratified ordering. For each α, the following is a well-ordering.

$$\frac{a > b}{a >_\alpha b}\ (\alpha 1) \qquad \frac{}{u >_\alpha b}\ (\alpha 2) \qquad \frac{u \rhd_\alpha \circ \geq_\alpha v}{u >_\alpha v}\ (\alpha 3) \qquad \frac{u >_{\mu_\alpha\{u,v\}} v \quad u >_\alpha /\rhd_\alpha v}{u >_\alpha v}\ (\alpha 4)$$

where $>$ is $\gtrdot$ and $u >_\alpha /\rhd_\alpha v$ means $u >_\alpha s$ for every $s \lhd_\alpha v$. Recall that a and b are leaves; u and v are not. Clearly, each stratum $>_\alpha$ has the "stratified" subtree property, namely: $x \rhd_\alpha y$ implies $x >_\alpha y$.

We note that $u >_\beta v$ iff $u >_\gamma v$ whenever $\gamma = \mu_\alpha(u,v) \gtrdot \beta \gtrdot \alpha$, there being no β-subtrees in u or v, so $(\alpha 3)$ is not applicable and the second hypothesis of $(\alpha 4)$ is vacuous.

Target ordering. The *ordinal path ordering* $>_\infty$ is a dependent lexicographic pair, consisting of the ordering $\gtrdot$ on nodes followed by the multiset extension of the ordering $>_\alpha$, selected by the shared node α, on immediate subtrees.

$$\frac{}{u >_\infty b}\ (\infty 1) \qquad \frac{\alpha > \beta}{\alpha(\dots,s_k,\dots) >_\infty \beta(\dots,t_\ell,\dots)}\ (\infty 2) \qquad \frac{\{\dots,s_k,\dots\} \gg_\alpha \{\dots,t_\ell,\dots\}}{\alpha(\dots,s_k,\dots) >_\infty \alpha(\dots,t_\ell,\dots)}\ (\infty 3)$$

where $>$ is $\gtrdot$ and $\gg_\alpha$ is the multiset extension of $>_\alpha$.

Another way to express this top-level ordering is to extend $\gtrdot$ so that non-leaves are compared by comparing their root nodes in the node ordering and non-leaves are always greater than leaves, and to define $\gg$ to compare non-leaf trees with equal root values to each other by comparing the multiset of immediate subtrees in the order indexed by the root-node value. (Trees with incomparable roots are incomparable.) Then $>_\infty$ is the union of these two (disjoint) orderings, and we can economize by using the following rules:

$$\frac{s \gtrdot t}{s >_\infty t}\ (\infty 1{,}2) \qquad \frac{s \gg t}{s >_\infty t}\ (\infty 3)$$

3 Tree Case

Given a base set Σ with minimal element 0, well-ordered by $\gtrdot$, we define a well-ordering $>_\infty$ over *unordered* (trees of) trees T with leaves from Σ and *trees* for internal nodes, that is:

$$T ::= \Sigma \mid T(T,\dots,T)$$

There is no longer a separate node vocabulary Π. Hence, the ordering on nodes is no longer $\gtrdot$, but instead is the lowest stratum $>_0$ of the same ordering as is being defined on trees. The definition is the same, except that $\gtrdot$ is replaced by $>_0$ throughout.
Notation: $\alpha, \beta, s, t, s_k, t_\ell$ are trees of T; a, b are leaves from Σ; $u, v \in T \setminus \Sigma$.

The following four definitions are mutually recursive.

Stratified subtrees. *A subtree of an α node with no smaller nodes en route.*

$$\frac{\beta \geq \alpha \quad s \trianglerighteq_\alpha u}{\beta(\dots,s,\dots) \rhd_\alpha u}$$

where $>$ here is $>_0$.

Minimal operator. As above: *The smallest node in a tree (or trees) that is greater than* α.

$$\begin{aligned} \mu_\alpha(a) &= \infty \\ \mu_\alpha(\beta(s_1,\dots,s_n)) &= \begin{cases} \min\{\beta, \mu_\alpha(s_1,\dots,s_n)\} & \beta > \alpha \\ \mu_\alpha(s_1,\dots,s_n) & \alpha \geq \beta \end{cases} \\ \mu_\alpha(s_1,\dots,s_n) &= \min\{\mu_\alpha(s_1),\dots,\mu_\alpha(s_n)\} \end{aligned}$$

where $>$ is $>_0$ and minima are taken with respect to $>_0$.

Stratified ordering. For each α, the following is a well-ordering.

$$\frac{a > b}{a >_\alpha b} \qquad \frac{}{u >_\alpha b} \qquad \frac{u \rhd_\alpha \circ \geq_\alpha v}{u >_\alpha v} \qquad \frac{u >_{\mu_\alpha\{u,v\}} v \quad u >_\alpha /\rhd_\alpha v}{u >_\alpha v}$$

where $>$ is $>_0$.

Target ordering. *Dependent lexicographic pair, ordering the roots followed by the multiset extension of the selected ordering on immediate subtrees.*

$$\frac{}{u >_\infty b} \qquad \frac{\alpha > \beta}{\alpha(\dots,s_k,\dots) >_\infty \beta(\dots,t_\ell,\dots)} \qquad \frac{\{\dots,s_k,\dots\} \gg_\alpha \{\dots,t_\ell,\dots\}}{\alpha(\dots,s_k,\dots) >_\infty \alpha(\dots,t_\ell,\dots)}$$

where $>$ is the node ordering $>_0$ and $\gg_\alpha$ is the multiset extension of $>_\alpha$.

4 Examples

We focus in the coming examples on unary trees (strings) and the atomic ordering, though the tree case is the more interesting.

4.1 An Example

Consider the rewriting rule $ffx \to fgfx$, with $\Pi = \{f, g\}$ and Σ anything, and let $f \succ g$. First, notice that $fx >_\infty gy$, for any y, and in particular $fx >_\infty gfx$. So the target ordering $>_\infty$ does not have the subtree property (which is what makes it useful in this—and many other—cases).

Since f is the largest node value in the precedence, we have $fx >_\infty gy$ for all trees x and y. Similarly, we have

$$\dfrac{\dfrac{\dfrac{f \succ g}{fx >_\infty gfy}\,(\infty 2) \qquad \dfrac{}{fx >_f /\rhd_f gfy}}{fx >_f gfy}\,(\alpha 4)}{\dfrac{\{fx\} \gg_f \{gfy\}}{ffx >_\infty fgfy}\,(\infty 3)}$$

since there are no f-subtrees in gfy.

Since $>_\infty$ is total and well-ordered, it cannot be monotonic. Still we want $s >_\infty t$ whenever s rewrites to t; in other words, we want $vffw >_\infty vfgfw$, for all $v, w \in \Pi^*$.

We show that if $u >_\infty v$, for u and v having the same root node, then $\alpha u >_\infty \alpha v$, for any node α (f or g). There are four cases: $gx >_\infty gy$ implies $fgx >_\infty fgy$ and $ggx >_\infty ggy$ and $fx >_\infty fy$ implies $ffx >_\infty ffy$ and $gfx >_\infty gfy$.

It is easy to verify that $g\,x >_\infty g\,y$ implies $f\,g\,x >_\infty f\,g\,y$ for all strings x and y:

$$\dfrac{\dfrac{g\,x >_\infty g\,y \quad \overline{g\,x >_f / \rhd_f g\,y}}{g\,x >_f g\,y}\ (\alpha 4)}{f\,g\,x >_\infty f\,g\,y}\ (\infty 3)$$

there being no f-subtrees in gy. (So this is also true for larger alphabets, as long as f is maximal.)

Furthermore, $fx >_\infty fy$ implies $ffx >_\infty ffy$:

$$\dfrac{\dfrac{fx >_\infty fy \quad \overline{\begin{array}{c}\vdots\\ fx >_f / \rhd_f fy\end{array}}}{fx >_f fy}\ (\alpha 4)}{ffx >_\infty ffy}\ (\infty 3)$$

because, for any $z \trianglelefteq_f y$, we have

$$\dfrac{\dfrac{\overline{fx \rhd_f x}}{fx >_f x}\ (\alpha 3) \qquad \dfrac{\dfrac{fx >_\infty fy}{x >_f y} \qquad y \trianglerighteq_f z}{x >_f z}}{fx >_f z}$$

and $x >_\alpha y \trianglerighteq_\alpha z$ always implies $x >_\alpha z$ on account of the subtree property and transitivity. Virtually the same argument (with one additional step) shows that $fx >_\infty fy$ implies $gfx >_\infty gfy$.

Lastly, one can show that $g\,x >_\infty g\,y$ implies $g\,g\,x >_\infty g\,g\,y$:

$$\dfrac{\dfrac{\dfrac{g\,x >_\infty g\,y}{\left[\ g\,x >_f g\,y\ \right]}\ (\alpha 4) \qquad \dfrac{g\,x >_g x >_g y}{g\,x >_g / \trianglerighteq_g y}}{g\,x >_g g\,y}\ (\alpha 4)}{g\,g\,x >_\infty g\,g\,y}\ (\infty 3)$$

there being no f-subtrees in $g\,y$, and $x >_g y$ being the only way that one can have $g\,x >_\infty g\,y$. The bracketed step is omitted if f does not occur in x or y.

4.2 A Counterexample

For the purposes of a counterexample in [3] (showing the necessity of a subterm condition for the critical-pair lemma in the case of normal conditional rewriting), the following inequalities were needed: $a > b$, $fa > ga$, $hfa > c > kfa$, $c > kgb$, $fx > hfx$ (!), $fx > kgb$, $hx > kx$. For that, we can interpret terms as follows:

$$\begin{aligned}
[\![a]\!] &= 1 \\
[\![b]\!] &= 0 \\
[\![c]\!] &= 0(1(1), 1) \\
[\![h(x)]\!] &= 0([\![x]\!], 2) \qquad \text{i.e. } [\![h]\!] = \lambda x.0(x, 2) \\
[\![f(x)]\!] &= 1([\![x]\!]) \\
[\![k(x)]\!] &= 0([\![x]\!]) \\
[\![g(x)]\!] &= 0([\![x]\!])
\end{aligned}$$

5 Conclusion

The use of ordinal diagrams, as made simple by the above inference rules, holds out some hope for helping in difficult (non-simplifying) termination proofs.

Acknowledgement. I thank Mitsu Okada and the readers for their questions and comments.

References

1 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`.

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

3 Nachum Dershowitz and Mitsuhiro Okada. Proof-theoretic techniques and the theory of rewriting. In *Proceedings of the Third IEEE Symposium on Logic in Computer Science (LICS)*, pages 104–111, Edinburgh, Scotland, July 1988. Available at `http://nachum.org/papers/ProofTheoretic.pdf`.

4 Nachum Dershowitz and Iddo Tzameret. Gap embedding for well-quasi-orderings. *Electr. Notes Theor. Comput. Sci.*, 84:80–90, 2003. Available at `http://www.sciencedirect.com/science/article/pii/S1571066104808466`.

5 Lev Gordeev. Generalizations of the Kruskal-Friedman theorems. *The Journal of Symbolic Logic*, 55(1):157–181, 1990.

6 Igor Kříž. Well-quasiordering finite trees with gap-condition. Proof of Harvey Friedman's conjecture. *Ann. of Math.*, 130:215–226, 1989.

7 Joseph B. Kruskal. Well-quasiordering, the Tree Theorem, and Vazsonyi's conjecture. *Trans. American Mathematical Society*, 95:210–223, May 1960.

8 Crispin St. J. A. Nash-Williams. On well-quasi-ordering finite trees. *Proc. Cambridge Phil. Soc.*, 59:833–835, October 1963.

9 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.

10 Mitsuhiro Okada and Gaisi Takeuti. On the theory of quasi-ordinal diagrams. In *Logic and Combinatorics (Arcata, CA, 1985)*, volume 65 of *Contemp. Math.*, pages 295–308. Amer. Math. Soc., Providence, RI, 1987.

11 Stephen G. Simpson. Nonprovability of certain combinatorial properties of finite trees. In L. A. Harrington, editor, *Harvey Friedman's Research on the Foundation of Mathematics*, pages 87–117. Elsevier, 1985.

12 Gaisi Takeuti. Ordinal diagrams. II. *J. Math. Soc. Japan*, 12:385–391, 1960.

Dependency Pairs are a Simple Semantic Path Ordering

Nachum Dershowitz

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

Abstract

We explicate the relation between the older semantic path ordering of Kamin and Lévy and the newer dependency-pair method of Arts and Giesl.

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

Keywords and phrases Termination, semantic path orderings, dependency pairs

1 Introduction

As pointed out by Cristina Borralleras in her dissertation [5, Thm. 7.3.2] (see also [4, Section 5.4]), the dependency-pair method of Thomas Arts and Jürgen Giesl [1] is actually a special case of the semantic path ordering of Sam Kamin and Jean-Jacques Lévy [14]. We expand and elaborate on that relationship in what follows.

2 The Semantic Path Ordering

Let irreflexive $\gtrdot$ and reflexive $\geq$ be two binary relations on terms such that their combined "modulo" relation $\gtrdot/\geq$ ($= \geq^* \circ \gtrdot \circ \geq^*$) is terminating (in the sense of [2]). This means that there is no sequence of terms

$$s_0 \geq \cdots \geq s'_0 \gtrdot s_1 \geq \cdots \geq s'_1 \gtrdot s_2 \geq \cdots \geq s'_2 \gtrdot \cdots$$

containing *infinitely* many $\gtrdot$ steps and is equivalent to saying that the modulo relation's transitive closure $(\gtrdot/\geq)^+$ is well-founded. Let's refer to this relation $\gtrdot/\geq$ as the *semantic* ordering. Typically, but not exclusively, it is defined via a homomorphism from terms to some well-founded domain.

Despite the deliberately misleading choice of symbols, $\geq$ need not be the reflexive-closure of $\gtrdot$. But if it is, then $\gtrdot/\geq$ is terminating if (and only if) $\gtrdot$ is.

If both relations are transitive and also are compatible with each other (meaning that either $\geq \circ \gtrdot \subseteq \gtrdot$ or $\gtrdot \circ \geq \subseteq \gtrdot$; see [14, p. 14] and [1, n. 5]), and provided $\gtrdot$ is itself well-founded, then the above termination condition holds. We prefer, however, not to bother requiring transitivity.

Let $\gtrdot$ and $\geq$ be as above and let $\blacktriangleright$ denote the *immediate* subterm relation. We can define a very simple semantic path ordering $\succ/\succsim$, with (base) semantic ordering $\gtrdot/\geq$, as follows:

$$\frac{\exists s_i.\ s \blacktriangleright s_i \succsim t}{s \succ t,\ s \succsim t} \qquad \frac{s \gtrdot t,\ \forall t_j \blacktriangleleft t.\ s \succ t_j}{s \succ t,\ s \succsim t} \qquad \frac{s \geq t,\ \forall t_j \blacktriangleleft t.\ s \succ t_j}{s \succsim t}$$

If $\doteq$ is the intersection $\geq \cap \leq$ and $\approx$ is $\succsim \cap \precsim$, then it follows that

$$\frac{s \doteq t,\ \forall t_j \blacktriangleleft t.\ s \succ t_j, \forall s_i \blacktriangleleft s.\ t \succ s_i}{s \approx t}$$

This is in essence the semantic path ordering of [14] (cf. [11, Def. 4]) over the semantic relation $\gtrsim$ with a trivial (empty) functional (lifting the ordering from subterms to terms), so that there is no recursion on subterms. We have added $\gtrsim$ to the definition (in particular, in the third case) in the obvious way. (The use of a quasi-order was also suggested in [14, p. 10].) Technically, the relation satisfies the *weak* (non-strict) monotonicity condition on the functional (cf. [14, p. 12]). It also satisfies the requirements of the general path ordering [11]. The conditions in the definition could be relaxed somewhat to take into account the possible non-transitivity of the relations.

▸ **Theorem 1.** *The simple semantic path ordering $\succ/\gtrsim$ is terminating.*

The proof is essentially as in [14] (and [11, Thm. 2]), but takes the quasi-ordered case into account.

Proof. Suppose the path relation is not terminating and look at a minimal counterexample $u_0 \gtrsim^* \succ \gtrsim^* u_1 \gtrsim^* \succ \gtrsim^* u_2 \gtrsim^* \succ \gtrsim^* \cdots$, minimal with respect to subterm. Let's number the cases as follows:

$$\frac{s_i \gtrsim t}{s \succ_1 t} \qquad \frac{s \gtrsim t,\ s \succ t_j}{s \succ_2 t} \qquad \frac{s \geq t,\ s \succ t_j}{s \gtrsim_3 t}$$

The minimal counterexample never employs the first case: Clearly, it is not the case that $u_0 \gtrsim_1 u_1$, since then the sequence beginning with the subterm of u_0 that justifies the inequality would be smaller. Suppose $u_i \succ_1 u_{i+1}$ is the first occurrence of case 1 in the counterexample ($i \geq 1$), and that it is justified by $t_i \gtrsim u_{i+1}$ for subterm t_i of u_i. Whether $u_{i-1} \succ_2 u_i$ or $u_{i-1} \gtrsim_3 u_i$, we would have $u_{i-1} \succ t_i \gtrsim u_{i+1}$ by the side requirement $s \succ t_j$ of the other two cases.

Whenever $u_i \succ_2 u_{i+1}$, we have $u_i \succ u_{i+1}$; when $u_i \gtrsim_3 u_{i+1}$, we have $u_i \geq u_{i+1}$. This contradicts termination of $\succ/\geq$. ◂

For reduction in a semantic path ordering of a term-rewriting system to provide termination, one shows first of all that $\ell \succ r$ for every rule $\ell \to r$ (meaning that $\ell\sigma \succ r\sigma$ for every substitution σ). As explained in [14, pp. 14–15], the semantic path ordering is not necessarily weakly monotonic. That is, it need *not* be the case that

$$s \succ t \Rightarrow f(\ldots, s, \ldots) \gtrsim f(\ldots, t, \ldots)$$

Therefore, one also needs to demonstrate ([14, p. 14, *post correctionem*])

$$s \to t \Rightarrow f(\ldots, s, \ldots) \geq f(\ldots, t, \ldots) \qquad (*)$$

so that $s \to t$ implies either $s \succ t$ (if it is a top-rewrite) or $s \gtrsim t$ (if not), which is enough to ensure that $s \gtrsim t$ whenever $s \to t$ and give termination [7, Second Termination Theorem]. With condition (*), the intersection of $\gtrsim$ and $\to$ is monotonic, since $s \gtrsim t$ and $f(\ldots, s, \ldots) \geq f(\ldots, t, \ldots)$ yield $f(\ldots, s, \ldots) \gtrsim_3 f(\ldots, t, \ldots)$.

3 The Dependency-Pair Method

Consider now the (basic) dependency-pair framework [1, Thm. 7]. For every rule $\ell \to r$ and nonvariable (not necessarily proper) subterm u of r that is not headed by a constructor (a symbol that never appears at the head of a left-hand side of any rule), we have a dependency

pair $\ell \to u$. Suppose we are given a pair of (partial and quasi-) orderings $\gtrdot, \gtrsim$ that are compatible (as above), and such that $\gtrdot$ is well-founded and $\gtrsim$ is weakly monotonic, meaning:

$$s \gtrsim t \Rightarrow f(\ldots, s, \ldots) \gtrsim f(\ldots, t, \ldots)$$

Then, a rewrite system terminates if $\ell \gtrsim r$ for every rule and $\ell \gtrdot u$ for every dependency pair (again, for all substitutions).

▸ **Theorem 2.** *If a rewrite system can be shown terminating by the basic dependency-pair method using the pair $\gtrsim$ and $\gtrdot$, then it is terminating by the semantic path ordering method using the same pair.*

Proof. Modify the ordering $\gtrdot$ so that all terms headed by constructors are smaller than all those that are not (see [5, Sect. 7.3], [9, n. 9], [4, Section 5.4]). Clearly, the ordering remains terminating and this change has no effect on dependency pairs, because constructor-headed terms are never compared. To maintain compatibility, also remove from $\gtrdot$ any pair whose left-side is a constructor term (they are never needed), and remove from $\gtrsim$ any pair with left-side a constructor and right-side not (also unnecessary).

Consider a rule $\ell \to r$. We show that $\ell \succ r$. If r is a proper subterm of ℓ, and in particular if r is a variable, then $\ell \succ_1 r$. If not, then $\ell \gtrdot r$, since it is one of the dependency pairs or else r is headed by a constructor. Furthermore, $\ell \succ r_j$, for every subterm r_j of j, either because r_j is a subterm of ℓ, or because of a dependency pair $\ell \to r_j$, or because r_j is headed by a constructor, so $\ell \gtrdot r_j$, and r_j's subterms are smaller (by induction).

For reduction in the semantic path ordering to provide termination, we said that one also needs condition (*) to hold. But the dependency pair conditions tell us that $\ell \gtrsim r$ for every rule, and weak monotonicity tells us that $c[\ell] \gtrsim c[r]$ for any context c. Therefore, $s \to t \Rightarrow f(\ldots, s, \ldots) \gtrsim f(\ldots, t, \ldots)$, as required. ◂

It is clear why there is no need to consider dependency pairs $\ell \to u$ when u is a proper subterm of ℓ, as suggested in [9, n. 8], since then $\ell \succ_1 u$. In fact, the pair can be ignored if ℓ has any proper subterm t, such that $t \gtrsim u$, as suggested in [10, Sect. 6.3].

4 The Monotonic Semantic Path Ordering

The dependency-pair conditions for termination also fulfill the requirements for the monotonic semantic path ordering of [3] (preceded by [12]). This method combines a (multiset) semantic path ordering $\succ$ over a well-founded quasi-order $\gtrsim$ with a (weakly-) monotonic quasi-ordering $\geq$. It demands that $\ell \gtrsim r$ and $\ell \succ r$ for each rule, and, furthermore, that the two base orderings satisfy

$$s \geq t \Rightarrow f(\ldots, s, \ldots) \geq f(\ldots, t, \ldots) \qquad (**)$$

See [9, Sect. 4]. (The latter condition is called "quasi-monotonicity" of $\gtrsim$ with respect to $\geq$ in [3] and "harmony" of $\gtrsim$ with $\geq$ in [9, Sect. 3].)

Suppose now that $\geq$ and $\gtrsim$ are one and the same well-founded monotonic quasi-ordering. The above condition (**) translates into weak-monotonicity of $\gtrsim$. Then to satisfy the requirements of the corresponding monotonic semantic path ordering, we have $\ell \gtrsim r$ by the demands of the dependency method and $\ell \succ r$ for the same reasons as in the above proof. See [5, Thm. 7.3.2] and [4, Section 5.4].

5 Conclusion

The ordinary semantic path ordering [14], general path ordering [11, 13], and monotonic semantic path ordering [3] all include recursive cases, where subterms are examined recursively (in some order or other) if two terms are semantically equivalent (vis-à-vis $\doteq$). As noted in [4, Section 5.4], dependency pairs do not make use of the recursive case of the path ordering. Including recursion on subterms refines the simple-minded ordering and can only be of service in termination proofs. (Some might view the absence of recursive comparisons an "advantage", in that the search space is reduced.)

On the other hand, the dependency-pair formulation of this termination method has the practical advantage of rephrasing the task as the termination problem of an enlarged rewrite system (one that includes rewrite rules that force $s \succ t_j$ to hold) for which it may be relatively easy to adapt ordinary termination-proof systems. In one standard version of the method, additional rules—with altered root symbols—are used to disentangle the strong-monotonicity and weak-monotonicity requirements.

It is commonplace with the dependency-pair method for $\gtrdot/\geq$ to be some version of the recursive path ordering [7, 8]. The same is true for the semantic path ordering, which often uses a simpler recursive path ordering for its semantic ordering. This kind of semantic ordering is something that David Plaisted and I used from the earliest days of path orderings.[1]

On account of the *weak* monotonicity requirement for the component ordering $\geq$ used for the semantic path ordering, general path ordering, or dependency-pair method, the ordering $\geq$ in all three cases can ignore selected subterms, which is very often useful.

Nothing we have said relates to the powerful data-flow techniques of [1], which take the narrowing ideas of [6] and others to a high degree of utility. Were one to want to, the analysis of dependency-pair chains could be captured by pattern-based semantic, perhaps akin to that in [15].

Acknowledgement. I thank Jügen Giesl, Jean-Pierre Jouannaud, and Ori Lahav for their advice and comments.

References

1 Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000. Preliminary version available at `http://verify.rwth-aachen.de/giesl/papers/ibn-97-46.ps`.

2 Leo Bachmair and Nachum Dershowitz. Commutation, transformation, and termination. In J. H. Siekmann, editor, *Proceedings of the Eighth International Conference on Automated Deduction (Oxford, England)*, volume 230 of *Lecture Notes in Computer Science*, pages 5–20, Berlin, July 1986. Springer-Verlag. Available at `http://nachum.org/papers/CommutationTermination.pdf`.

3 Cristina Borralleras, Maria Ferreira, and Albert Rubio. Complete monotonic semantic path orderings. In *Proceedings of the 17th International Conference on Automated Deduction (Pittsburgh, PA)*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 346–364, Berlin, June 2000. Springer-Verlag. Available at `http://www.lsi.upc.edu/~albert/papers/mspo.ps.gz`.

[1] This is what was being referred to in the citation of the personal communication "[Plaisted, 1979]" in [8]).

4 Cristina Borralleras and Albert Rubio. Orderings and constraints: Theory and practice of proving termination. In H. Comon-Lundh, C. Kirchner, and H. Kirchner, editors, *Rewriting, Computation and Proof: Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*, volume 4600 of *Lecture Notes in Computer Science*, pages 28–43. Springer-Verlag, Berlin, June 2007. Available at `http://www.lsi.upc.edu/~albert/papers/jean-pierre-60.pdf`.

5 Cristina Borralleras Andreu. *Ordering-Based Methods for Proving Termination Automatically*. PhD thesis, Departament de Llenguatges i Sistemes Informàtics de la Universitat Politècnica de Catalunya, Barcelona, Spain, April 2003. Available at `http://www.lsi.upc.edu/~albert/cristinaphd.ps.gz`.

6 Jacques Chabin and Pierre Réty. Narrowing directed by a graph of terms. In Ronald V. Book, editor, *Rewriting Techniques and Applications*, volume 488 of *Lecture Notes in Computer Science*, pages 112–123. Springer-Verlag, Berlin, 1991. Available at `http://www.univ-orleans.fr/lifo/Members/chabin/articles/rta1991.pdf`.

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

8 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`.

9 Nachum Dershowitz. Termination by abstraction. In *Proceedings of the Twentieth International Conference on Logic Programming (St. Malo, France)*, volume 3132 of *Lecture Notes in Computer Science*, pages 1–18, Berlin, September 2004. Springer-Verlag. Available at `http://nachum.org/papers/TerminationByAbstraction.pdf`.

10 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`.

11 Nachum Dershowitz and Charles Hoot. Natural termination. *Theoretical Computer Science*, 142(2):179–207, 1995. Available at `http://nachum.org/papers/natural-sterm94.pdf`.

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

13 Alfons Geser. An improved general path order. *Applicable Algebra in Engineering, Communication and Computing*, 7(6):469–511, 1996. Available at `http://webdoc.sub.gwdg.de/ebook/e/2001/mip/gpo.ps.Z`.

14 Sam Kamin and Jean-Jacques Lévy. Two generalizations of the recursive path ordering. Unpublished letter to Nachum Dershowitz, Department of Computer Science, University of Illinois, Urbana, IL, February 1980. Available at `http://nachum.org/term/kamin-levy80spo.pdf`.

15 Laurence Puel. Embedding with patterns and associated recursive path ordering. In N. Dershowitz, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications (Chapel Hill, NC)*, volume 387 of *Lecture Notes in Computer Science*, pages 371–387, Berlin, 1989. Springer-Verlag.

Predicative Lexicographic Path Orders: Towards a Maximal Model for Primitive Recursive Functions*

Naohi Eguchi

Institute of Computer Science, University of Innsbruck, Austria
naohi.eguchi@uibk.ac.at

Abstract

The predicative lexicographic path order (PLPO for short), a syntactic restriction of the lexicographic path order, is presented. As well as lexicographic path orders, several non-trivial primitive recursive equations, e.g., primitive recursion with parameter substitution, unnested multiple recursion, or simple nested recursion, can be oriented with PLPOs. It can be shown that PLPOs however only induce primitive recursive upper bounds for derivation lengths of compatible rewrite systems. This yields an alternative proof of a classical fact that the class of primitive recursive functions is closed under these non-trivial primitive recursive equations.

1998 ACM Subject Classification F.4.1, F.3.3

Keywords and phrases Primitive recursive functions, Derivational complexity, Lexicographic path orders, Predicative recursion

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

1 Introduction

As shown by R. Péter [9], the class of primitive recursive functions is closed under a recursion schema that is not an instance of primitive recursion, e.g., primitive recursion with parameter substitution (**PRP**) $f(x+1,y) = h(x,y,f(x,p(x,y)))$, unnested multiple recursion (**UMR**) $f(x+1,y+1) = h(x,y,f(x,p(x,y)),f(x+1,y))$, or simple nested recursion (**SNR**) $f(x+1,y) = h(x,y,f(x,p(x,y,f(x,y))))$. H. Simmons [10] showed Péter's results in a general framework aiming to answer a deep question why primitive recursive functions are closed under these non-trivial primitive recursive equations. As observed by E. A. Cichon and A. Weiermann [6], in order to assess the complexity of a given function, we can discuss about maximal lengths of rewriting sequences, which is known as derivation lengths, in a term rewrite system that defines the function. More precisely, if every derivation length in a given rewrite system $\mathcal{R}$ is bonded by a function in a class $\mathcal{F}$, then the function defined by $\mathcal{R}$ is elementary recursive in $\mathcal{F}$ measured by the size of a starting term. In [2] M. Avanzini and G. Moser have shown that "elementary recursive in" can be replaced by "polynomial time in" if one only considers of rewriting sequences starting with terms whose arguments are already normalised. In [6] alternative proofs of Péter's results were given employing primitive recursive number-theoretic interpretations of rewrite systems corresponding to those non-trivial primitive recursive equations mentioned above. On the other side, any equation of (**PRP**), (**UMR**) and (**SNR**) can be oriented with a termination order known as the lexicographic path order (LPO for short). As shown by Weiermann [11], LPOs induce multiply recursive upper bounds for those derivation lengths. Thus, in order to discuss about (**PRP**), (**UMR**) or (**SNR**), it is natural to restrict LPOs. In [5] Cichon

* This work is supported by JSPS posdoctoral fellowships for young scientists.

introduced the ramified lexicographic path order (RLPO for short), a syntactic restriction of LPO, capturing (**PRP**) and (**UMR**). This work is an attempt to find a *maximal* model for primitive recursive functions based on termination orders in a way different from [5] but stemming from Simmons' approach in [10]. The recursion-theoretic characterisation given in [10] is based on a restrictive (higher order primitive) recursion that is commonly known as *predicative recursion*. A brief explanation about predicative recursion can be found in the paragraph after Example 5 on page 3. Taking the idea of predicative recursion into the lexicographic comparison, we introduce the *predicative lexicographic path order* (PLPO for short), a syntactic restriction of LPO. As well as LPOs, (**PRP**) (**UMR**) and (**SNR**) can be oriented with PLPOs. However, in contrast to LPOs, PLPOs only induce primitive recursive upper bounds for derivation lengths of compatible rewrite systems. This yields an alternative proof of the fact that primitive recursive functions are closed under (**PRP**) (**UMR**) and (**SNR**). The definition of PLPO is also strongly motivated by a more recent work [1] by Avanzini, Moser and the author.

2 Predicative Lexicographic Path Orders

Let $\mathcal{V}$ denote a countably infinite set of variables. A *signature* $\mathcal{F}$ is a finite set of function symbols. The number of argument positions of a function symbol $f \in \mathcal{F}$ is denoted as $\mathsf{arity}(f)$. We write $\mathcal{T}(\mathcal{V},\mathcal{F})$ to denote the set of terms over $\mathcal{V}$ and $\mathcal{F}$. The signature $\mathcal{F}$ can be partitioned into the set $\mathcal{C}$ of *constructors* and the set $\mathcal{D}$ of *defined* symbols. We suppose that $\mathcal{C}$ contains at least one constant. We assume a specific (possibly empty) subset $\mathcal{D}_{\mathsf{lex}}$ of $\mathcal{D}$. A *precedence* $\succcurlyeq_{\mathcal{F}}$ on the signature $\mathcal{F}$ is a quasi-order whose strict part $>_{\mathcal{F}}$ is well-founded on $\mathcal{F}$. We write $f \approx_{\mathcal{F}} g$ if $f \succcurlyeq_{\mathcal{F}} g$ and $g \succcurlyeq_{\mathcal{F}} f$. We also assume that the argument positions of every function symbol are separated into two kinds. The separation is indicated by semicolon as $f(t_1,\dots,t_k;t_{k+1},\dots,t_{k+l})$, where $t_1,\dots,t_k$ are called *normal* arguments whereas $t_{k+1},\dots,t_{k+l}$ are called *safe* ones. The equivalence $\approx_{\mathcal{F}}$ is extended to the term equivalence $\approx$. We write $f(s_1,\dots,s_k;s_{k+1},\dots,s_{k+l}) \approx g(t_1,\dots,t_k;s_{k+1},\dots,t_{k+l})$ if $f \approx_{\mathcal{F}} g$ and $s_j \approx t_j$ for all $j \in \{1,\dots,k+l\}$. An auxiliary relation $s = f(s_1,\dots,s_k;s_{k+1},\dots,s_{k+l}) \sqsupseteq_{\mathsf{plpo}} t$ holds if one of the following cases holds, where $s \sqsupseteq_{\mathsf{plpo}} t$ denotes $s \sqsupset_{\mathsf{plpo}} t$ or $s \approx t$.

1. $f \in \mathcal{C}$ and $s_i \sqsupseteq_{\mathsf{plpo}} t$ for some $i \in \{1,\dots,k+l\}$.
2. $f \in \mathcal{D}$ and $s_i \sqsupseteq_{\mathsf{plpo}} t$ for some $i \in \{1,\dots,k\}$.
3. $f \in \mathcal{D}$, $t = g(t_1,\dots,t_m;t_{m+1},\dots,t_{m+n})$ for some g such that $f >_{\mathcal{F}} g$, and $s \sqsupset_{\mathsf{plpo}} t_j$ for all $j \in \{1,\dots,m+n\}$.

Now we define the *predicative lexicographic path order* (PLPO for short) denoted as $>_{\mathsf{plpo}}$. We write $s \succcurlyeq_{\mathsf{plpo}} t$ if $s >_{\mathsf{plpo}} t$ or $s \approx t$, like the relation $\sqsupseteq_{\mathsf{plpo}}$, write $(s_1,\dots,s_k) \succcurlyeq_{\mathsf{plpo}} (t_1,\dots,t_k)$ if $s_j \succcurlyeq_{\mathsf{plpo}} t_j$ for all $j \in \{1,\dots,k\}$, and we write $(s_1,\dots,s_k) >_{\mathsf{plpo}} (t_1,\dots,t_k)$ if $(s_1,\dots,s_k) \succcurlyeq_{\mathsf{plpo}} (t_1,\dots,t_k)$ and additionally $s_i >_{\mathsf{plpo}} t_i$ for some $i \in \{1,\dots,k\}$.

▶ **Definition 1.** $s = f(s_1,\dots,s_k;s_{k+1},\dots,s_{k+l}) >_{\mathsf{plpo}} t$ holds if one of the following holds.

1. $s \sqsupset_{\mathsf{plpo}} t$.
2. $s_i \succcurlyeq_{\mathsf{plpo}} t$ for some $i \in \{1,\dots,k+l\}$.
3. $f \in \mathcal{D}$, $t = g(t_1,\dots,t_m;t_{m+1},\dots,t_{m+l})$ for some g such that $f >_{\mathcal{F}} g$, $s \sqsupset_{\mathsf{plpo}} t_j$ for all $j \in \{1,\dots,m\}$, and $s >_{\mathsf{plpo}} t_j$ for all $j \in \{m+1,\dots,m+n\}$.
4. $f \in \mathcal{D}\backslash\mathcal{D}_{\mathsf{lex}}$, $t = g(t_1,\dots,t_k;t_{k+1},\dots,t_{k+l})$ for some g such that $f \approx_{\mathcal{F}} g$, $(s_1,\dots,s_k) \succcurlyeq_{\mathsf{plpo}} (t_1,\dots,t_k)$, and $(s_{k+1},\dots,s_{k+l}) >_{\mathsf{plpo}} (t_{k+1},\dots,t_{k+l})$.
5. $f \in \mathcal{D}_{\mathsf{lex}}$, $t = g(t_1,\dots,t_m;t_{m+1},\dots,t_{m+l})$ for some g such that $f \approx_{\mathcal{F}} g$, and there exists $i_0 \in \{1,\dots,\min(k,m)\}$ such that $s_j \approx t_j$ for all $j \in \{1,\dots,i_0-1\}$, $s_{i_0} >_{\mathsf{plpo}} t_{i_0}$, $s \sqsupset_{\mathsf{plpo}} t_j$ for all $j \in \{i_0+1,\dots,m\}$, and $s >_{\mathsf{plpo}} t_j$ for all $j \in \{m+1,\dots,m+n\}$.

By induction according to the definition of $>_{\mathsf{plpo}}$, the inclusion $>_{\mathsf{plpo}} \subseteq >_{\mathsf{lpo}}$ can be shown for the LPO $>_{\mathsf{lpo}}$ induced by the same precedence. The converse inclusion does not hold in general.

▶ **Example 2.** $\mathcal{R}_{\mathsf{PR}} = \{\mathsf{f}(;0,y) \to \mathsf{g}(;y), \mathsf{f}(;\mathsf{s}(;x),y) \to \mathsf{h}(;x,y,\mathsf{f}(;x,y))\}$.

The sets $\mathcal{C}$ and $\mathcal{D}$ are defined by $\mathcal{C} = \{0,\mathsf{s}\}$ and $\mathcal{D} = \{\mathsf{g},\mathsf{h},\mathsf{f}\}$. Let $\mathcal{D}_{\mathsf{lex}} = \emptyset$. Define a precedence $\succcurlyeq_{\mathcal{F}}$ by $\mathsf{f} \approx_{\mathcal{F}} \mathsf{f}$ and $\mathsf{f} >_{\mathcal{F}} \mathsf{g},\mathsf{h}$. Define an argument separation as indicated in the rules. Then $\mathcal{R}_{\mathsf{PR}}$ can be oriented with the PLPO $>_{\mathsf{plpo}}$ induced by $\succcurlyeq_{\mathcal{F}}$. For the first rule $\mathsf{f}(;0,y) >_{\mathsf{plpo}} y$ and hence $\mathsf{f}(;0,y) >_{\mathsf{plpo}} \mathsf{g}(;y)$ by Case 3 in Definition 1. Consider the second rule. Since $(\mathsf{s}(;x),y) >_{\mathsf{plpo}} (x,y)$, $\mathsf{f}(;\mathsf{s}(;x),y) >_{\mathsf{plpo}} \mathsf{f}(;x,y)$ holds as an instance of Case 4. An application of Case 3 allows us to conclude $\mathsf{f}(;\mathsf{s}(;x),y) >_{\mathsf{plpo}} \mathsf{h}(;x,y,\mathsf{f}(;x,y))$.

▶ **Example 3.** $\mathcal{R}_{\mathsf{PRP}} = \{\mathsf{f}(0;y) \to \mathsf{g}(;y), \mathsf{f}(s(;x);y) \to \mathsf{h}(x;y,\mathsf{f}(x;\mathsf{p}(x;y)))\}$.

The sets $\mathcal{C}$ and $\mathcal{D}$ are defined as in the previous example. Define the set $\mathcal{D}_{\mathsf{lex}}$ by $\mathcal{D}_{\mathsf{lex}} = \{\mathsf{f}\}$. Define a precedence $\succcurlyeq_{\mathcal{F}}$ by $\mathsf{f} \approx_{\mathcal{F}} \mathsf{f}$ and $\mathsf{f} >_{\mathcal{F}} \mathsf{q}$ for all $\mathsf{q} \in \{\mathsf{g},\mathsf{p},\mathsf{h}\}$. Define an argument separation as indicated. Then $\mathcal{R}_{\mathsf{PRP}}$ can be oriented with the induced PLPO $>_{\mathsf{plpo}}$. We only consider the most interesting case. Namely we oriente the second rule. Since $\mathsf{s}(;x) \sqsupseteq_{\mathsf{plpo}} x$, $\mathsf{f}(\mathsf{s}(;x);y) \sqsupseteq_{\mathsf{plpo}} x$ holds by the definition of $\sqsupseteq_{\mathsf{plpo}}$. This together with Case 3 yields $\mathsf{f}(\mathsf{s}(;x);y) >_{\mathsf{plpo}} \mathsf{p}(x;y)$. Hence an application of Case 5 yields $\mathsf{f}(\mathsf{s}(;x);y) >_{\mathsf{plpo}} \mathsf{f}(x;\mathsf{p}(x;y))$. Another application of Case 3 allows us to conclude $\mathsf{f}(\mathsf{s}(;x);y) >_{\mathsf{plpo}} \mathsf{h}(x;y,\mathsf{f}(x;\mathsf{p}(x;y)))$.

▶ **Example 4.** $\mathcal{R}_{\mathsf{UMR}} = \left\{ \begin{array}{rcl} \mathsf{f}(0,y;) & \to & \mathsf{g}_0(y;), \\ \mathsf{f}(\mathsf{s}(;x),0;) & \to & \mathsf{g}_1(x;\mathsf{f}(x,\mathsf{q}(x;);)), \\ \mathsf{f}(\mathsf{s}(;x),\mathsf{s}(;y);) & \to & \mathsf{h}(x,y;\mathsf{f}(x,\mathsf{p}(x,y;);),\mathsf{f}(\mathsf{s}(;x),y;)) \end{array} \right\}$.

The sets $\mathcal{C}$ and $\mathcal{D}$ are defined as in the former two examples and the set $\mathcal{D}_{\mathsf{lex}}$ is defined in the previous example. Define a precedence $\succcurlyeq_{\mathcal{F}}$ by $\mathsf{f} \approx_{\mathcal{F}} \mathsf{f}$ and $\mathsf{f} >_{\mathcal{F}} \mathsf{g}$ for all $\mathsf{g} \in \{\mathsf{g}_0,\mathsf{g}_1,\mathsf{p},\mathsf{q},\mathsf{h}\}$. Define an argument separation as indicated. Then $\mathcal{R}_{\mathsf{UMR}}$ can be oriented with the induced PLPO $>_{\mathsf{plpo}}$. Let us consider the most interesting case. Namely we oriente the third rule. Since $\mathsf{f} >_{\mathcal{F}} \mathsf{p}$ and $\mathsf{s}(;u) \sqsupseteq_{\mathsf{plpo}} u$ for each $u \in \{x,y\}$, $\mathsf{f}(\mathsf{s}(;x),\mathsf{s}(;y);) \sqsupseteq_{\mathsf{plpo}} \mathsf{p}(x,y;)$ holds by the definition of $\sqsupseteq_{\mathsf{plpo}}$. Hence, since $\mathsf{s}(;x) >_{\mathsf{plpo}} x$, an application of Case 5 in Definition 1 yields $\mathsf{f}(\mathsf{s}(;x),\mathsf{s}(;x);) >_{\mathsf{plpo}} \mathsf{f}(x,\mathsf{p}(x,y;);)$. Another application of Case 5 yields $\mathsf{f}(\mathsf{s}(;x),\mathsf{s}(;y);) >_{\mathsf{plpo}} \mathsf{f}(\mathsf{s}(;x),y;)$. Clearly $\mathsf{f}(\mathsf{s}(;x),\mathsf{s}(;y);) \sqsupseteq_{\mathsf{plpo}} u$ for each $u \in \{x,y\}$. Hence an application of Case 3 allows us to conclude $\mathsf{f}(\mathsf{s}(;x),\mathsf{s}(;y);) >_{\mathsf{plpo}} \mathsf{h}(x,y;\mathsf{f}(x,\mathsf{p}(x,y;);),\mathsf{f}(\mathsf{s}(;x),y;))$.

▶ **Example 5.** $\mathcal{R}_{\mathsf{SNR}} = \{\mathsf{f}(0;y) \to \mathsf{g}(;y), \mathsf{f}(\mathsf{s}(;x);y) \to \mathsf{h}(x;y,\mathsf{f}(x;\mathsf{p}(x;y,\mathsf{f}(x;y))))\}$.

The sets $\mathcal{C}$, $\mathcal{D}$ and $\mathcal{D}_{\mathsf{lex}}$ are defined as in the former three examples. Define a precedence $\succcurlyeq_{\mathcal{F}}$ as in the previous example. Define an argument separation as indicated. Then $\mathcal{R}_{\mathsf{SNR}}$ can be oriented with the induced PLPO $>_{\mathsf{plpo}}$. We only oriente the second rule. As we observed in the previous example, $\mathsf{f}(\mathsf{s}(;x);y) >_{\mathsf{plpo}} \mathsf{f}(x;y)$ holds by Case 5. Hence $\mathsf{f}(\mathsf{s}(;x);y) >_{\mathsf{plpo}} \mathsf{p}(x;y,\mathsf{f}(x;y))$ holds by Case 3. This together with Case 5 yields $\mathsf{f}(\mathsf{s}(;x);y) >_{\mathsf{plpo}} \mathsf{f}(x;\mathsf{p}(x;y,\mathsf{f}(x;y)))$. Thus another application of Case 3 allows us to conclude $\mathsf{f}(\mathsf{s}(;x);y) >_{\mathsf{plpo}} \mathsf{h}(x;y,\mathsf{f}(x;\mathsf{p}(x;y,\mathsf{f}(x;y))))$.

Careful readers may observe that the general form of nested recursion, e.g., defining equations for the Ackermann function, cannot be oriented with PLPOs. As intended in [4], predicative recursion is a syntactic restriction of the standard (primitive) recursion, where the number of recursive calls is measured only by a normal argument whereas results of recursion are allowed to be substituted only for safe arguments: $f(x+1,\vec{y};\vec{z}) = h(x,\vec{y};\vec{z},f(x,\vec{y};\vec{z}))$. In [10] the meaning of predicative recursion is modified (though [10] is an earlier work than [4]) in such a way that recursive calls are allowed even on safe arguments for the

standard primitive recursion (see Example 2) but still restricted on normal arguments for multiple (nested) recursion (see Example 3–5). In the sequel we present a primitive recursive interpretation for PLPOs. This yields that the maximal length of rewriting sequences in any rewrite system compatible with a PLPO is bounded by a primitive recursive function in the size of the starting term. All the missing details can be found in a technical report [7]. Following [6, page 214], given a natural $d \geq 2$, we define the primitive recursive function F_m by $F_0(x) = d^{x+1}$ and $F_{m+1}(x) = F_m^{d(1+x)}(x)$, where F_m^d denotes the d-fold iteration of F_m.

▶ **Definition 6.** Given k, we inductively define the k-ary primitive recursive function $F_{m,n}$ by $F_{m,0}(x_1, \ldots, x_k) = 0$, $F_{m,n+1}(x_1, \ldots, x_k) = \begin{cases} F_m^{F_{m,n}(x_1,\ldots,x_k)+d(1+x_{n+1})}(\sum_{j=1}^{n+1} x_j) & \text{if } n < k, \\ F_m^{F_{m,n}(x_1,\ldots,x_k)}(\sum_{j=1}^{k} x_j) & \text{if } k \leq n. \end{cases}$

▶ **Definition 7.** Let ℓ be a natural such that $2 \leq \ell$, $\mathcal{F}$ a signature and $\succcurlyeq_{\mathcal{F}}$ a precedence on $\mathcal{F}$. The *rank* $\mathsf{rk} : \mathcal{F} \to \mathbb{N}$ is defined in accordance with $\succcurlyeq_{\mathcal{F}}$, i.e., $\mathsf{rk}(f) \geq \mathsf{rk}(g) \Leftrightarrow f \succcurlyeq_{\mathcal{F}} g$. Define a natural K by $K = \max\{k \mid f \in \mathcal{F} \text{ and } f \text{ has } k \text{ normal argument positions}\}$. Then a primitive recursive interpretation $\mathcal{I} : \mathcal{T}(\mathcal{F}) \to \mathbb{N}$ is defined by $\mathcal{I}(t) = d^{F_{\mathsf{rk}(f)+\ell,K+1}(\mathcal{I}(t_1),\ldots,\mathcal{I}(t_k))}.\ \sum_{j=1}^{l} \mathcal{I}(t_{k+j})$, where $t = f(t_1, \ldots, t_k; t_{k+1}, \ldots, t_{k+l}) \in \mathcal{T}(\mathcal{F})$.

Let $2 \leq \ell$. We define a restriction $\sqsupset_{\mathsf{plpo}}^{\ell}$ of $\sqsupset_{\mathsf{plpo}}$: $s = f(s_1, \ldots, s_k; s_{k+1}, \ldots, s_{k+l}) \sqsupset_{\mathsf{plpo}}^{\ell} t$ holds if one of the following cases holds, where $s \sqsupseteq_{\mathsf{plpo}}^{\ell} t$ denotes $s \sqsupset_{\mathsf{plpo}}^{\ell} t$ or $s \approx t$.

1. $f \in \mathcal{C}$ and $s_i \sqsupseteq_{\mathsf{plpo}}^{\ell} t$ for some $i \in \{1, \ldots, k+l\}$.
2. $f \in \mathcal{D}$ and $s_i \sqsupseteq_{\mathsf{plpo}}^{\ell} t$ for some $i \in \{1, \ldots, k\}$.
3. $f \in \mathcal{D}$, $t = g(t_1, \ldots, t_m; t_{m+1}, \ldots, t_{m+n})$ for some g such that $f >_{\mathcal{F}} g$, and $s \sqsupset_{\mathsf{plpo}}^{\ell-1} t_j$ for all $j \in \{1, \ldots, m+n\}$.

We write $>_{\mathsf{plpo}}^{\ell}$ to denote the PLPO induced by $\sqsupset_{\mathsf{plpo}}^{\ell}$ and $|t|$ to denote the size of a term t. In addition, for a rewrite system $\mathcal{R}$ and a relation $>$, we write $\mathcal{R} \subseteq >$ if $l > r$ holds for every rule $(l \to r) \in \mathcal{R}$.

▶ **Lemma 8.** *Let $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{F})$ be a ground substitution. Suppose $\max(\{\mathsf{arity}(f) \mid f \in \mathcal{F}\} \cup \{\ell \cdot (K+2) + 2\} \cup \{|t| + 1\}) \leq d$. If $s >_{\mathsf{plpo}}^{\ell} t$, then, for the interpretation $\mathcal{I}$ induced by ℓ and d, $\mathcal{I}(s\sigma) > \mathcal{I}(t\sigma)$ holds.*

▶ **Lemma 9.** *Let $s, t \in \mathcal{T}(\mathcal{F})$ be ground terms and $C(\Box) \in \mathcal{T}(\mathcal{F} \cup \{\Box\})$ a (ground) context. If $\mathcal{I}(s) > \mathcal{I}(t)$, then $\mathcal{I}(C(s)) > \mathcal{I}(C(t))$ holds.*

▶ **Theorem 10.** *Let $\mathcal{R}$ be a rewrite system over a signature $\mathcal{F}$ such that $\mathcal{R} \subseteq >_{\mathsf{plpo}}^{\ell}$ for some $\ell \geq 2$ and $s, t \in \mathcal{T}(\mathcal{F})$ be ground terms. Suppose $\max(\{\mathsf{arity}(f) \mid f \in \mathcal{F}\} \cup \{\ell \cdot (K+2)+2\} \cup \{|r| + 1 \mid (l \to r) \in \mathcal{R}\}) \leq d$. If $s \to_{\mathcal{R}} t$, then, for the interpretation induced by ℓ and d, $\mathcal{I}(s) > \mathcal{I}(t)$ holds.*

▶ **Theorem 11.** *For any rewrite system $\mathcal{R}$ such that $\mathcal{R} \subseteq >_{\mathsf{plpo}}$ for some PLPO $>_{\mathsf{plpo}}$, the length of any rewriting sequence in $\mathcal{R}$ starting with a ground term is bounded by a primitive recursive function in the size of the starting term.*

▶ **Corollary 12.** *The class of primitive recursive functions is closed under primitive recursion with parameter substitution, unnested multiple recursion and simple nested recursion.*

3 Concluding remarks

A novel termination order, the predicative lexicographic path order PLPO, was presented. As well as LPOs, any instance of (**PRP**), (**UMR**) and (**SNR**) can be oriented with a PLPO. Note that *general* simple nested recursion briefly discussed in [6, page 221], e.g., simple

nested recursion with more than one recursion parameters, can be even oriented with PLPOs. On the other side, PLPOs only induce primitive recursive upper bounds for derivation lengths of compatible rewrite systems. It turns out that the presented primitive recursive interpretation is not affected even if in Case 4 of Definition 1 one allows permutations of safe argument positions on $\{k+1,\ldots,k+l\}$. Allowance of permutations of normal argument positions is not clear at present. One would recall that, as shown by D. Hofbauer in [8], *multiset path orders* only induce primitive recursive upper bounds for derivation lengths of compatible rewrite systems. Allowance of multiset comparison is not clear in the case even for safe arguments. We mention that every PLPO is a slight extension of an *exponential path order* EPO* defined in [1] though EPO*s only induce exponential (innermost) derivational complexity. An auxiliary relation $\sqsupset_{\mathsf{epo}}$ employed to define EPO* is strictly included in $\sqsupset_{\mathsf{plpo}}$. We also mention that the auxiliary relation $\sqsupset_{\mathsf{plpo}}$ is exactly the same as the relation $>_{\mathsf{pop}}$ introduced in [3] to define the *polynomial path order* POP*. By induction according to the inductive definition of an EPO* $>_{\mathsf{epo}*}$, it can be shown that $>_{\mathsf{epo}*}\subseteq>_{\mathsf{plpo}}$ holds with the same precedence and the same argument separation. In general none of (**PRP**), (**UMR**) and (**SNR**) can be oriented with EPO*s. Perhaps it should be emphasised that a significant difference between PLPO and EPO* lies in Case 4 of Definition 1. Without Case 4 PLPOs would only induce elementary recursive derivational complexity.

References

1 M. Avanzini, N. Eguchi, and G. Moser. A Path Order for Rewrite Systems that Compute Exponential Time Functions. In *Proceedings of 22nd RTA*, volume 10 of *Leibniz International Proceedings in Informatics*, pages 123–138, 2011.

2 M. Avanzini and G. Moser. Closing the Gap Between Runtime Complexity and Polytime Computability. In *Proceedings of 21st RTA*, volume 6 of *Leibniz International Proceedings in Informatics*, pages 33–48, 2010.

3 M. Avanzinii and G. Moser. Complexity Analysis by Rewriting. In *Proceedings of 9th FLOPS 2008*, volume 4989 of *Lecture Notes in Computer Science*, pages 130–146, 2008.

4 S. Bellantoni and S. Cook. A New Recursion-theoretic Characterization of the Polytime Functions. *Computational Complexity*, 2(2):97–110, 1992.

5 E. A. Cichon. Termination Orderings and Complexity Characterizations. In P. Aczel, H. Simmons, and S. S. Wainer, editors, *Proof Theory*, pages 171–193. Cambridge University Press, 1992.

6 E. A. Cichon and A. Weiermann. Term Rewriting Theory for the Primitive Recursive Functions. *Annals of Pure and Applied Logic*, 83(3):199–223, 1997.

7 N. Eguchi. Predicative Lexicographic Path Orders: Towards a Maximal Model for Primitive Recursive Functions. Technical report. Available online at arXiv: 1308.0247 [math.LO].

8 D. Hofbauer. Termination Proofs by Multiset Path Orderings Imply Primitive Recursive Derivation Lengths. *Theoretical Computer Science*, 105(1):129–140, 1992.

9 R. Péter. *Recursive Functions*. Academic Press, New York-London, The 3rd revised edition, Translated from the German, 1967.

10 H. Simmons. The Realm of Primitive Recursion. *Archive for Mathematical Logic*, 27:177–188, 1988.

11 A. Weiermann. Termination Proofs for Term Rewriting Systems by Lexicographic Path Ordering Imply Multiply Recursive Derivation Lengths. *Theoretical Computer Science*, 139(1–2):355–362, 1995.

SAT-Based Loop Detection in Graph Rewriting

Marcus Ermler

University of Bremen, Department of Computer Science
P.O.Box 33 04 40, 28334 Bremen, Germany
`maermler@informatik.uni-bremen.de`

Abstract

In this paper, we propose an approach for detecting loops in derivations of graph rewriting systems via a translation of the derivation process and loop conditions into propositional formulas. A satisfying assignment represents a derivation with a detected loop and so it witnesses that the corresponding graph rewriting system does not terminate.

Keywords and phrases Non-termination, Loops, Graph Rewriting, SAT Encoding

1 Introduction

The question of termination or non-termination of graph rewriting systems seems to be an interesting issue, but is in general undecidable [5]. Nevertheless, techniques to prove termination of graph rewriting systems were introduced (cf. e.g. [4]). Termination of graph rewriting is defined in the following way (cf. [4]).

▶ **Definition 1** (Termination)**.** A graph rewriting system GRS is *terminating*, if it does not admit an infinite derivation.

The question of looping or non-looping has attracted much attention over the last years in the context of string and term rewriting systems, but is not so much studied in the area of graph rewriting. Also, SAT-based approaches are applied to string and term rewriting (cf. e.g. [6]). In this paper, we propose a new approach for detecting loops in derivations of graph rewriting systems via a translation of derivations and loop conditions into propositional formulas. The idea of translating graph rewriting into SAT was introduced in [3], the proposed technique for loop detection is supplemented to this approach. By using a translation to SAT, we want to benefit from fast solving techniques implemented in modern SAT solvers.

2 Graph Rewriting

We use *edge labeled directed graphs without multiple edges* and with a finite node set. For a finite set Σ of labels, such a graph is a pair $G = (V, E)$ where $V = \{1, \ldots, n\} = [n]$ for some $n \in \mathbb{N}$ is a finite set of *nodes*, numbered from 1 to n, and $E \subseteq V \times \Sigma \times V$ is a set of *labeled edges*. n is called the *size* of G. The components V and E are also denoted by V_G and E_G. We call an edge (v, x, v) a *sling* and an edge $(v, *, v')$ *unlabeled* and omit the label $*$ in drawings. Two edges (v_1, x, v_2) and (v_1', x', v_2') are considered as an *undirected* edge if $v_1 = v_2'$ and $v_2 = v_1'$. A special graph is the *empty graph* $\emptyset = (\emptyset, \emptyset)$. We call G a *subgraph* of H, denoted by $G \subseteq H$, if $V_G \subseteq V_H$ and $E_G \subseteq E_H$.

Furthermore, we use injective graph morphisms for the matching. Let G, H be two graphs as defined above. An *injective graph morphism* from $g\colon G \to H$ is an injective mapping $g_V\colon V_G \to V_H$, that is structure- and label-preserving, i.e. for each edge $(v, x, v') \in E_G$, $(g_V(v), x, g_V(v')) \in E_H$. An injective graph morphism $g\colon G \to H$ yields the image $g(G) = (g_V(V_G), g_E(E_G)) \subseteq H$ with $g_E(E_G)) = \{(g_V(v), x, g_V(v')) \mid (v, x, v') \in E_G\}$ called the

match of G in H. In the following, we will write $g(v)$ and $g(e)$ for nodes $v \in V_G$ and edges $e \in E_G$ because the type of the argument indicates the indices V and E.

▶ **Example 2.** Figure 1 shows the example graph G_0. Its nodes are numbered from 1 to 4 and all its edges except the slings are unlabeled and undirected. The slings are labeled with $\varnothing$ to denote that the corresponding nodes have not yet been chosen. The graph on the left-hand of the arrow in Figure 2, called L_{choose} in the following, is a subgraph of G_0. One can choose four injective mappings from L_{choose} to G_0: $g = \{1 \mapsto 1\}$, $g = \{1 \mapsto 2\}$, $g = \{1 \mapsto 3\}$, or $g = \{1 \mapsto 4\}$.

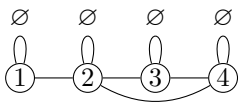


Figure 1 The graph G_0

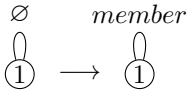


Figure 2 The rule *choose*

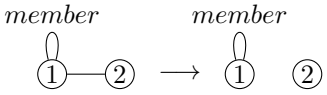


Figure 3 The rule *remEdge*

A *rule* $r = (L \supseteq K \subseteq R)$ consists of three graphs: the *left-hand side* L, the *gluing graph* K, and the *right-hand side* R. In our approach, we only consider rules with $E_K = \emptyset$ and an invariant node set, i.e. $V_L = V_K = V_R$. For that reason, we simplify the rule notation to $r = (L \to R)$, denote the set of nodes of a rule r by V_r and its size by $size(r) = size(L)$.

The application of a rule to a graph works as follows. Let $r = (L \to R)$ be a rule, G a graph, and $g\colon L \to G$ an injective graph morphism. Remove the edges in $g(L)$ from G yielding D and add R disjointly to D. Finally, glue R and D as follows. (1) Merge each $v \in V_R$ with $g(v)$. (2) If there is an edge $(v, x, v') \in E_R$ with $v, v' \in V_R$ and an edge $(g(v), x, g(v')) \in E_D$ then these edges are identified. The application of a rule r to a graph G with respect to an injective graph morphism g yielding a graph H is denoted by $G \underset{r,g}{\Longrightarrow} H$. This is called *rule application* or *direct derivation* and fits into the *double-pushout approach* (cf. [1]). The sequential composition $d = G_0 \underset{r_1,g_1}{\Longrightarrow} G_1 \underset{r_2,g_2}{\Longrightarrow} \cdots \underset{r_n,g_n}{\Longrightarrow} G_n$ of n direct derivations for some $n \in \mathbb{N}$ is called a *derivation*, shortly denoted by $G_0 \overset{*}{\underset{P}{\Longrightarrow}} G_n$ for $n \geq 0$ and $G_0 \overset{+}{\underset{P}{\Longrightarrow}} G_n$ for $n > 0$ if $r_1, \ldots, r_n \in P$.

▶ **Example 3.** In Figure 2, one can find an example for a graph rewriting rule which adds a *member* sling to a node. As described above, there are four mappings from L_{choose} to G_0. In Figure 4 a derivation is shown. The first graph G_1 of the derivation is the result of applying *choose* to G_0 twice by using the mappings $g = \{1 \mapsto 2\}$ and, then, $g = \{1 \mapsto 4\}$. Applying the rule *remEdge* from Figure 3 with the mapping $g = \{1 \mapsto 4, 2 \mapsto 3\}$ results in the second graph. Finally, a three time application of the rule *remEdge* yields the last graph.

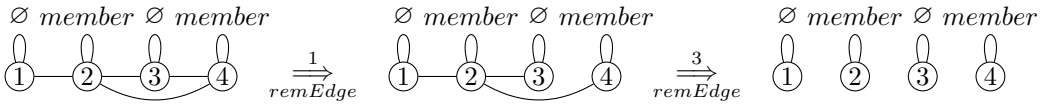


Figure 4 A sample derivation

A *graph grammar* is a system $GG = (G_0, P, \Delta)$ consisting of an *initial graph* G_0, a finite set P of graph rewriting rules, and a set $\Delta \subseteq \Sigma$ of *terminal symbols*. The graph grammar GG specifies all derivations from the initial graph G_0 to graphs labeled over Δ.

▶ **Example 4.** As an example, we consider the *vertex cover problem* that refers to the question, whether for a graph $G = (V, E)$, a subset $X \subseteq V$ exists, such that for all $(v, x, v') \in E$, $v \in X$ or $v' \in X$. The corresponding graph grammar for the graph G_0 from Figure 1 is $VC = (G_0, \{choose, remEdge\}, \{member, \varnothing\})$. The derivation in Figure 4 states the last part of a possible computation with G_0 as input. The two-time application of *choose* yields the first graph of the derivation as detailed above. A vertex cover is found, because the last graph of the derivation has only edges labeled with *member* or $\varnothing$.

3 Detecting Loops in Graph Rewriting via SAT

Every propositional formula with variable set $\{edge(v, a, v', k) \mid (v, a, v') \in [n] \times \Sigma \times [n], k \in [m]\}$ represents a sequence $G_1, \ldots, G_m$ of graphs for each variable assignment f satisfying the formula, i.e. the graph G_k contains the edge (v, a, v') if and only if $f(edge(v, a, v', k)) = TRUE$. Please note, that this translation does not allow node addition or deletion. A single initial graph G in the kth derivation step can be described by the formula

$$\mathsf{graph(G)(k)} = \bigwedge_{(v,a,v') \in E_G} edge(v, a, v', k) \wedge \bigwedge_{(v,a,v') \in ([n] \times \Sigma \times [n]) - E_G} \neg edge(v, a, v', k).$$

▶ **Example 5.** Then, the graph G_0 in Figure 1 is expressed via the following formula

$$\mathsf{graph(G_0)(0)} = \bigwedge_{(v,a,v') \in E_0} edge(v, a, v', 0) \wedge \bigwedge_{(v,a,v') \in ([n] \times \Sigma \times [n]) - E_0} \neg edge(v, a, v', 0)$$

where $E_0 = \{(1, *, 2), (2, *, 1), (2, *, 3), (3, *, 2), (3, *, 4), (4, *, 3), (2, *, 4), (4, *, 2), (1, \varnothing, 1), (2, \varnothing, 2), (3, \varnothing, 3), (4, \varnothing, 4)\}$. Please note, that in case of undirected edges both corresponding directed edges have to occur in the formula.

For a rule $r = (L \to R)$ the set of injective graph morphisms from $[size(r)]$ to the set of nodes $[n]$ is denoted by $\mathcal{M}(r, n)$. Let $k \in \mathbb{N}$ be a derivation step, G_{k-1} be a graph with the node set $[n]$, $r = (L \to R)$ be a rule, and $g \in \mathcal{M}(r, n)$. The application of r to G_{k-1} with respect to g is then expressed by the following formulas

- $\mathsf{morph(r, g, k)} = \bigwedge_{(v,a,v') \in E_L} edge(g(v), a, g(v'), k - 1)$,
- $\mathsf{rem(r, g, k)} = \bigwedge_{(v,a,v') \in E_L - E_R} \neg edge(g(v), a, g(v'), k)$,
- $\mathsf{add(r, g, k)} = \bigwedge_{(v,a,v') \in E_R} edge(g(v), a, g(v'), k)$,
- $\mathsf{keep(r, g, k)} = \bigwedge_{(v,a,v') \notin g(E_L \cup E_R)} \big(edge(v, a, v', k - 1) \leftrightarrow edge(v, a, v', k)\big)$ where $g(E_L \cup E_R) = \{(g(v), a, g(v')) \mid (v, a, v'), \in E_L \cup E_R\}$,
- $\mathsf{apply(r, g, k)} = \mathsf{morph(r, g, k)} \wedge \mathsf{rem(r, g, k)} \wedge \mathsf{add(r, g, k)} \wedge \mathsf{keep(r, g, k)}$.

The formula morph describes that g is a graph morphism from L to G_{k-1}. The removal of the images of every edge of the left-hand side L from G_{k-1} is expressed by rem. The addition of edges of the right-hand side R is described by add. That edges that have been neither deleted nor added must be kept, corresponds to the formula keep. Finally, apply describes the whole application of r to G_{k-1} with respect to g. The following theorem states that a satisfying assignment to apply corresponds to a direct derivation.

▶ **Theorem 6.** *$G_{k-1} \underset{r,g}{\Longrightarrow} G_k$ if and only if there is a satisfying assignment to the formula* $\mathsf{graph(G_{k-1})(k-1)} \wedge \mathsf{apply(r, g, k)} \wedge \mathsf{graph(G_k)(k)}$.

▶ **Example 7.** For the rule *remEdge* in Figure 3, the first graph G_1 in Figure 4, and the graph morphism $g = \{1 \mapsto 4, 2 \mapsto 3\}$ one gets the following formulas: $\mathsf{morph(remEdge, g, 2)} = edge(4, member, 4, 1) \wedge edge(3, *, 4, 1) \wedge edge(4, *, 3, 1)$, $\mathsf{rem(remEdge, g, 2)} = \neg edge(3, *, 4, 2) \wedge \neg edge(4, *, 3, 2)$, and $\mathsf{add(remEdge, g, 2)} = edge(4, member, 4, 2)$.

For each rule $r \in P$ and each mapping $g \in \mathcal{M}(r,n)$, the possible applications of r to G_{k-1} are expressed via $\mathsf{step(k)} = \bigvee_{r\in P, g\in M(r,n)} \mathsf{apply(r,g,k)}$. All derivation starting in G_0 of length m are described by $\mathsf{der(G_0,m)} = \mathsf{graph(G_0)(0)} \wedge \bigwedge_{k=1}^{m} \mathsf{step(k)}$. The following theorem states the connection between der and successful derivations.

▶ **Theorem 8.** *Let G_0 and G_T be two graphs of size n, P be a rule set, and $m \in \mathbb{N}$ be the number of derivation steps. Then there is a successful derivation $G_0 \overset{m}{\underset{P}{\Longrightarrow}} G_T$ if and only if there is a satisfying assignment to* $\mathsf{der(G_0,m)} \wedge \mathsf{graph(G_T)(m)}$.

Let n be the size of G_0 and $p(n)$ be a polynomial bound. Then all derivations starting in G_0 with a length of at most $p(n)$ are expressed via $\mathsf{all\_der(G_0,p(n))} = \bigvee_{m=0}^{p(n)} \mathsf{der(G_0,m)}$.

▶ **Example 9.** A satisfying assignment to the following subformula yields the derivation detailed in Example 3 where c is an abbreviation for choose and rE for remEdge
$\mathsf{graph(G_0)(0)} \wedge \mathsf{apply(c,\{1\mapsto 2\},1)} \wedge \mathsf{apply(c,\{1\mapsto 4\},2)} \wedge \mathsf{apply(rE,\{1\mapsto 4, 2\mapsto 3\},3)} \wedge$
$\mathsf{apply(rE,\{1\mapsto 2, 2\mapsto 1\},4)} \wedge \mathsf{apply(rE,\{1\mapsto 2, 2\mapsto 3\},5)} \wedge \mathsf{apply(rE,\{1\mapsto 4, 2\mapsto 2\},6)}$.

Let G_0 and H_0 be two graphs and $g\colon G_0 \to H_0$ be an injective graph morphism. Then the JOIN-Theorem in ([2], p. 101) states the conditions under which a derivation $G_0 \overset{n}{\Longrightarrow} G_n$ can be extended to a derivation $H_0 \overset{n}{\Longrightarrow} H_n$. Loosely speaking, $H_0 - G_0$ is joined with each G_i. This result can be used to find derivations of the form $G_0 \overset{n}{\Longrightarrow} G_n \Longrightarrow H_0 \supsetneq G_0$, i.e. to detect loops in derivations. According to the JOIN-Theorem, the derivation $G_0 \overset{n}{\Longrightarrow} G_n$ would be extended to a derivation $H_0 \overset{n}{\Longrightarrow} H_n$. Important here is that nodes connecting $g(G_0)$ and $H_0 - g(G_0)$ are not deleted during the derivation. In this paper, we only consider rules that do not delete nodes but this is still a special case. Thus, we use the term *simple looping* instead of looping and define it as follows.

▶ **Definition 10** (Simple Loops)**.** A graph rewriting system with an initial graph G_0 and a rule set P containing no node deletion rules is called *simple looping*, if there are graphs G, H and an injective graph morphism $g\colon G \to H$ such that $G_0 \overset{*}{\underset{P}{\Longrightarrow}} G \overset{+}{\underset{P}{\Longrightarrow}} H$.

▶ **Example 11.** Let us consider again Example 4 and let us assume that *choose* has been defined in a wrong way (see Figure 5). The application of $\overline{choose}$ to G_0 yields the last graph in Figure 6 where G_0 is isomorphic to a subgraph of it, i.e. the derivation contains a simple loop. The rule $\overline{choose}$ can be applied infinitely often to node 2.

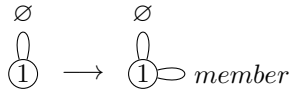


Figure 5 The altered rule $\overline{choose}$

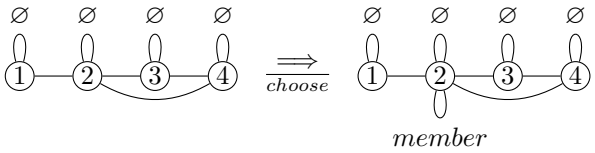


Figure 6 A detected loop

For some graph in a derivation of length m isomorphic to a subgraph of the last graph of this derivation, we define the following propositional formula

$$\mathsf{loop(m)} = \bigvee_{k=1}^{m-1} \bigvee_{g\in\mathcal{M}(n,n)} \bigwedge_{(v,a,v')\in[n]\times\Sigma\times[n]} \Big(edge(v,a,v',k) \to edge(g(v),a,g(v'),m) \Big).$$

Please note, that this formula generates up to n^n possible morphisms ($g \in \mathcal{M}(n,n)$) and has to be restricted in some way. An idea could be to use node types to reduce the possible

matchings. In the derivation from Example 3, we could use the *member*- and ∅-slings as node types such that *member*-nodes could only match *member*-nodes (the same for ∅-nodes).

Detecting a simple loop in all derivations up to a length of $p(n)$ is defined as follows

$$\mathsf{loop\_detection}(\mathsf{G_0}, \mathsf{p(n)}) = \bigvee_{m=0}^{p(n)} \big(\mathsf{der}(\mathsf{G_0}, \mathsf{m}) \wedge \mathsf{loop}(\mathsf{m})\big).$$

Simple looping corresponds to a satisfying assignment for loop_detection.

▶ **Theorem 12.** *Let GRS be a graph rewriting system with an initial graph G_0 of size n and P be a rule set containing no node deletion rules. Then GRS is simple looping if and only if there is a polynomial $p(n)$ such that there is a satisfying assignment to* $\mathsf{loop\_detection}(\mathsf{G_0}, \mathsf{p(n)})$.

▶ **Example 13.** A satisfying assignment to the subformula $\mathsf{der}(\mathsf{G_0}, 1) \wedge \mathsf{loop}(1)$ would yield the detected simple loop from Example 11.

4 Conclusion

In this paper, we have introduced a SAT-based approach for detecting simple loops in derivations of graph rewriting systems. In future, we want to investigate looping for subgraphs, i.e. we want to find derivations $G_0 \overset{*}{\Longrightarrow} G \overset{+}{\Longrightarrow} H$ where a subgraph $\overline{G}$ of G is isomorphic to a subgraph of H. This idea would cover simple looping. Moreover, we want to devise a translation to SAT for graph rewriting systems with node deletion rules.

Acknowledgements The author is grateful to Johannes Waldmann for his suggestion that the SAT-based approach to graph rewriting could be used for loop detection, to Hans-Jörg Kreowski for his suggestion that the JOIN-Theorem could be useful for Definition 10, and to the anonymous referees for their helpful comments.

References

1 Andrea Corradini, Hartmut Ehrig, Reiko Heckel, Michael Löwe, Ugo Montanari, and Francesca Rossi. Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*, pages 163–245. World Scientific, 1997.

2 Hans-Jörg Kreowski. *Manipulationen von Graphmanipulationen.* PhD thesis, TU Berlin, 1978.

3 Hans-Jörg Kreowski, Sabine Kuske, and Robert Wille. Graph transformation units guided by a SAT solver. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *Proc. 5th Intl. Conference on Graph Transformations (ICGT 2010)*, volume 6372 of *Lecture Notes in Computer Science*, pages 27–42. Springer, 2010.

4 Detlef Plump. On termination of graph rewriting. In Manfred Nagl, editor, *Proc. Graph-Theoretic Concepts in Computer Science*, volume 1017 of *Lecture Notes in Computer Science*, pages 88–100, 1995.

5 Detlef Plump. Termination of graph rewriting is undecidable. *Fundamenta Informaticae*, 33(2):201–209, 1998.

6 Harald Zankl, Christian Sternagel, Dieter Hofbauer, and Aart Middeldorp. Finding and certifying loops. In Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorný, and Bernhard Rumpe, editors, *Proc. 36th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2010)*, volume 5901 of *Lecture Notes in Computer Science*, pages 755–766. Springer, 2010.

Towards Generic Inductive Constructions in Systems of Nets

Stéphane Gimenez

Institute of Computer Science
University of Innsbruck, Austria
`stephane.gimenez@uibk.ac.at`

Abstract

As an alternative to recursion [2] or cycles [6], which grant Turing-completeness, we sketch a system of nets that is sufficiently expressive to manipulate complex inductive and co-inductive objects, but enforces termination at the same time. We aim at a logically sound framework, close to prior fixed-point logics [1], that can be computationally implemented as a graph-based rewriting system in the style of proof nets. A complete and fully satisfactory formalization might require a refined approach, based on systems of interaction nets that are enriched with a deep-inference typing system as presented in [3].

1998 ACM Subject Classification F.1.2, F.4.1

Keywords and phrases Interaction Nets, Proof Nets, Linear Logic, Fixed Points, Inductive and Co-Inductive Types.

1 Introduction

Inductive types include typically integers, list, trees, or more complex finite objects, possibly simply-typed λ-terms. Co-inductive types typically represent infinite data-structures like streams, however some infinite co-inductive terms also admit finite representations.

For instance, lists of elements of type A are either the empty list or a construct made of an element of type A and another list. An unfolded definition of their type could be written, using some standard functional programming syntax, as follows:

```
Nil | Cons of A * (Nil | Cons of A * (Nil | Cons of A * (...)))
```

With linear logic connectives it writes $1 \oplus A \otimes (1 \oplus A \otimes (1 \oplus A \otimes (\dots)))$, which can also be represented finitely with a μ abstraction: $\mu\alpha.\, 1 \oplus A \otimes \alpha$.

Using instead a ν abstraction, the dual connector, type $\nu\alpha.\, 1 \oplus A \otimes \alpha$ denotes streams, i. e. possibly infinite sequences of elements of A. In particular, streams of natural numbers, for instance, include recurrent sequences such as $[1, 2, 1, 2, 1, 2, \dots[$, or slightly more complex sequences like $[1, 1, 2, 1, 2, 3, 1, 2, 3, 4 \dots[$, which can be defined finitely in most programming languages.

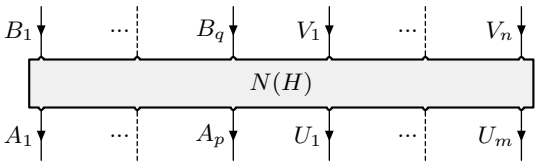
A few specific inductive types like integers, lists or trees are easily represented in systems of nets like interaction nets [5], but co-inductive types are not commonly found. We describe in this paper a generic method to implement any inductive or co-inductive type, from the multiplicative, additive and exponential constructions provided by linear logic. We will moreover show that the exponential constructions are not strictly required.

2 Unfolding Boxes

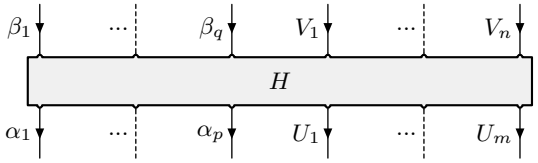
Given the usual system of proof nets for MELL [4], we consider sets $\mathcal{N}_\Gamma\,[P_1 : \Gamma_1, \dots, P_k : \Gamma_k]$ of nets of interface Γ containing open emplacements (also called net variables) P_i of interface

Γ_i. These are called *open nets*. The usual principles behind net reduction are unchanged; open emplacements just do not actively take part in the interaction process.

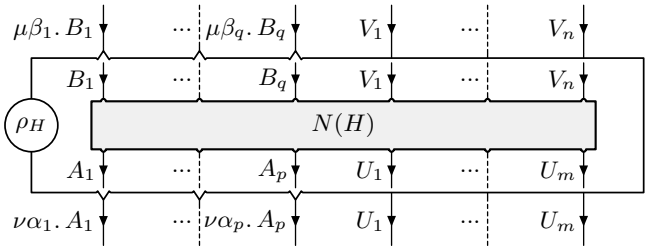
Unfolding boxes (a finite description of a co-inductive object). Let $N(H)$ be an open net with interface:



whose open emplacement H is typed with atomic type variables $\alpha_1, \ldots, \alpha_p$ and $\beta_1, \ldots, \beta_q$ as follows:



Such a net can be enclosed within an *unfolding box* as described hereafter. The open emplacement H and type variables α_i and β_i, which are expected to appear in their respective A_i or B_i and are assumed not to appear in any of the V_i or the U_i, are bound in the process.



From a logical viewpoint, the content of an unfolding box ρ_H is the body of an inductive proof in which the open emplacement H is used as induction hypothesis. Ports typed $\nu\alpha_i.\, A_i$ or $\mu\beta_i.\, B_i$ are principal ports, and ports typed V_i or U_i are auxiliary ports which do not interact immediately.

Moreover, two atomic variables among the α_i or the β_i can be chosen equal when their respective A_i or B_i (i. e. the types they mask) are equal.

Reduction of unfolding boxes Any logical operator which interacts through a principal port of an unfolding box will open, or "unfold", this box. Type-wise, type variables α_i and β_i are not made free, but are respectively substituted with type $\nu\alpha_i.\, A_i$ or type $\mu\beta_i.\, B_i$. Bounded open emplacements which materialize inductive calls are substituted with copies of the full original unfolding box.

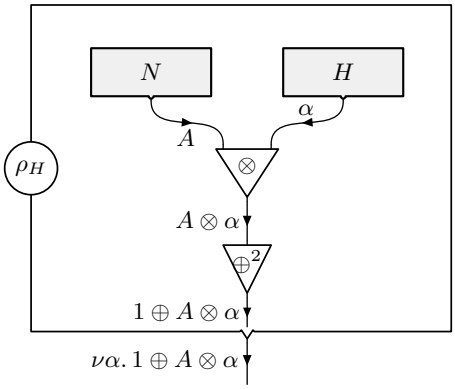
The typing system ensures that recursive calls are made deep inside the inductive structure of the interacting object and prevents non-termination.

Self-reduction Open emplacements may anytime be substituted with copies of the contents of the whole box which binds them. It optimizes later reductions of the net and could be triggered once after each normal unfolding step (again, to avoid non-termination).

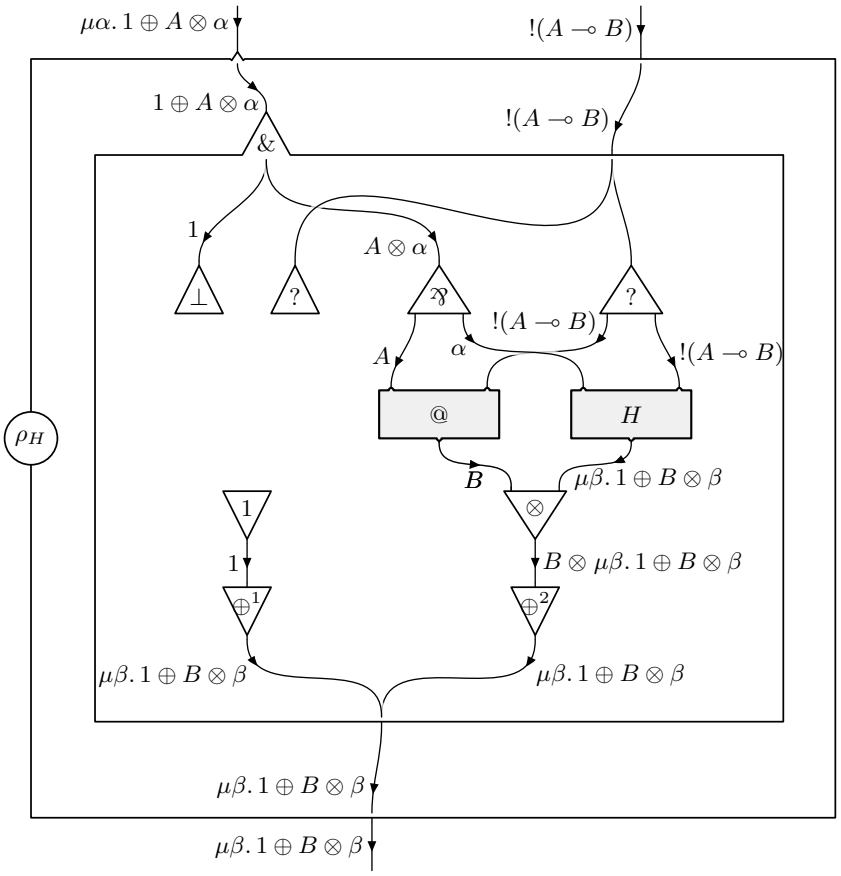
3 Examples

Our system allows one to write any operation on inductive or co-inductive data types, like usual integer addition, multiplication, exponentiation, etc. or list concatenation, mapping, folding, flattening, etc. or stream intermixing, mapping, etc.

First, we provide the very basic example of a net producing (on demand) an infinite stream $\nu\alpha.\, 1 \oplus A \otimes \alpha$ of a given element of type A produced by a net N.



Follows another example, which takes as parameters a promoted function $!(A \multimap B)$, that maps elements of type A to elements of type B, and a list $\mu\alpha.\, 1 \oplus A \otimes \alpha$ with elements of the former type, and returns a list $\mu\beta.\, 1 \oplus B \otimes \beta$ with elements the latter type:

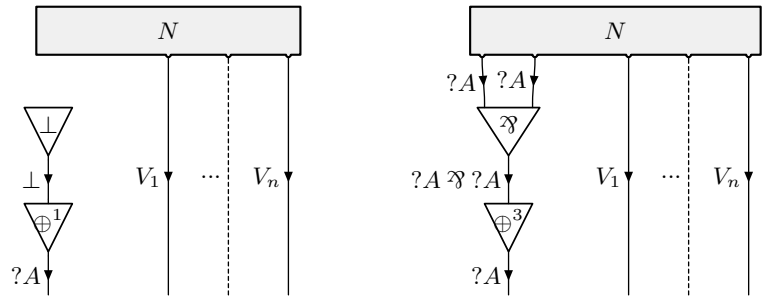


This unfolding box will unfold as soon as the top-level constructor of the $\mu\alpha.\, 1 \oplus A \otimes \alpha$ list will interact through its principal port. This constructor will then be matched against the additive box. Only one of the two additive slices will be kept. If the input list is not empty, the first element is extracted and a copy of the function is applied to it by the net labeled @. In parallel, an inductive call is made to deal with the tail of the list. Resulting objects are combined to produce the output list. When the input list is eventually depleted, notice that no inductive call is made, since the filled H emplacement is garbage-collected with the entire right slice of the additive box.

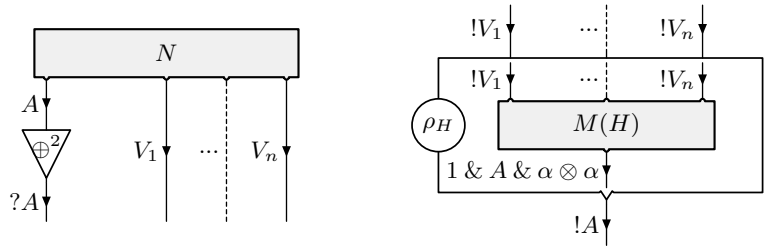
4 A Co-Inductively Defined Exponential

Exponential constructions can be encoded with the provided co-induction scheme, using type $!A := \nu\alpha.\, 1 \,\&\, A \,\&\, \alpha \otimes \alpha$, whose relevance was already mentioned in [4]. The implementation of *weakening*, *dereliction*, *contraction* and *promotion* rules is provided below. Obviously, this definition of the exponential is not free. In particular, would we not restrict generation of $!A$ to those four constructions only, we could build non-uniform exponentials.

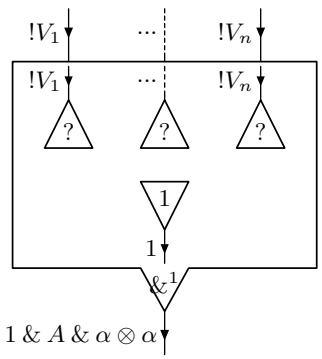
Rules *weakening* and *contraction* write:

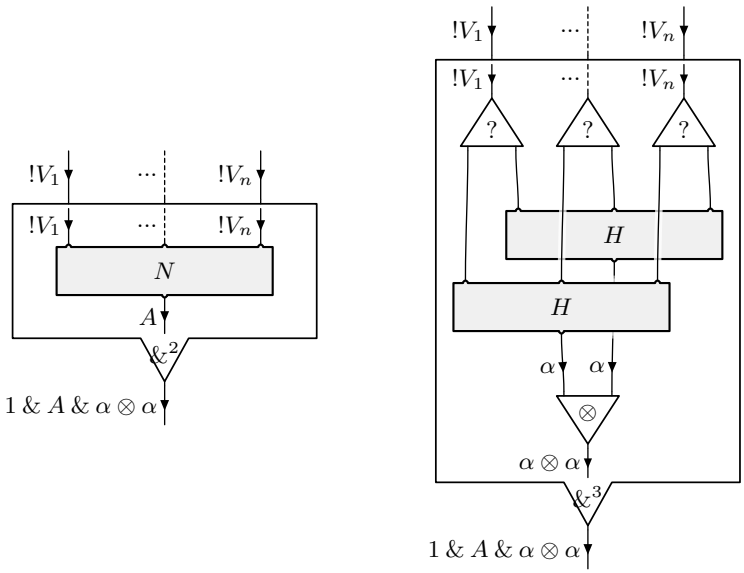


Rules *dereliction* and *promotion* of a net N write:



where $M(H)$ denotes the net obtained by superposing the following additive-box slices. Operators *weakening* and *contraction* used in those slices have already been defined.





Main reductions of exponential constructions are simulated by unfoldings followed by standard reduction steps. As an aside, additive commutation steps are expensive (except in the case of a fully-parallel reduction strategy) and are usually disabled. It is then harmless to allow interaction on upper ports of unfolding boxes that are typed with an exponential type, so that exponential commutations are made available as well.

5 Conclusion

We presented inductive types and sketched a generic method to handle them in the proof net formalism, as an extension of MELL. The exponential fragment can in fact automatically be obtained trough an encoding, although a native handling would certainly be more efficient.

We provided examples that make proper use of the unfolding boxes, but a correctness criterion was not discussed. Deep-inference typing systems such as the one presented in [3] might offer a good framework to properly structure an unfolding construction. In forthcoming works, we aim to extend their reducibility-based strong normalization theorem to inductive and co-inductive types.

References

1 D. Baelde. Least and greatest fixed points in linear logic. *CoRR*, abs/0910.3383, 2009.
2 S. Gimenez. *Programmer, calculer et raisonner avec les réseaux de la logique linéaire*. PhD thesis, Université Paris Diderot, 2009.
3 S. Gimenez and G. Moser. The structure of interaction. In *Computer Science Logic (CSL '13)*, 2013.
4 J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
5 Y. Lafont. Interaction nets. *Principles of Programming Languages (POPL '90)*, pages 95–108, 1990.
6 R. Montelatici. Polarized proof nets with cycles and fixpoints semantics. In *Typed Lambda Calculi and Applications*, volume 2701 of *Lecture Notes in Computer Science*, pages 256–270. Springer, 2003.

Synthesizing Matrix Interpretations via Backward Completion*

Dieter Hofbauer

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

Abstract

Various approaches to automatically synthesizing termination proofs via matrix interpretations are used in state-of-the-art provers, most notably those based on satisfiability solving. In this talk, an alternative idea for the particular case of string rewriting is presented, exploiting the view of matrix interpretations as weighted automata. A demo of a prototype implementation will show the utility of this approach, as well as its limitations.

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

Keywords and phrases Termination, string rewriting, matrix interpretations

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

1 Introduction

Matrix interpretations for string rewriting interpret the free monoid of strings in a ring structure, where concatenation of factors corresponds to multiplication and replacement of factors corresponds to subtraction. For termination, we use an (infinite) ordered ring, which is well-founded on its *positive cone*, see [1].

▶ **Example 1.** In $(\mathbb{Z}, 0, 1, +, \cdot)$ we can prove termination of $\{aba \to aa\}$ by the interpretation $i : a \mapsto 1, b \mapsto 2$ as $i(aba \to aa) = i(aba) - i(aa) = i(a) \cdot i(b) \cdot i(a) - i(a) \cdot i(a) = 1 > 0$, but neither $\{ab \to ba\}$ nor $\{aa \to aba\}$ can be proven terminating in $\mathbb{Z}$ as multiplication is commutative and due to the totality of the ordering employed, respectively.

For that reason, we use non-commutative rings with a non-total ordering, in this case rings of square matrices over the natural numbers; for definitions and notations we refer to [1].

▶ **Example 2.** For proving termination of $\{ab \to ba\}$ consider the E_2-interpretation $i : a \mapsto \left(\begin{smallmatrix}1&1\\0&1\end{smallmatrix}\right), b \mapsto \left(\begin{smallmatrix}1&0\\0&2\end{smallmatrix}\right)$, where $i(ab \to ba) = i(ab) - i(ba) = i(a) \cdot i(b) - i(b) \cdot i(a) = \left(\begin{smallmatrix}1&2\\0&2\end{smallmatrix}\right) - \left(\begin{smallmatrix}1&1\\0&2\end{smallmatrix}\right) = \left(\begin{smallmatrix}0&1\\0&0\end{smallmatrix}\right) \in P_2$, and for proving termination of $\{aa \to aba\}$ the E_1-interpretation $i : a \mapsto \left(\begin{smallmatrix}1&1\\1&0\end{smallmatrix}\right), b \mapsto \left(\begin{smallmatrix}1&0\\0&0\end{smallmatrix}\right)$ gives $i(aa \to aba) = i(aa) - i(aba) = \left(\begin{smallmatrix}2&1\\1&1\end{smallmatrix}\right) - \left(\begin{smallmatrix}1&1\\1&1\end{smallmatrix}\right) = \left(\begin{smallmatrix}1&0\\0&0\end{smallmatrix}\right) \in P_1$.

Various approaches for synthesizing matrix interpretations have been proposed, for instance complete enumeration of restricted matrix shapes, random guesses for small matrix dimensions, evolutionary programming, and, most prominently, constraint solving. In this talk, we present *backward completion* as another approach for the same purpose. This is related to *forward completion* procedures for match-bound termination proofs as in [2].

* This work was partially supported by a travel grant from the Japan Advanced Institute of Science and Technology (JAIST).

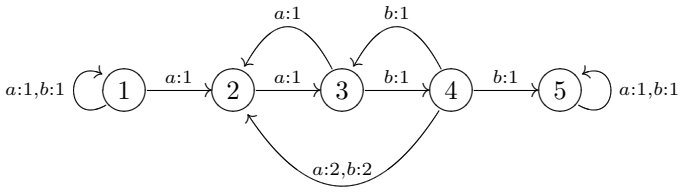
2 Matrix interpretations as weighted automata

Matrix interpretations correspond to *weighted automata* in a direct way. A *weighted automaton* is a mapping weight : $Q \times \Sigma \times Q \to \mathbb{N}$, where Q is a finite set (of *states*). This mapping is extended to weight : $Q \times \Sigma^* \times Q \to \mathbb{N}$ by multiplying weights along a single path and adding weights of different paths. A transition from state p to state q with weight n for letter a corresponds to $i(a)_{p,q} = n$ in a matrix interpretation, and vice versa.

▶ **Example 3.** The matrix interpretation

$$a \mapsto \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \qquad b \mapsto \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 2 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

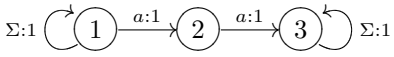
of type $E_{\{1,5\}}$ corresponds to the following weighted automaton, proving termination of $\{aabb \to bbbaaa\}$ (TPDB problem `z001`, see [3], known as *Zantema's problem*):



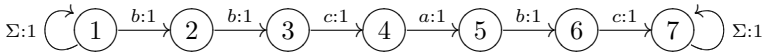
3 Backward completion

Often, automata of a particularly simple form can prove termination of string rewriting systems. These *straight-line automata* essentially consist of a single path that corresponds to the left-hand side of a rule, where each transition has weight 1.

▶ **Example 4.** For $\{aa \to aba\}$, the following automaton proves termination:



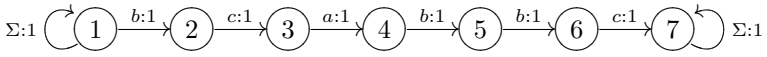
▶ **Example 5.** In the same way, for $\{bbcabc \to abbcbca\}$ (TPDB problem z061) the automaton



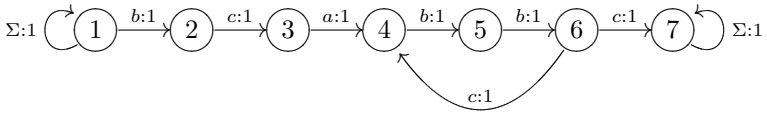
serves as a termination certificate.

Of course, straight-line automata fail in many cases.

▶ **Example 6.** Consider $\{bcabbc \to abcbbca\}$ (TPDB problem z062). For the automaton



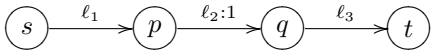
we get $\text{weight}(1, bcabbc, 4) = 0 \not\geq 1 = \text{weight}(1, abcbbca, 4)$, and the termination proof fails. However, if we compensate for that defect by adding an edge



we are done: $\text{weight}(1, bcabbc, 4) = 1 = \text{weight}(1, abcbbca, 4)$.

This motivates *backward completion*. Define a pair of states (s, t) with $\text{weight}(s, \ell, t) < \text{weight}(s, r, t)$ to be a *defect* for the rewriting rule $\ell \to r$. Compensate for such a defect by adding a path for ℓ from s to t. In general, for a *backward completion step* choose a defect (s, t) for some rule $\ell \to r$ and a factorization $\ell = \ell_1\ell_2\ell_3$ so that, for some states p, q,

and add a path of weight 1 from p to q for ℓ_2:

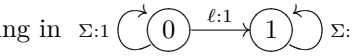


4 Implementation

One possiblity is to implement backward completion as a probabilistic algorithm, as follows.

(1) Choose as start automaton 

with just two states.

(2) Randomly choose the left-hand side ℓ of a rule $\ell \to r$ and add a path from 0 to 1 for ℓ, resulting in 

. Note that now $\text{weight}(0, \ell, 1) > \text{weight}(0, r, 1)$ unless ℓ is a factor of r, in which case the rewriting system is already non-terminating.

(3) As long as defects exist and $\text{weight}(0, \ell, 1) > \text{weight}(0, r, 1)$ keeps holding true, randomly perform some backward completion step.

This might fail or go on forever, but in case the procedure stops successfully, the resulting automaton provides a termination certificate.

▶ **Example 7.** The automaton in example 3 can easily be synthesized in this way. Futher examples include rewriting systems from the TPDB that haven't been solved during the 2013 termination competition (see `termination-portal.org/wiki/Termination_Competition/`).

5 Extensions

In the talk, we present various variants of this strategy, in particular a non-probabilistic variant where completion steps are enumerated under a breadth-first strategy. Building on results from [4], a specialized strategy aims at automatically proving polynomial upper bounds on derivation lengths. Backward completion can be easily adapted to the setting of relative termination proofs, thereby considerably strengthening its applicability. A demo implemention shows the utility of the approach, as well as its limitations.

References

1 D. Hofbauer, J. Waldmann. *Termination of string rewriting with matrix interpretations.* Proc. 17th Int. Conf. on Rewriting Techniques and Applications (RTA), pp. 328–342, 2006.

2 A. Geser, D. Hofbauer, J. Waldmann, H. Zantema. *Finding finite automata that certify termination of string rewriting.* Int. J. Found. Comput. Sci. 16(3):471–486, 2005.

3 The Termination Problems Data Base, `termination-portal.org/wiki/TPDB/`.

4 J. Waldmann. *Polynomially bounded matrix interpretations.* Proc. 21st Int. Conf. on Rewriting Techniques and Applications (RTA), pp. 357–372, 2010.

Termination of LCTRSs*

Cynthia Kop[1]

1 **Department of Computer Science, University of Innsbruck**
Technikerstraße 21a, 6020 Innsbruck, Austria
`Cynthia.Kop@uibk.ac.at`

Abstract

Logically Constrained Term Rewriting Systems (LCTRSs) provide a general framework for term rewriting with constraints. We discuss a simple dependency pair approach to prove termination of LCTRSs. We see that existing techniques transfer to the constrained setting in a natural way.

1 Introduction

In [4], *logically constrained term rewriting systems* are introduced (building on [3] and [2]). These *LCTRSs* combine many-sorted term rewriting with constraints in an arbitrary theory, and can be used for analysing for instance imperative programs.

Termination is an important part of such analysis, both for its own sake (to guarantee finite program evaluation), and to create an induction principle that can be used as part of other analyses (for instance proofs of confluence [6] or function equality [3]).

In unconstrained term rewriting, many termination techniques exist, often centred around *dependency pairs* [1]. Some of these methods have also been transposed to integer rewriting with constraints [2]. However, that setting is focused purely on proving termination for its own sake, and thus poses very strong restrictions on term and rule formation.

In this paper, we will see how a basic dependency pair approach can be defined for LCTRSs, and extend several termination methods which build around dependency pairs.

2 Preliminaries (from [4])

We assume standard notions of many-sorted term rewriting to be well-uderstood.

Let $\mathcal{V}$ be an infinite set of sorted variables, $\Sigma = \Sigma_{terms} \cup \Sigma_{theory}$ be a many-sorted signature, $\mathcal{I}$ a mapping which assigns to each sort occurring in Σ_{theory} a set, and $\mathcal{J}$ a function which maps each $f : [\iota_1 \times \ldots \times \iota_n] \Rightarrow \kappa \in \Sigma_{theory}$ to a function $\mathcal{J}_f$ in $\mathcal{I}_{\iota_1} \Longrightarrow \ldots \Longrightarrow \mathcal{I}_{\iota_n} \Longrightarrow \mathcal{I}_\kappa$. For every sort ι occurring in Σ_{theory} we also fix a set $\mathcal{V}al_\iota \subseteq \Sigma_{theory}$ of *values*: function symbols $a : [] \Rightarrow \iota$, where $\mathcal{J}$ gives a one-to-one mapping from $\mathcal{V}al_\iota$ to $\mathcal{I}_\iota$. A value c is identified with the term $c()$. The elements of Σ_{theory} and Σ_{terms} overlap only on values.

We call a term in $\mathcal{T}erms(\Sigma_{theory}, \mathcal{V})$ a *logical term*. For ground logical terms, we define $[\![f(s_1, \ldots, s_n)]\!] := \mathcal{J}_f([\![s_1]\!], \ldots, [\![s_n]\!])$. A ground logical term s *has value* t if t is a value such that $[\![s]\!] = [\![t]\!]$. Every ground logical term has a unique value. A *constraint* is a logical term of some sort bool with $\mathcal{I}_{\mathsf{bool}} = \mathbb{B}$, the set of booleans. A constraint s is *valid* if $[\![s\gamma]\!]_{\mathcal{J}} = \top$ for all substitutions γ which map the variables in $Var(s)$ to a value.

A *rule* is a triple $l \to r\ [\varphi]$ where l and r are terms with the same sort and φ is a constraint; l is not a logical term (so also not a variable). If $\varphi = \mathsf{true}$ with $\mathcal{J}(\mathsf{true}) = \top$, the rule is just denoted $l \to r$. We define $LVar(l \to r\ [\varphi]))$ as $Var(\varphi) \cup (\,Var(r) \setminus Var(l))$. A substitution γ *respects* $l \to r\ [\varphi]$ if $\gamma(x)$ is a value for all $x \in LVar(l \to r\ [\varphi])$ and $\varphi\gamma$ is valid.

* The research described in this paper is supported by the Austrian Science Fund (FWF) international project I963 and the Japan Society for the Promotion of Science.

Given a set of rules $\mathcal{R}$, the *rewrite relation* $\to_{\mathcal{R}}$ is the union of $\to_{\texttt{rule}}$ and $\to_{\texttt{calc}}$, where:

- $C[l\gamma] \to_{\texttt{rule}} C[r\gamma]$ if $l \to r\ [\varphi] \in \mathcal{R}$ and γ respects $l \to r\ [\varphi]$;
- $C[f(s_1, \ldots, s_n)] \to_{\texttt{calc}} C[v]$ if $f \in \Sigma_{theory} \setminus \Sigma_{terms}$, all s_i values and v is the value of $f(\vec{s})$

A reduction step with $\to_{\texttt{calc}}$ is called a *calculation*. In an LCTRS with rules $\mathcal{R}$, the *defined symbols* are all symbols f such that a rule $f(\vec{l}) \to r\ [\varphi]$ exists in $\mathcal{R}$. Symbols $f \in \Sigma_{theory} \setminus \mathcal{V}al$ are called *calculation symbols* and all other symbols are *constructors*.

▶ **Example 1.** We consider an LCTRS with sorts int and bool, with $\mathcal{I}_{\mathsf{bool}} = \mathbb{B}$ and int mapped to the set of 16-bit signed integers; addition is sensitive to overflow. The rules are a naive implementation of the Ackermann function (which will likely fall prey to overflows):

$$\begin{array}{lcll} \mathsf{A}(m,n) & \to & \mathsf{A}(m-1,\mathsf{A}(m,n-1)) & [m \neq 0 \wedge n \neq 0] \\ \mathsf{A}(m,0) & \to & \mathsf{A}(m-1,1) & [m \neq 0] \end{array} \qquad \begin{array}{lcl} \mathsf{A}(0,n) & \to & n+1 \end{array}$$

A is a defined symbols, $+, -, \neq, \wedge$ calculation symbols, and all integers are constructors.

3 Dependency Pairs

As the basis for termination analysis, we will consider *dependency pairs* [1]. We first introduce a fresh sort dpsort, and for all defined symbols $f : [\iota_1 \times \ldots \times \iota_n] \Rightarrow \kappa$ also a new symbol $f^\sharp : [\iota_1 \times \ldots \times \iota_n] \Rightarrow \mathsf{dpsort}$. If $s = f(s_1, \ldots, s_n)$ with f defined, then $s^\sharp := f^\sharp(s_1, \ldots, s_n)$.

The dependency pairs of a given rule $l \to r\ [\varphi]$ are all rules of the form $l^\sharp \to p^\sharp\ [\varphi]$ where p is a subterm of r which is headed by a defined symbol. The set of dependency pairs for a given set of rules $\mathcal{R}$, notation $\mathsf{DP}(\mathcal{R})$, consists of all dependency pairs of any rule in $\mathcal{R}$.

▶ **Example 2.** Noting that for instance $\mathsf{A}^\sharp(m,0) \to m -^\sharp 1$ is *not* a dependency pair, since $-$ is a calculation symbol and not a defined symbol, Example 1 has three dependency pairs:

$$\begin{array}{lcll} 1. & \mathsf{A}^\sharp(m,0) & \to & \mathsf{A}^\sharp(m-1,1) & [m \neq 0] \\ 2. & \mathsf{A}^\sharp(m,n) & \to & \mathsf{A}^\sharp(m-1,\mathsf{A}(m,n-1)) & [m \neq 0 \wedge n \neq 0] \\ 3. & \mathsf{A}^\sharp(m,n) & \to & \mathsf{A}^\sharp(m,n-1) & [m \neq 0 \wedge n \neq 0] \end{array}$$

Fixing a set $\mathcal{R}$ of rules, and given a set $\mathcal{P}$ of dependency pairs, a *$\mathcal{P}$-chain* is a sequence $\rho_1, \rho_2, \ldots$ of dependency pairs such that all ρ_i are elements of $\mathcal{P}$, but with distinctly renamed variables, and there is some γ which respects all ρ_i, such that for all i: if $\rho_i = l_i \to p_i\ [\varphi_i]$ and $\rho_{i+1} = l_{i+1} \to p_{i+1}\ [\varphi_{i+1}]$, then $p_i\gamma \to_{\mathcal{R}}^* l_{i+1}\gamma$. Also, the strict subterms of $l_i\gamma$ terminate. We call $\mathcal{P}$ a *DP problem* and say that $\mathcal{P}$ is *chain-free* if there is no infinite $\mathcal{P}$-chain.[1][2]

▶ **Theorem 3.** *An LCTRS $\mathcal{R}$ is terminating if and only if $\mathsf{DP}(\mathcal{R})$ is chain-free.*

4 The Dependency Graph

To prove chain-freeness of a DP problem, we might for instance use the dependency graph:

▶ **Definition 4.** A *dependency graph approximation* of a DP problem $\mathcal{P}$ is a graph G whose nodes are the elements of $\mathcal{P}$ and which has an edge between ρ_1 and ρ_2 if (ρ_1, ρ_2') is a $\mathcal{P}$-chain, where ρ_2' is a copy of ρ_2 with fresh variables. G may have additional edges.

▶ **Theorem 5.** *A DP problem $\mathcal{P}$ with graph approximation G is chain-free if and only if $\mathcal{P}'$ is chain-free for every strongly connected component (SCC) $\mathcal{P}'$ of G.*

[1] In the literature, we consider tuples of sets and flags, which is necessary if we also want to consider non-minimal chains, innermost termination or non-termination. For simplicity those are omitted here.

[2] In the literature, the word *finite* is used instead of *chain-free*. Since we have a single set instead of a tuple, we used a different word to avoid confusion (as "finite" might refer to the number of elements).

▶ **Example 6.** Consider an LCTRS with rules $\mathcal{R} = \{f(x) \to f(0 - x)\ [x > 0]\}$. Then $\mathsf{DP}(\mathcal{R}) = \{f^\sharp(x) \to f^\sharp(-x)\ [x > 0]\}$. The dependency graph of $\mathsf{DP}(\mathcal{R})$ has one node, and no edges, since there is no substitution γ which satisfies both $\gamma(x) > 0$ and $\gamma(y) > 0$ and yet has $(-x)\gamma \to^*_{\mathcal{R}} y\gamma$ (as logical terms reduce only with $\to_{\mathtt{calc}}$). Thus, clearly every SCC of this graph is terminating, so $\mathsf{DP}(\mathcal{R})$ is chain-free, so $\mathcal{R}$ is terminating!

Of course, manually choosing a graph approximation is one thing, but finding a good one *automatically* is more difficult. We consider one way to choose such an approximation:

Given a DP problem $\mathcal{P}$, let $G_\mathcal{P}$ be the graph with the elements of $\mathcal{P}$ as nodes, and with an edge from $l_1 \to r_1\ [\varphi_1]$ to $l_2 \to r_2\ [\varphi_2]$ if the formula $\varphi_1 \wedge \varphi_2' \wedge \psi(r_1, l_2', LVar(l_1 \to r_1\ [\varphi_1]) \cup LVar(l_2' \to r_2'\ [\varphi_2']))$ is satisfiable (or its satisfiability cannot be determined). Here, $l_2' \to r_2'\ [\varphi_2']$ is a copy of $l_2 \to r_2\ [\varphi_2]$ with fresh variables, and $\psi(s, t, L)$ is given by the clauses:

- $\psi(s, t, L) = \top$ if either s is a variable not in L, or $s = f(s_1, \ldots, s_n)$ and one of:
  - f is a defined symbol, and $s \notin \mathcal{T}erms(\Sigma_{theory}, L)$,
  - f is a calculation symbol, t a value or variable, and $s \notin \mathcal{T}erms(\Sigma_{theory}, L)$,
  - f is a constructor and t a variable not in L;
- $\psi(s, t, L) = \bigwedge_{i=1}^n \psi(s_i, t_i, L)$ if $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$ and f not defined;
- $\psi(s, t, L)$ is the formula $s = t$ if $s \in \mathcal{T}erms(\Sigma_{theory}, L)$, $t \in \mathcal{T}erms(\Sigma_{theory}, \mathcal{V})$ and s and t are not headed by the same theory symbol (we already covered that case);
- $\psi(s, t, L) = \bot$ in all other cases.

▶ **Theorem 7.** *$G_\mathcal{P}$ is a graph approximation for $\mathcal{P}$.*

This graph result and the given approximation correspond largely with the result of [5].

▶ **Example 8.** The graph in Example 6 is calculated with this method: $\psi(f^\sharp(-x), f^\sharp(y), \{x, y\}) \wedge x > 0 \wedge y > 0$ evaluates to $-x = y \wedge x > 0 \wedge y > 0$ (as $f^\sharp$ is a constructor with respect to $\mathcal{R}$), which is not satisfiable (as any decent SMT-solver over the integers can tell us).

5 The Value Criterion

To quickly handle DP problems, we consider a technique similar to the subterm criterion in the unconstrained case. This *value criterion* can also be seen as a simpler version of polynomial interpretations, which does not require ordering rules (see Section 6).

▶ **Definition 9.** Fixing a set $\mathcal{P}$ of dependency pairs, a *projection function* for $\mathcal{P}$ is a function ν which assigns to each symbol $f^\sharp : [\iota_1 \times \ldots \times \iota_n] \Rightarrow \mathsf{dpsort}$ a number $\nu(f^\sharp) \in \{1, \ldots, n\}$. A projection function is extended to a function on terms as follows: $\overline{\nu}(f^\sharp(s_1, \ldots, s_n)) = s_{\nu(f^\sharp)}$.

▶ **Theorem 10.** *Let $\mathcal{P}$ be a set of dependency pairs, ι a sort and ν a projection function for $\mathcal{P}$, with the following property: for any dependency pair $l \to r\ [\varphi] \in \mathcal{P}$, if $\overline{\nu}(l)$ has sort ι and is a logical term (this includes variables), then the same holds for $\overline{\nu}(r)$. Let moreover $\succ$ be a well-founded ordering relation on $\mathcal{I}_\iota$ and $\succeq$ a quasi-ordering such that $\succ \cdot \succeq \subseteq \succ$. Suppose additionally that we can write $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$, such that for all $\rho = l \to r\ [\varphi] \in \mathcal{P}$:*

- *if $\overline{\nu}(l)$ is a logical term of sort ι, then so is $\overline{\nu}(r)$, and $Var(\overline{\nu}(r)) \subseteq Var(\overline{\nu}(l))$;*
- *if $\rho \in \mathcal{P}_1$, then $\overline{\nu}(l)$ has sort ι and $\overline{\nu}(l) \in \mathcal{T}erms(\Sigma_{theory}, LVar(\rho))$;*
- *if $\overline{\nu}(l)$ has sort ι and $\overline{\nu}(l) \in \mathcal{T}erms(\Sigma_{theory}, \mathcal{V})$, then $\varphi \Rightarrow \overline{\nu}(l) \succ \overline{\nu}(r)$ is valid if $\rho \in \mathcal{P}_1$, and $\varphi \Rightarrow \overline{\nu}(l) \succeq \overline{\nu}(r)$ is valid if $\rho \in \mathcal{P}_2$.*

Then $\mathcal{P}$ is chain-free if and only if $\mathcal{P}_2$ is chain-free.

Proof. A chain with infinitely many elements of $\mathcal{P}_1$ gives an infinite $\succeq^* \cdot \succ$ reduction. ◀

▶ **Example 11.** Using the value criterion, we can complete termination analysis of the Ackermann example. Choosing for $\succ$ the *unsigned* comparison on bitvectors (so $n \succ m$ if either n is negative and m is not, or $sign(n) = sign(m)$ and $n > m$), and $\nu(\mathsf{A}) = 1$, we have:

- $\mathsf{A}^\sharp(m, 0) \to \mathsf{A}^\sharp(m-1, 1)$ $[m \neq 0]$: $(m \neq 0) \Rightarrow m \succ m-1$
- $\mathsf{A}^\sharp(m, n) \to \mathsf{A}^\sharp(m-1, \mathsf{A}(m, n-1))$ $[m \neq 0 \wedge n \neq 0]$: $(m \neq 0 \wedge n \neq 0) \Rightarrow m \succ m-1$
- $\mathsf{A}^\sharp(m, n) \to \mathsf{A}^\sharp(m, n-1)$ $[m \neq 0 \wedge n \neq 0]$ $(m \neq 0 \wedge n \neq 0) \Rightarrow m \succeq m$

All three are valid, so $\mathcal{P}$ is chain-free if $\mathcal{P}_2 = \{\mathsf{A}^\sharp(m, n) \to \mathsf{A}^\sharp(m, n-1)\ [m \neq 0 \wedge 0 \wedge n \neq 0]\}$ is. This we prove with another application of the value criterion, now taking $\nu(\mathsf{A}^\sharp) = 2$.

Note that the difficulty to apply the value criterion is in finding a suitable value ordering. There are various systematic techniques for doing this (depending on the underlying theory), but their specifics are beyond the scope of this paper.

6 Reduction Pairs

Finally, the most common method to prove chain-freeness is the use of a *reduction pair*.

A reduction pair $(\succsim, \succ)$ is a pair of a *monotonic quasi-ordering* and a *well-founded partial ordering* on terms such that $s \succ t \succsim q$ implies $s \succ q$. Note that it is not required that $\succ$ is included in $\succsim$; $\succsim$ might also for instance be an equivalence relation. A rule $l \to r\ [\varphi]$ is *compatible* with $R \in \{\succsim, \succ\}$ if for all substitutions γ which respect the rule we have: $l\gamma\ R\ r\gamma$.

▶ **Theorem 12.** *A set of dependency pairs $\mathcal{P}$ is chain-free if and only if there is a reduction pair $(\succsim, \succ)$ and we can write $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$ such that $\mathcal{P}_2$ is chain-free, and:*

- *all $\rho \in \mathcal{P}_1$ are compatible with $\succ$ and all $\rho \in \mathcal{P}_2$ are compatible with $\succsim$;*
- *either all $\rho \in \mathcal{R}$ are compatible with $\succsim$,*
  or all $\rho \in \mathcal{P}$ have the form $l \to f(s_1, \ldots, s_i)\ [\varphi]$ with all $s_i \in \mathcal{T}erms(\Sigma_{theory}, LVar(\rho))$;
- *$f(\vec{v}) \succsim w$ if f is a calculation symbol, $v_1, \ldots, v_n$ are values and w is the value of $f(\vec{v})$.*

Note that all rules must be compatible with $\succsim$, unless the subterms of the right-hand sides in $\mathcal{P}$ can only be instantiated to ground logical terms; in this (reasonably common!) case, we can ignore the rules in the termination argument. This is a weak step in the direction of *usable rules*, a full treatment of which is beyond the scope of this short paper.

For the reduction pair, we might for instance use the recursive path ordering described in [4]. Alternatively, we could consider *polynomial interpretations*:

▶ **Theorem 13.** *Given a mapping μ which assigns to each function symbol $f : [\iota_1 \times \ldots \times \iota_n] \Rightarrow \kappa \in \Sigma_{terms} \cup \Sigma_{theory}$ an n-ary polynomial over $\mathbb{Z}$, and a valuation α which maps each variable to an integer, every term s corresponds to an integer $\overline{\mu}_\alpha(s)$. Let $s \succ t$ if for all α: $\overline{\mu}_\alpha(s) > \max(0, \overline{\mu}_\alpha(t))$, and $s \succsim t$ if for all α: $\overline{\mu}_\alpha(s) = \overline{\mu}_\alpha(t)$. Then $(\succsim, \succ)$ is a reduction pair.*

Here, $\succsim$ is an equivalence relation. Alternatively we might base $\succsim$ on the $\geq$ relation in $\mathbb{Z}$, but then we must pose an additional weak monotonicity requirement on μ.

▶ **Example 14.** We consider an LCTRS over the integers, without overflow. This example uses bounded iteration, which is common in systems derived from imperative programs:

$$\mathsf{sum}(x, y) \to 0\ [x > y] \qquad \mathsf{sum}(x, y) \to x + \mathsf{sum}(x+1, y)\ [x \leq y]$$

This system admits one dependency pair: $\mathsf{sum}^\sharp(x, y) \to \mathsf{sum}^\sharp(x+1, y)\ [x \leq y]$. Neither the dependency graph nor the value criterion can handle this pair. We *can* orient it using polynomial interpretations, with $\mu(\mathsf{sum}) = \lambda nm.m - n + 1$; integer functions and integers are interpreted as themselves. Then $x \leq y \Rightarrow y - x + 1 > \max(0, y - (x+1) + 1)$ is valid, so the pair is compatible with $\succ$ as required.

Thus, $\mathsf{DP}(\mathcal{R})$ is chain-free if and only if $\emptyset$ is chain-free, which is obviously the case!

7 Related Work

The most important related work is [2], where a constrained term rewriting formalism over the integers is introduced, and methods are developed to prove termination similar to the ones discussed here. The major difference with the current work is that the authors of [2] impose very strong type restrictions: they consider only theory symbols (of sort `int`) and defined symbols (of sort `unit`). Rules have the form $f(x_1, \ldots, x_n) \to g(s_1, \ldots, s_n)$, where the x_i are variables and all s_i are logical terms. This significantly simplifies the analysis (for example, the dependency pairs are exactly the rules), but has more limited applications; it suffices for proving termination of simple (imperative) integer programs, but does not help directly for analysing confluence or function equivalence.

8 Conclusion

In this paper, we have seen how termination methods for normal TRSs, and in particular the dependency pair approach, extend naturally to the setting of LCTRSs. Decision procedures are handled by solving validity of logical formulas. While this is undecidable in general, many practical cases can be handled using today's powerful SMT-solvers.

Considering termination results, we have only seen the tip of the iceberg. In the future, we hope to extend the constrained dependency pair framework to handle also innermost termination and non-termination. Moreover, the dependency pair approach can be strengthened with various techniques for simplifying dependency pair processors, both adaptations of existing techniques for unconstrained term rewriting (such as usable rules) and specific methods for constrained term rewriting (such as the *chaining* method used in [2] or methods to add constraints in some cases).

In addition, we hope to provide an automated termination tool for LCTRSs in the near future. Such a tool could for instance be coupled with a transformation tool from e.g. C or Java to be immediately applicable for proving termination of imperative programs, or can be used as a back-end for analysis tools of confluence or function equivalence.

References

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

2 S. Falke and D. Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In R. Schmidt, editor, *Proc. CADE 09*, volume 5663 of *LNCS*, pages 277–293. Springer, 2009.

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

4 C. Kop and N. Nishida. Term rewriting with logical constraints. In *Proc. FroCoS 13*, 2013. To Appear, `http://cl-informatik.uibk.ac.at/users/kop/frocos13.pdf`.

5 T. Sakata, N. Nishida, and T. Sakabe. On proving termination of constrained term rewrite systems by eliminating edges from dependency graphs. In H. Kuchen, editor, *Proc. WFLP 11*, LNCS, pages 138–155. Springer, 2011.

6 Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in TCS*. Cambridge University Press, 2003.

Program Termination analysis using MAX-SMT*

Daniel Larraz[1], Albert Oliveras[1], Enric Rodríguez-Carbonell[1], and Albert Rubio[1]

1 Universitat Politècnica de Catalunya, Barcelona, Spain

Abstract

We show how Max-SMT can be exploited in constraint-based program termination proving. The generation of a ranking function is expressed as a Max-SMT optimization problem where constraints are assigned different weights. As a result, quasi-ranking functions –functions that almost satisfy all conditions for ensuring well-foundedness– are produced in a lack of ranking functions. This allows our method to progress in the termination analysis where other approaches would get stuck. Moreover, Max-SMT makes it easy to combine the process of building the termination argument with the usually necessary task of generating supporting invariants. The method has been implemented in a prototype and succesfully tested on a wide set of programs showing its potential in practice.

1 Introduction

Proving termination is necessary to ensure total correctness of programs. Still, termination bugs are difficult to trace and are hardly notified: as they do not arise as system failures but as unresponsive behavior, when faced to them users tend to restart their devices without reporting to software developers. Due to this, approaches for proving termination of imperative programs have regained an increasing interest in the last decade [1, 2, 3, 4].

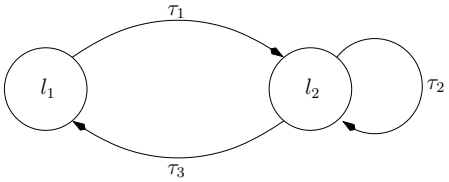
One of the major difficulties in these methods is that often *supporting invariants* are needed. In [5], by formulating both invariant and ranking function synthesis as constraint problems, both can be solved simultaneously, so that only the necessary supporting invariants for the targeted ranking functions –namely, *lexicographic linear ranking functions*– need to be discovered.

Based on this idea, we present a Max-SMT constraint-based approach for proving termination. The crucial observation in our method is that, although our goal is to show that transitions cannot be executed infinitely by finding a ranking function or an invariant that disables them, if we only discover an invariant, or a *quasi-ranking function* that almost fulfills all needed properties for well-foundedness, we have made some progress: either we can remove *part of a transition* and/or we have improved our knowledge on the behavior of the program. A natural way to implement this idea is by considering that some of the constraints are *hard* (the ones guaranteeing invariance) and others are *soft* (those guaranteeing well-foundedness) in a Max-SMT framework. Moreover, by giving different weights to the constraints we can set priorities and favor those invariants and (quasi-) ranking functions that lead to the furthest progress.

The technique has been implemented in our prototype of C++ analyzer CppInv. Thanks to our tool, we have proved termination of a wide set of programs, which have been taken from the programming learning environment Jutge.org [6] and from benchmark suites in the literature [7].

* This work has been partially supported by the Spanish MEC/MICINN under grantTIN 2010-68093-C02-01

```
int main () {
  int x, y, z;
  ℓ1 : while (y ≥ 1) {
    x−−;
    ℓ2 : while (y < z) {
      x++;
      z−−;
    }
    y = x + y;
  }
}
```



$$\Theta(\ell_1) \equiv \mathit{true} \qquad \Theta(\ell_2) \equiv \mathit{false}$$

$$\begin{array}{lllll} \rho_{\tau_1}: & y \geq 1, & x' = x - 1, & y' = y, & z' = z \\ \rho_{\tau_2}: & y < z, & x' = x + 1, & y' = y, & z' = z - 1 \\ \rho_{\tau_3}: & y \geq z, & x' = x, & y' = x + y, & z' = z \end{array}$$

Figure 1 Program and its transition system.

2 Encoding Termination using MAX-SMT

In this paper we model imperative programs by means of *transition systems*. See Fig. 1 for an example of a program together with the corresponding transition system. Note that primed versions of the variables represent the values of the variables after the transition and that Θ is a map from locations to formulas characterizing the initial values of the variables. From now on we assume that variables take integer values and programs are linear, i.e., the initial conditions Θ and transition relations ρ are described as conjunctions of linear inequalities.

An important class of invariant maps is that of *inductive invariant maps*:

▶ **Definition 1.** An invariant map μ is said to be *inductive* if:

- **[Initiation]** For every location ℓ: $\Theta(\ell) \models \mu(\ell)$
- **[Consecution]** For every transition $\tau = (\ell, \ell', \rho)$: $\mu(\ell) \wedge \rho \models \mu(\ell')'$.

The basic idea of the approach we follow for proving program termination [8] is to argue by contradiction that no transition is infinitely executable. First of all, it is obvious that disabled transitions (i.e., that can never be executed) cannot be infinitely executable. Moreover, one just needs to focus on transitions joining locations in the same strongly connected component (SCC): if a transition is executed over and over again, then its pre and post locations must belong to the same SCC. So let us assume that one has found a *ranking function* for such a transition τ, according to the following definition:

▶ **Definition 2.** Let $\tau = (\ell, \ell', \rho)$ be a transition such that ℓ and ℓ' belong to the same SCC, denoted by C. A function R is said to be a *ranking function* for τ if:

- **[Boundedness]** $\rho \models R \geq 0$
- **[Strict Decrease]** $\rho \models R > R'$
- **[Non-increase]** For every $\hat{\tau} = (\hat{\ell}, \hat{\ell}', \hat{\rho})$ such that $\hat{\ell}, \hat{\ell}' \in C$: $\hat{\rho} \models R \geq R'$

Note that boundedness and strict decrease *only* depend on τ, while non-increase depends on *all* transitions in the SCC.

Similarly to [5], we consider linear invariant and linear ranking function templates and take the following constraints from the definitions of inductive invariant and ranking function:

Initiation:	For ℓ:	$\mathbb{I}_\ell \stackrel{def}{=} \Theta(\ell) \vdash I_\ell$
Disability:	For $\tau = (\ell, \ell', \rho)$:	$\mathbb{D}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash 1 \leq 0$
Consecution:	For $\tau = (\ell, \ell', \rho)$:	$\mathbb{C}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash I'_{\ell'}$
Boundedness:	For $\tau = (\ell, \ell', \rho)$:	$\mathbb{B}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash R \geq 0$
Strict Decrease:	For $\tau = (\ell, \ell', \rho)$:	$\mathbb{S}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash R > R'$
Non-increase:	For $\tau = (\ell, \ell', \rho)$:	$\mathbb{N}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash R \geq R'$

Finally, let L and T be the sets of locations and transitions in the SCC under consideration, respectively. Let also P be the set of transitions that are *p*ending to be proved finitely executable. Then we construct the following constraint system, which is later on transformed into an SMT problem over linear and non-linear arithmetic:

$$\bigwedge_{\ell \in L} \mathbb{I}_\ell \wedge \bigwedge_{\tau \in T} \big(\mathbb{D}_\tau \vee \mathbb{C}_\tau\big) \wedge \bigvee_{\tau \in P} \big(\mathbb{D}_\tau \vee (\mathbb{B}_\tau \wedge \mathbb{S}_\tau)\big) \wedge \big((\bigwedge_{\tau \in P} \mathbb{N}_\tau) \vee \bigvee_{\tau \in P} \mathbb{D}_\tau\big).$$

The first two conjuncts guarantee that an invariant map is computed; the other two, that at least one of the pending transitions can be discarded. Notice that, if there is no disabled transition, we ask that *all* transitions in P are non-increasing, but only that at least *one* transition in P (the next to be removed) is both bounded and strict decreasing. Note also that for finding invariants one has to take into account *all* transitions in the SCC, even those that have already been proved to be finitely executable: otherwise some reachable states might not be covered, and the invariant generation would become unsound. Hence in our termination analysis we consider two transition systems: the original transition system for invariant synthesis, whose transitions are T and which remains all the time the same; and the *termination transition system*, whose transitions are P, i.e, where transitions already shown to be finitely executable have been removed. This duplication is similar to the *cooperation graph* of [7].

The idea is to consider the constraints guaranteeing invariance as *hard*, so that any solution to the constraint system will satisfy them, while the rest are *soft*. Let us consider propositional variables $p_\mathbb{B}$, $p_\mathbb{S}$ and $p_\mathbb{N}$, which intuitively represent if the conditions of boundedness, strict decrease and non-increase in the definition of ranking function are violated respectively, and corresponding weights $\omega_\mathbb{B}$, $\omega_\mathbb{S}$ and $\omega_\mathbb{N}$. We consider now the next constraint system (where soft constraints are written $[\cdot, \omega]$, and hard ones as usual):

$$\bigwedge_{\ell \in L} \mathbb{I}_\ell \wedge \bigwedge_{\tau \in T} \Big(\mathbb{D}_\tau \vee \mathbb{C}_\tau\Big) \wedge \bigvee_{\tau \in P} \Big(\mathbb{D}_\tau \vee \big((\mathbb{B}_\tau \vee p_\mathbb{B}) \wedge (\mathbb{S}_\tau \vee p_\mathbb{S})\big)\Big) \wedge$$
$$\Big((\bigwedge_{\tau \in P} \mathbb{N}_\tau) \vee \bigvee_{\tau \in P} \mathbb{D}_\tau \vee p_\mathbb{N}\Big) \wedge [\neg p_\mathbb{B}, \omega_\mathbb{B}] \wedge [\neg p_\mathbb{S}, \omega_\mathbb{S}] \wedge [\neg p_\mathbb{N}, \omega_\mathbb{N}].$$

Note that, since all constraints are fulfiled, ranking functions have cost 0, and (if no transition is disabled) functions that fail in any of the conditions are penalized with the respective weight. Thus, the Max-SMT solver looks for the best solution and gets a ranking function if feasible; otherwise, the weights guide the search to get invariants and quasi-ranking functions that satisfy as many conditions as possible.

Hence this Max-SMT approach allows recovering information even from problems that would be unsatisfiable in the initial method. This information can be exploited to perform dynamic trace partitioning [9] as follows. Assume that the optimal solution to the above Max-SMT formula has been computed, and let us consider a transition $\tau \in P$ such that $\mathbb{D}_\tau \vee ((\mathbb{B}_\tau \vee p_\mathbb{B}) \wedge (\mathbb{S}_\tau \vee p_\mathbb{S}))$ evaluates to true in the solution. Then we distinguish several cases depending on the properties satisfied by τ and the computed function R:

- If τ is disabled then it can be removed.

- If R is non-increasing and satisfies boundedness and strict decrease for τ, then τ can be removed too: R is a ranking function for it.
- If R is non-increasing and satisfies boundedness for τ but not strict decrease, one can split τ in the termination transition system into two new transitions: one where $R > R'$ is added to τ, and another one where $R = R'$ is enforced. Then the new transition with $R > R'$ is automatically eliminated, as R is a ranking function for it. Equivalently, this can be seen as adding $R = R'$ to τ. Now, if the solver could not prove R to be a true ranking function for τ because it was missing an invariant, this transformation will guide the solver to find that invariant so as to disable the transition with $R = R'$.
- If R is non-increasing and satisfies strict decrease for τ but not boundedness, the same technique from above can be applied: it boils down to adding $R < 0$ to τ.
- If R is non-increasing but neither strict decrease nor boundedness are fulfilled for τ, then τ can be split into two new transitions: one with $R < 0$, and another one with $R \geq 0 \wedge R = R'$.
- If R does not satisfy the non-increase property, then it is rejected; however, the invariant map from the solution can be used to strengthen the transition relations for the following iterations of the termination analysis.

Note this analysis may be worth applying on other transitions τ in the termination transition system apart from those that make $\mathbb{D}_\tau \vee ((\mathbb{B}_\tau \vee p_\mathbb{B}) \wedge (\mathbb{S}_\tau \vee p_\mathbb{S}))$ true. E.g., if R is a ranking function for a transition τ but fails to be so for another one τ' because strict decrease does not hold, then, according to the above discussion, τ' can be strengthened with $R = R'$.

On the other hand, working in this iterative way requires imposing additional constraints to avoid getting to a standstill. Namely, in the case where non-increase does not hold and so one would like to exploit the invariant, it is necessary to impose that the invariant is not redundant.

Another advantage of this Max-SMT approach is that by using different weights we can express priorities over conditions. Since, as explained above, violating the property of non-increase invalidates the computed function R, it is convenient to make $\omega_\mathbb{N}$ the largest weight. On the other hand, when non-increase and boundedness are fulfilled but not strict decrease an equality is added to the transition, whereas when non-increase and strict decrease are fulfilled but not boundedness just an inequality is added. As we prefer the former to the latter, in our implementation we set $\omega_\mathbb{B} > \omega_\mathbb{S}$.

Further refinements are possible. E.g., the termination transition system can also be used for generating properties that are guaranteed to eventually hold at a location for some computations. More specifically, we devised the following light-weight approach for generating what we call *termination implications*. In a nutshell, for each location ℓ a new linear inequality template J_ℓ is introduced and the following constraint is imposed: $\bigwedge_{\tau=(\hat{\ell},\ell,\rho)\in P}(\mathbb{D}_\tau \vee I_{\hat{\ell}} \wedge \rho \vdash J'_\ell)$. The rationale is that, if we find a property J_ℓ that is implied by all transitions going into ℓ and ℓ is finally reached, then J_ℓ must hold. Then this termination implication can be propagated forward to the transitions going out from ℓ, i.e., J_ℓ can be conjoined to $I_\ell \wedge \rho$ in the termination transition system. Finally, additional constraints are imposed to ensure that new termination implications are not redundant with the already computed invariants and termination implications.

▶ **Example 3.** Let us show a termination analysis of the program in Fig. 1. In the first round, the solver finds the invariant $y \geq 1$ at ℓ_2 and the ranking function z for τ_2. While $y \geq 1$ can be added to τ_3 (resulting into a new transition τ'_3), the ranking function allows eliminating τ_2 from the termination transition system.

In the second round, the solver cannot find a ranking function. However, thanks to the Max-SMT formulation, it can produce the quasi-ranking function x, which is non-increasing and strict decreasing for τ_1, but not bounded. This quasi-ranking function can be used to split transition τ_1 into two new transitions $\tau_{1.1}$ and $\tau_{1.2}$ as follows:

$$\rho_{\tau_{1.1}}: \quad \boldsymbol{x \geq 0}, \quad y \geq 1, \quad x' = x - 1, \quad y' = y, \quad z' = z$$
$$\rho_{\tau_{1.2}}: \quad \boldsymbol{x < 0}, \quad y \geq 1, \quad x' = x - 1, \quad y' = y, \quad z' = z$$

Then $\tau_{1.1}$ is immediately removed, since x is a ranking function for it.

In the third and final round, the termination implication $x < 0$ is generated at ℓ_2, together with the ranking function y for transition τ_3'. Note that the termination implication is crucial to prove the strict decrease of y for τ_3', and that the previously generated invariant $y \geq 1$ at ℓ_2 is needed to ensure boundedness. Now τ_3' can be removed, which makes the graph acyclic. This concludes the termination proof.

3 Conclusion

The method presented here has been implemented in the tool CppInv[1].

This tool has been proved competitive in comparison with the new version of T2, which according to the results given in [7] is performing much better when proving termination than most of the existing tools.

For a full description of the method, its implementation and the experimental evaluation, see [10].

References

1 D. Dams, R. Gerth, O. Grumberg, A heuristic for the automatic generation of ranking functions, in: Workshop on Advances in Verification, 2000, pp. 1–8.

2 M. Colón, H. Sipma, Synthesis of linear ranking functions, in: TACAS, Vol. 2031 of Lecture Notes in Computer Science, Springer, 2001, pp. 67–81.

3 A. Podelski, A. Rybalchenko, A complete method for the synthesis of linear ranking functions, in: VMCAI, Vol. 2937 of Lecture Notes in Computer Science, Springer, 2004, pp. 239–251.

4 A. Tiwari, Termination of linear programs, in: CAV, Vol. 3114 of Lecture Notes in Computer Science, Springer, 2004, pp. 70–82.

5 A. R. Bradley, Z. Manna, H. B. Sipma, Linear ranking with reachability, in: CAV, Vol. 3576 of Lecture Notes in Computer Science, Springer, 2005, pp. 491–504.

6 J. Petit, O. Giménez, S. Roura, Jutge.org: an educational programming judge, in: SIGCSE, ACM, 2012, pp. 445–450.

7 M. Brockschmidt, B. Cook, C. Fuhs, Better termination proving through cooperation, in *CAV*, 2013, to appear.

8 M. Colón, H. Sipma, Practical methods for proving program termination, in: CAV, Vol. 2404 of Lecture Notes in Computer Science, Springer, 2002, pp. 442–454.

9 L. Mauborgne, X. Rival, Trace partitioning in abstract interpretation based static analyzers, in: M. Sagiv (Ed.), European Symposium on Programming (ESOP'05), Vol. 3444 of Lecture Notes in Computer Science, Springer-Verlag, 2005, pp. 5–20.

10 D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, A. Rubio, Proving termination of imperative programs using max-smt, in *FMCAD*, 2013, to appear.

[1] CppInv, together with all benchmarks used in the experimental evaluation, is available at `www.lsi.upc.edu/~albert/cppinv-term-bin.tar.gz`.

Piecewise-Defined Ranking Functions*

Caterina Urban[1]

1 **École Normale Supérieure - CNRS - INRIA**
Paris, France
urban@di.ens.fr

Abstract

We present the design and implementation of an abstract domain for proving program termination by abstract interpretation. The domain automatically synthesizes piecewise-defined ranking functions and infers sufficient conditions for program termination. The analysis is sound, meaning that all program executions respecting these sufficient conditions are indeed terminating.

We discuss the limitations of the proposed framework, and we investigate possible future work. In particular, we explore potential extensions of the abstract domain considering piecewise-defined non-linear ranking functions such as polynomials or exponentials.

1998 ACM Subject Classification D.1.4 Sequential Programming, D.2.4 Software/Program Verification, F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Abstract Interpretation, Ranking Function, Termination

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

1 Introduction

The traditional method for proving program termination [6] is based on the synthesis of ranking functions, which map program states to elements of a well-ordered set. A program terminates if a ranking function that decreases during program execution is found. In [4], Patrick Cousot and Radhia Cousot introduced the idea of the computation of a ranking function by Abstract Interpretation [3], a general theory of programs semantics approximation. In a recent work [10], we built on their proposed general framework, to design and implement a suitable abstract domain for proving termination of imperative programs.

Intuitively, we can define a ranking function from the states of a program to ordinal numbers, in an incremental way: we start from the program final states, where the function has value 0; then, we add states to the domain of the function, retracing the program backwards and counting the number of performed program steps as value of the function.

However, such ranking function is obviously not computable. Hence, we resort to abstract interpretation to automatically compute an abstract ranking function, which consists of abstract invariants attached to program points. These abstract invariants are represented by elements of an abstract domain and state properties about the program variables whenever control reaches that program point. More specifically, the elements of the abstract domain are piecewise-defined affine functions of the program variables, representing an upper bound on the number of program execution steps remaining before termination.

The domain automatically synthesizes such piecewise-defined ranking functions through backward invariance analysis. The analysis does not rely on assumptions about the structure

* The research leading to these results was partially funded by the MBAT project (EU ARTEMIS Joint Undertaking under grant agreement no. 269335).

	1st iteration	2nd iteration	3rd/4th iteration
4	$f(x) = 0$	$f(x) = 0$	$f(x) = 0$
1	$f(x) = \begin{cases} 1 & x > 10 \\ \bot & x \leq 10 \end{cases}$	$f(x) = \begin{cases} 1 & x > 10 \\ 4 & 9 \leq x \leq 10 \\ \bot & x \leq 8 \end{cases}$	$f(x) = \begin{cases} 1 & x > 10 \\ 4 & 9 \leq x \leq 10 \\ -3x + 38 & 7 \leq x \leq 8 \\ \bot & x \leq 6 \end{cases}$
3	$f(x) = \begin{cases} 2 & x > 8 \\ \bot & x \leq 8 \end{cases}$	$f(x) = \begin{cases} 2 & x > 8 \\ 5 & 7 \leq x \leq 8 \\ \bot & x \leq 6 \end{cases}$	$f(x) = \begin{cases} 2 & x > 8 \\ 5 & 7 \leq x \leq 8 \\ -3x + 33 & x \leq 6 \end{cases}$
2	$f(x) = \begin{cases} 3 & x > 8 \\ \bot & 7 \leq x \leq 8 \\ \bot & x \leq 6 \end{cases}$	$f(x) = \begin{cases} 3 & x > 8 \\ 6 & 7 \leq x \leq 8 \\ \bot & x \leq 6 \end{cases}$	$f(x) = \begin{cases} 3 & x > 8 \\ 6 & 7 \leq x \leq 8 \\ \bot & x \leq 6 \end{cases}$

Figure 2 Simple Example Analysis.

of the analyzed program: for example, is not limited to simple loops, as in [8]. To handle disjunctions arising from tests and loops, the analysis automatically partitions the space of values for the program variables into intervals, inducing a piecewise-definition of the affine ranking functions. During the analysis, pieces are split by tests, modified by assignments and joined when merging control flows. Widening limits the number of pieces of a ranking function to a maximum given as a parameter of the analysis.

Moreover, the domain naturally infers sufficient conditions for program termination. The analysis is sound: all program executions respecting these sufficient conditions are indeed terminating, while an execution that does not respect these conditions might not terminate.

Example Let us consider a small sequential programming language with no procedures, no pointers and no recursion. The language statements include assignments, branches and while loops. All program variables have (mathematical) integer values. In particular, let us consider the simple program in Figure 1. Figure 2 illustrates the details of the backward invariance analysis. We map each program control point to a function $f \in \mathbb{Z} \mapsto \mathbb{N}$ of the variable x, representing an upper bound on the number of execution steps before termination.

```
int : x
while [1](x <= 10) do
   if [2](x > 6) then
      [3]x := x + 2
   fi
od[4]
```

Figure 1 Simple Example

The analysis is performed backwards starting from the total function $f(x) = 0$ at program point 4. At program point 1, the loop test $x \leq 10$ splits the domain of the function and enforces termination in 1 step. At program point 3, the assignment $x := x + 2$ modifies the domain of the function and increases its value to 2. The, the test $x > 6$ further splits the domain of the function. Finally, a second iteration starts joining the function at program point 4 after $x > 10$ with the function at program point 2 after $x \leq 10$.

At the fourth iteration, a fix-point is reached yielding the following ranking function

$f \in \mathbb{Z} \mapsto \mathbb{N}$ as loop invariant at program point 1:

$$f(x) = \begin{cases} \bot_{\mathsf{F}} & x \leq 6 \\ -3x + 38 & 7 \leq x \leq 8 \\ 4 & 9 \leq x \leq 10 \\ 1 & x \geq 11 \end{cases}$$

The analysis provides $x > 6$ as a sufficient condition for termination, revealing potential non-termination for $x \leq 6$. Indeed, for $x \leq 6$, the program is non-terminating. ◀

2 Piecewise-Defined Affine Ranking Functions Abstract Domain

In the following, due to space constraints, we do not recall the results presented in [4] and we introduce straightaway our abstract domain of piecewise-defined affine ranking functions. Most definitions will be only hinted, we refer to [10] for more details and examples.

The elements of the abstract domain belong to $\mathcal{V}^\# \triangleq \mathcal{S}^\# \mapsto \mathcal{F}^\#$, where $\mathcal{S}^\#$ is the set of abstract program states (in particular, we abstract the program states using the intervals abstract domain [2]) and $\mathcal{F}^\# \triangleq \{\bot_{\mathsf{F}}\} \cup \{f^\# \mid f^\# \in \mathbb{Z}^n \mapsto \mathbb{N}\} \cup \{\top_{\mathsf{F}}\}$ is the set of natural-valued ranking functions of the integer-valued program variables (in addition to the function $\bot_{\mathsf{F}}$ representing potential non-termination, and the function $\top_{\mathsf{F}}$ representing the lack of enough information to conclude). More specifically, an abstract function $v^\# \in \mathcal{V}^\#$ has the form:

$$v^\# \equiv \begin{cases} s_1^\# \mapsto f_1^\# \\ s_2^\# \mapsto f_2^\# \\ \dots \\ s_k^\# \mapsto f_k^\# \end{cases}$$

where the abstract states $s_1^\#, \dots, s_k^\#$ induce a partition of the space of values for the program variables, and the ranking functions $f_1^\#, \dots, f_k^\#$ are affine functions of the program variables.

The concretization function $\gamma \in (\mathcal{S}^\# \mapsto \mathcal{F}^\#) \mapsto (\mathcal{S} \hookrightarrow \mathbb{O})$ is applied piecewise and maps an abstract function to a partial function from program states to ordinals:

$$\gamma(s^\# \mapsto \bot_{\mathsf{F}}) = \dot{\emptyset}$$
$$\gamma(s^\# \mapsto f^\#) = \lambda s \in \gamma_{\mathsf{S}}(s^\#).\ f^\#(s(x_1), \dots, s(x_n))$$
$$\gamma(s^\# \mapsto \top_{\mathsf{F}}) = \dot{\emptyset}$$

where $\dot{\emptyset}$ denotes the totally undefined function, and the function $\gamma_{\mathsf{S}} \in \mathcal{S}^\# \mapsto \mathfrak{P}(\mathcal{S})$ maps an abstract state to the corresponding set of program states.

The domain operators for the abstract approximation order $\sqsubseteq$, the abstract computational order $\preccurlyeq$ and the abstract join $\sqcup$ rely on a partition unification algorithm that, given two abstract functions $v_1^\#$ and $v_2^\#$, modifies the partitions on which they are defined, into a common refined partition of the space of values for each program variable. In particular, since the partitions are determined by intervals with constant bounds, the unification simply introduces new bounds consequently splitting intervals in both partitions. Then, the binary operators can be applied piecewise: the abstract orders, first compare the abstract states on which each function is defined, and then compare the values of the ranking functions on each abstract state; the join operator $\sqcup$ reuses the convex-hull of polyhedra [5].

The widening operator $\triangledown$ prevents the number of pieces of an abstract function from growing indefinitely. First, it performs a partition unification that keeps only the interval

bounds occurring in the first abstract function. Then, it widens the functions piecewise, reusing the convex-hull and the widening of polyhedra.

In order to handle assignments, the abstract domain is equipped with an operation to substitute an arithmetic expression for a variable within an affine function. An assignment is carried on piecewise and independently on each abstract state and each ranking function. Then, the resulting covering induced by the abstract states is refined to obtain a partition.

Finally, to deal with tests, the abstract domain merely applies piecewise to each abstract state the abstract filter operator from the intervals domain.

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 equals 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 with join and widening around loops [1].

Thanks to the soundness of all abstract operators, 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 implemented a research prototype static analyzer [9], based on our abstract domain of piecewise-defined affine ranking functions, and we have used it to analyze programs written in a small non-deterministic imperative language. The prototype is written in OCaml, and the operators from the intervals and convex polyhedra abstract domains are provided by the Apron library [7].

The analysis proceeds by structural induction on the program syntax, iterating loops until an abstract fix-point is reached. In case of nested loops, a fix-point on the inner loop is computed for each iteration of the outer loop, following [1].

3 Future Work

As might be expected, the implemented domain has a limited expressiveness that translates into an imprecision of the analysis especially in the case of nested loops (and, in general, of programs with non-linear complexity). For this reason, we would like to design other abstract domains, based on more sophisticated abstract states and on non-linear ranking functions such as polynomials or exponentials.

Piecewise-Defined Non-Linear Ranking Functions Let us consider the program in Figure 3: it is the (skeleton of) the Bubble Sort algorithm for an array of length 10, once we have removed all tests and assignments on the array. Since the program has a polynomial time complexity, we need non-linear (polynomial) ranking functions to prove its termination.

```
int : x1, x2
while 1(x1 <= 10) do
  2x2 := 10
  while 3(x2 > 1) do
    4x2 := x2 - 1
  od5
  6x1 := x1 + 1
od7
```

Figure 3 Bubble Sort

Figure 4 illustrates the iterates of the backward invariance analysis limited at program control point 1. At the third iteration, the analysis tries to synthesize an affine ranking function for the program. However, such function is not a fix-point: at the next iteration, for $x_1 \leq 8$, we obtain an affine function $f_2(x_1, x_2) = -24x_1 + 259$ with greater slope than $f_1(x_1, x_2) = -22x_1 + 243$; this manifests the need for a polynomial function and, in particular, it leads to the parabola $f(x_1, x_2) = \frac{1}{2}x_1^2 - 31x_1 + \frac{567}{2}$ tangent to both $f_1(x_1, x_2)$ and $f_2(x_1, x_2)$ and passing through

	1
1st iteration	$f(x_1,x_2)=\begin{cases}1 & x_1>10\\ \bot & x_1\leq 10\end{cases}$
2nd iteration	$f(x_1,x_2)=\begin{cases}1 & x_1>10\\ 23 & x_1=10\\ \bot & x_1\leq 9\end{cases}$
3rd iteration	$f(x_1,x_2)=\begin{cases}1 & x_1>10\\ 23 & x_1=10\\ -22x_1+243 & x_1\leq 9\end{cases}$
4th iteration	$f(x_1,x_2)=\begin{cases}1 & x_1>10\\ 23 & x_1=10\\ 43 & x_1=9\\ -24x_1+259 & x_1\leq 8\end{cases}$
4th/5th iteration	$f(x_1,x_2)=\begin{cases}1 & x_1>10\\ 23 & x_1=10\\ \frac{1}{2}x_1^2-31x_1+\frac{567}{2} & x_1\leq 9\end{cases}$

Figure 4 Bubble Sort Analysis.

the point $x_1 = 9$ of $f_1(x_1) = -22x_1 + 243$. At the fifth iteration, a fix-point is reached, proving program termination for all values of x_1 and x_2. ◀

It also remains to investigate the possibility of structuring computations as suggested by [4]. In addition, we plan to extend our research to proving other liveness properties.

References

1 François Bourdoncle. Efficient Chaotic Iteration Strategies with Widenings. In *FMPA*, pages 128–141, 1993.

2 Patrick Cousot and Radhia Cousot. Static Determination of Dynamic Properties of Programs. In *Symposium on Programming*, pages 106–130, 1976.

3 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.

4 Patrick Cousot and Radhia Cousot. An Abstract Interpretation Framework for Termination. In *POPL*, pages 245–258, 2012.

5 Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*, pages 84–96, 1978.

6 Robert W. Floyd. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

7 Bertrand Jeannet and Antoine Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV*, pages 661–667, 2009.

8 Andreas Podelski and Andrey Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI*, pages 239–251, 2004.

9 Caterina Urban. FuncTion. `http://www.di.ens.fr/~urban/FuncTion.html`.

10 Caterina Urban. The Abstract Domain of Segmented Ranking Functions. In *SAS*, pages 43–62, 2013.

Partial Status for KBO

Akihisa Yamada[1], Keiichirou Kusakari[1], and Toshiki Sakabe[1]

1 Graduate School of Information Science, Nagoya University, Japan

Abstract

We propose an extension of the Knuth-Bendix order (KBO) called KBO with partial status. A standard status indicates permutation of arguments to each function symbol, but we extend them to allow some arguments to be ignored. This idea is similar to the argument filtering, but benefits of these methods are independent and hence can be combined. In addition, we introduce further refinements of KBO that become possible by partial status. Significance of the proposed method is verified through experiments.

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

Keywords and phrases term rewriting, termination, Knuth-Bendix order

1 Introduction

Reduction orders are used to prove termination of term rewrite systems (TRSs). The *Knuth-Bendix order* (KBO) [3] is a classical example of reduction orders.

The *dependency pair (DP) framework* (e.g. [2]) significantly enhances the method of reduction orders. In the DP framework, dependencies between rewrite rules are analyzed. Then each cycle of dependency is shown to be *finite* using a *reduction pair* $\langle \succsim, \succ \rangle$, which is typically designed from a reduction order by applying *argument filtering*. However, argument filtering is not always helpful as the following example illustrates:

▶ **Example 1.** Consider the following set of constraints:

$$\mathsf{F}(\mathsf{s}(x)) \succ \mathsf{F}(\mathsf{p}(\mathsf{s}(x))) \qquad \mathsf{p}(\mathsf{s}(x)) \succsim x$$

In order to satisfy the first constraint by KBO (or any other *simplification order*), the argument of p must be filtered. However, the second constraint cannot be satisfied under such an argument filtering.

In this note, we propose a reduction pair that can satisfy the above constraints by generalizing KBO *with status* [5]. Usually, a status assigns a new position to every argument of a function symbol. When defining a reduction pair, however, not every argument must be assigned a new position, but some may be ignored. We say such a status is *partial*. The difference between a partial status and an argument filter with standard (i.e. total) status may look subtle; indeed, a trivial definition of LPO with partial status should be subsumed by LPO with argument filters and total status. On the other hand, KBO benefits from partial status because of weights of ignored arguments, which would be lost if those arguments were *filtered* beforehand by an argument filter. Indeed, the constraints in Example 1 are satisfied by KBO with partial status defined in Section 2. We further introduce two refinements that become possible using partial status. Then we demonstrate the significance of our approach through experiments.

2 KBO with Partial Status

Below we define the notion of partial status and the KBO reduction pair.

▶ **Definition 2.** A *partial status function* σ is a mapping that assigns for n-ary symbol f a list $[i_1, \dots, i_{n'}]$ of distinct positions in $\{1, \dots, n\}$.

We write $\vec{s}_{\sigma(f)}$ to denote the sequence $s_{i_1}, \dots, s_{i_{n'}}$, where $\sigma(f) = [i_1, \dots, i_{n'}]$.

▶ **Definition 3** (KBO with partial status)**.** Let $\gtrsim_{\mathcal{F}}$ be a quasi-precedence, σ a partial status function and $\langle w, w_0\rangle$ a *weight function*, i.e. $w : \mathcal{F} \to \mathbb{N}$, $w_0 > 0$ and $w(c) \geq w_0$ for every constant $c \in \mathcal{F}$. The weight $w(s)$ of a term s is defined as usual:

$$w(s) := \begin{cases} w_0 & \text{if } s \in \mathcal{V} \\ w(f) + \sum_{i=1}^{n} w(s_i) & \text{if } s = f(\vec{s}_n) \end{cases}$$

The *Knuth-Bendix order pair* $\langle \succsim_{\mathrm{KBO}}, \succ_{\mathrm{KBO}}\rangle$ is defined recursively as follows: $s \succsim (\text{resp. } \succ)_{\mathrm{KBO}}$ t iff $|s|_x \geq |t|_x$ for all $x \in \mathcal{V}$ and either

1. $w(s) > w(t)$, or
2. $w(s) = w(t)$ and either
   - **a.** $s = f_1(\dots f_k(t)\dots)$, $\sigma(f_1) = \dots = \sigma(f_k) = [1]$ and $t \in \mathcal{V}$ for some $k \geq (\text{resp. } >)$ 0, or
   - **b.** $s = f(\vec{s}_n)$, $t = g(\vec{t}_m)$ and either
     - **i.** $f >_{\mathcal{F}} g$, or
     - **ii.** $f \sim_{\mathcal{F}} g$ and $[\vec{s}_{\sigma(f)}] \succsim (\text{resp. } \succ)^{\mathsf{lex}}_{\mathrm{KBO}} [\vec{t}_{\sigma(g)}]$.

Here $\succ^{\mathsf{lex}}_{\mathrm{KBO}}$ denotes the lexicographic extension of $\succ_{\mathrm{KBO}}$ modulo $\succsim_{\mathrm{KBO}}$.

The major difference to the standard KBO (e.g. [6]) is case (**2a**), where we exclude the case if $\sigma(f_i) = [\,]$ for some f_i. Because of this modification, the *admissibility* constraint of KBO can be eased as follows:

▶ **Definition 4.** A weight function w is said to be *admissible for* $\gtrsim_{\mathcal{F}}$ *and* σ iff every unary symbol f s.t. $w(f) = 0$ and $\sigma(f) = [1]$ is greatest in $\gtrsim_{\mathcal{F}}$, i.e. $f \gtrsim_{\mathcal{F}} g$ for every $g \in \mathcal{F}$.

In the remainder of this note, we always assume admissibility. Note that a unary symbol f of weight 0 need not be greatest in $\gtrsim_{\mathcal{F}}$, if $\sigma(f) = [\,]$.

▶ **Example 5.** Consider again the constraints in Example 1. Suppose w, $>_{\mathcal{F}}$ and σ satisfy $w(\mathsf{s}) > w(\mathsf{p}) = 0$, $\sigma(\mathsf{s}) = [1]$, $\sigma(\mathsf{p}) = [\,]$, and $\mathsf{s} >_{\mathcal{F}} \mathsf{p}$. Then, $\mathsf{F}(\mathsf{s}(x)) \succ_{\mathrm{KBO}} \mathsf{F}(\mathsf{p}(\mathsf{s}(x)))$ because of cases (**2b–ii**) and (**2b–i**), and $\mathsf{p}(\mathsf{s}(x)) \succsim_{\mathrm{KBO}} x$ because of case (**1**).

Note that in the above example, it also holds that $\mathsf{s}(x) \succ_{\mathrm{KBO}} \mathsf{p}(\mathsf{s}(x))$. Hence, $\succ_{\mathrm{KBO}}$ is not a simplification order anymore, or not even a reduction order. Nonetheless, we can show the following result which is sufficient for the DP framework:

▶ **Theorem 6.** *The KBO pair* $\langle \succsim_{\mathrm{KBO}}, \succ_{\mathrm{KBO}}\rangle$ *is a reduction pair.*

Due to lack of space, we only present a proof for well-foundedness of $\succ_{\mathrm{KBO}}$. We prove the following auxiliary lemma first:

▶ **Lemma 7.** *If* $\vec{s}_{\sigma(f)} \in \mathsf{SN}(\succ_{\mathrm{KBO}})$ *and* $s \succ_{\mathrm{KBO}} t$*, then* $t \in \mathsf{SN}(\succ_{\mathrm{KBO}})$.

Proof. By induction on the quadruple $\langle w(s), f, [\vec{s}_{\sigma(f)}], |t|\rangle$, which is ordered by the lexicographic composition of $>$, $>_{\mathcal{F}}$, $\succ^{\mathsf{lex}}_{\mathrm{KBO}}$ and $>$. Trivially, it is sufficient to consider $t = g(\vec{t}_m)$. Let $[j_1, \dots, j_{m'}] = \sigma(g)$.

- Suppose $w(s) > w(t)$. Then we have $w(s) > w(t) \geq w(t_{j_k})$ and hence $s \succ_{\mathrm{KBO}} t_{j_k}$ for every $k \in \{1, \ldots, m'\}$. By the induction hypothesis on the fourth component, we obtain $t_{j_k} \in \mathsf{SN}(\succ_{\mathrm{KBO}})$. Thus for arbitrary u s.t. $t \succ_{\mathrm{KBO}} u$, the induction hypothesis on the first component yields $u \in \mathsf{SN}(\succ_{\mathrm{KBO}})$.
- Suppose $w(s) = w(t)$. First we show $t_{j_k} \in \mathsf{SN}(\succ_{\mathrm{KBO}})$ for every $k \in \{1, \ldots, m'\}$. It is trivial if no such k exists, i.e. if $\sigma(g) = [\,]$. Hence suppose $\sigma(g) \neq [\,]$.
  - If $w(t) = w(t_{j_k})$, then g must be unary with $w(g) = 0$ and $\sigma(g) = [1]$. Because of the admissibility, only case (**2b–ii**) can be applied for $s \succ_{\mathrm{KBO}} g(t_1) = t$. Hence, we obtain $s_{i_1} \succsim_{\mathrm{KBO}} t_1$ and thus $t_1 \in \mathsf{SN}(\succ_{\mathrm{KBO}})$, since $s_{i_1} \in \mathsf{SN}(\succ_{\mathrm{KBO}})$.
  - If $w(t) > w(t_{j_k})$, then $s \succ_{\mathrm{KBO}} t_{j_k}$ by case (**1**). By the induction hypothesis on the fourth component, $t_j \in \mathsf{SN}(\succ_{\mathrm{KBO}})$.

  Now let us consider arbitrary u s.t. $t \succ_{\mathrm{KBO}} u$. Since we have either $f >_{\mathcal{F}} g$ or $f \sim_{\mathcal{F}} g$ and $[\vec{s}_{\sigma(f)}] \succ^{\mathsf{lex}}_{\mathrm{KBO}} [\vec{t}_{\sigma(g)}]$, $\langle w(s), f, [\vec{s}_{\sigma(f)}], |t| \rangle$ is greater than $\langle w(t), g, [\vec{t}_{\sigma(g)}], |u| \rangle$. Hence, the induction hypothesis yields $u \in \mathsf{SN}(\succ_{\mathrm{KBO}})$. ◀

▶ **Lemma 8.** *The relation $\succ_{\mathrm{KBO}}$ is well-founded.*

Proof. Let us show $s \in \mathsf{SN}(\succ_{\mathrm{KBO}})$ for every term s by induction on $|s|$. Suppose $s = f(\vec{s}_n) \succ_{\mathrm{KBO}} t$. By the induction hypothesis, we have $\vec{s}_n \in \mathsf{SN}(\succ_{\mathrm{KBO}})$ and thus $\vec{s}_{\sigma(f)} \in \mathsf{SN}(\succ_{\mathrm{KBO}})$. Hence by Lemma 7, we get $t \in \mathsf{SN}(\succ_{\mathrm{KBO}})$. ◀

3 Refinements

In this section, we refine $\succsim_{\mathrm{KBO}}$ in order to encompass the *polynomial order (POLO)* that is induced by the weight function.

▶ **Definition 9.** The *empty status function* is the partial status σ s.t. $\sigma(f) = [\,]$ for all $f \in \mathcal{F}$.

KBO induced by the quasi-precedence $\succsim_{\mathcal{F}} = \mathcal{F}^2$ and the empty status is quite similar to POLO induced by the interpretation $\mathcal{A}$: $f_{\mathcal{A}}(\vec{x}_n) = w(f) + \sum_{i=1}^{n} x_i$. However, the latter is slightly more powerful; the constraint $x \succsim \mathsf{p}(x)$ can be satisfied by POLO s.t. $\mathsf{p}_{\mathcal{A}}(x) = x$, but the weak part of KBO cannot satisfy this constraint even if $w(\mathsf{p}) = 0$.

In [6], $\succsim_{\mathrm{KBO}}$ is refined s.t. $x \succsim_{\mathrm{KBO}} c$ for a minimal constant c. In our setting, a similar refinement can be applied for non-constants:

▶ **Proposition 10.** *Let $s \in \mathcal{V}$ and $t = g(\vec{t}_m)$ s.t.*
- *$|t|_s \leq 1$ and $|t|_x = 0$ for every $x \in \mathcal{V} \setminus \{s\}$,*
- *$w(t) = w_0$,*
- *g is minimal w.r.t. $\succsim_{\mathcal{F}}$, and*
- *$\sigma(g) = [\,]$.*

Then for any term $s' = f(\vec{s}_n)$, $s' \succsim_{\mathrm{KBO}} t[s \mapsto s']$. ◀

Hence, we refine $s \succsim_{\mathrm{KBO}} t$ by adding the following subcase for case (**2**) of Definition 3 (note that the first two conditions above are already satisfied in case (**2**)):

c. $s \succsim_{\mathrm{KBO}} t$ if $s \in \mathcal{V}$ and $t = g(\vec{t}_m)$ s.t. g is minimal w.r.t. $\succsim_{\mathcal{F}}$ and $\sigma(g) = [\,]$.

▶ **Example 11.** Consider the following set of constraints:

$$\mathsf{F}(\mathsf{s}(x), y) \succ \mathsf{F}(\mathsf{p}(\mathsf{s}(x)), \mathsf{p}(y)) \qquad \mathsf{F}(x, \mathsf{s}(y)) \succsim \mathsf{F}(\mathsf{p}(x), \mathsf{p}(\mathsf{s}(y))) \qquad \mathsf{p}(\mathsf{s}(x)) \succsim x$$

Let $\sigma(\mathsf{p}) = [\,]$, $\sigma(\mathsf{F}) = [1]$, $w(\mathsf{s}) > w(\mathsf{p}) = 0$ and p be minimal w.r.t. $\succsim_{\mathcal{F}}$. As analogous to Example 5, the first and the third constraints are satisfied. For the second constraint, it

yields $x \gtrsim \mathsf{p}(x)$, for which case (**c**) of the refined $\succsim_{\mathrm{KBO}}$ applies. Note that the argument of p cannot be filtered by an argument filter, because of the third constraint. Hence the refinement of [6] does not work for this example.

We can also refine $\succsim_{\mathrm{KBO}}$ when the right-hand side is a variable.

▶ **Proposition 12.** *Let $s = f(\vec{s}_n)$ and $t \in \mathsf{Var}(s)$ s.t.*
- *$\sigma(f) = [\,]$, and*
- *for any $g \in \mathcal{F}$, $f \gtrsim_{\mathcal{F}} g$ if $\sigma(g) = [\,]$ and $f >_{\mathcal{F}} g$ otherwise.*

Then for any term $t' = g(\vec{t}_m)$, $s[t \mapsto t'] \succsim_{\mathrm{KBO}} t'$. ◀

Hence, we refine $s \succsim_{\mathrm{KBO}} t$ by adding the following subcase for case (**2**):

d. $s \succsim_{\mathrm{KBO}} t$ if $s = f(\vec{s}_n)$ and $t \in \mathcal{V}$ s.t. $\sigma(f) = [\,]$ and for any $g \in \mathcal{F}$, $f \gtrsim_{\mathcal{F}} g$ if $\sigma(g) = [\,]$ and $f >_{\mathcal{F}} g$ otherwise.

It is easy to prove the following result:

▶ **Theorem 13.** *Let $\langle w, w_0 \rangle$ be a weight function, σ the empty status function and $\gtrsim_{\mathcal{F}} = \mathcal{F}^2$. Then the refined KBO is equivalent to POLO[1] induced by the carrier set $\{n \geq w_0\}$ and the interpretation $f_{\mathcal{A}}(\vec{x}_n) := w(f) + \sum_{i=1}^{n} x_i$.* ◀

▶ **Example 14.** Consider the following set of constraints:

$$\mathsf{F}(\mathsf{g}(\mathsf{h}(x))) \succ \mathsf{F}(\mathsf{h}(\mathsf{g}(\mathsf{g}(\mathsf{h}(\mathsf{h}(x)))))) \qquad\qquad \mathsf{g}(\mathsf{h}(x)) \gtrsim x$$

Because of the second constraint, arguments of g and h cannot be filtered. Then the first constraint requires $w(\mathsf{g}) = w(\mathsf{h}) = 0$ and moreover one of the following alternatives to hold:
- $\sigma(\mathsf{g}) = \sigma(\mathsf{h}) = [\,]$ and $\mathsf{g} >_{\mathcal{F}} \mathsf{h}$: In this case, the second constraint can be satisfied only if $\succsim_{\mathrm{KBO}}$ is refined by case (**d**).
- $\sigma(\mathsf{h}) = [1]$ and $\mathsf{g} >_{\mathcal{F}} \mathsf{h}$: This case is not admissible.
- $\sigma(\mathsf{g}) = [1]$, $\sigma(\mathsf{h}) = [\,]$ and $\mathsf{g} \gtrsim_{\mathcal{F}} \mathsf{h}$: In this case the second constraint cannot be satisfied.

Hence the set of constraints can be satisfied by KBO with partial status only if it is refined by case (**d**). Note that POLO (and LPO) cannot satisfy the set of constraints, since the first rule is not simply terminating and neither g nor h may have 0-coefficient.

4 Experiments and Future Work

We implemented our method via an SMT encoding that extends [9]. For the DP framework, we implemented a simple estimation of *dependency graphs*, and *strongly connected components* are sequentially processed in order of size where smaller ones are precedent. We also implemented *usable rules* w.r.t. argument filters following the encoding proposed in [1].

The experiments[2] are run on a server equipped with two quad-core Intel Xeon W5590 processors running at a clock rate of 3.33GHz and 48GB of main memory, though only one thread of SMT solver runs at once. As the SMT solver, we choose z3 4.3.1. The test set of termination problems are the 1463 TRSs from the TRS Standard category of TPDB 8.0.6[3] and 1315 from the SRS Standard category. Timeout is set to 60 seconds.

[1] Note that $c_{\mathcal{A}} \geq w_0 > 0$ is required for every constant c.
[2] Detailed results are available at `http://www.sakabe.i.is.nagoya-u.ac.jp/~ayamada/WST2013/`.
[3] The Termination Problems Data Base. `http://termination-portal.org/wiki/TPDB`.

Table 1 Experiments

Method	Solo			Combination			Combination (SRS)		
	yes	T.O.	time	yes	T.O.	time	yes	T.O.	time
KBO	439	3	1079.23	565	5	1008.69	117	467	28654.39
Def. 3	463	4	1255.77	583	5	1036.46	120	468	28780.43
Prop. 10	464	4	1305.19	584	5	1044.68	120	468	28853.21
Prop. 12	464	4	1275.01	584	5	1036.92	121	474	29356.51
Prop. 10+12	465	4	1323.83	585	5	1049.84	121	474	29380.88

In Table 1, the 'Method' field indicates the reduction pair processor used. 'KBO' row is the standard KBO with (total) status and 'Def. 3' is the KBO with partial status. 'Prop. 10' and 'Prop. 12' applies the refinement of Proposition 10 and Proposition 12, resp. All the methods are with quasi-precedences, argument filters and usable rules.

'Solo' field only applies the reduction pair processor indicated by the 'Method' field. Partial status gives measurable increase in the number of successes (indicated by 'yes' field), though the efficiency is affected ('time' field). Each refinement of Section 3 gains one success with probably acceptable increase in runtime.

'Combination' field applies several reduction pair processors first: It applies the linear POLO (with/without max) with coefficient at most 1, and then LPO with quasi-precedence and status. In this situation partial status becomes more attractive; the increase in number of successes remains measurable, while increase of runtime gets dramatically smaller. For 'Combination (SRS)' field, it applies linear POLO before the indicated processor.

Despite the benefit observed in our experimental implementation, all the TPDB examples our tool proved terminating are also proved terminating by existing termination tools such as AProVE. Our next task is to apply partial status to other extensions of KBO (e.g. [4, 7, 8]) to further increase the number of successful termination proofs.

References

1 M. Codish, P. Schneider-Kamp, V. Lagoon, R. Thiemann, and J .Giesl. SAT solving for argument filterings. In *Proc. LPAR '06*, volume 4246 of *LNCS*, pages 30–44, 2006.

2 J. Giesel, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

3 D.E. Knuth and P. Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, New York, 1970.

4 M. Ludwig and U. Waldmann. An extension of the Knuth-Bendix ordering with LPO-like properties. In *Proc. LPAR '07*, volume 4790 of *LNAI*, pages 348–362, 2007.

5 J. Steinbach. Extensions and comparison of simplification orders. In *Proc. RTA '89*, volume 355 of *LNCS*, pages 434–448, 1989.

6 C. Sternagel and R. Thiemann. Formalizing Knuth-Bendix orders and Knuth-Bendix completion. In *Proc. RTA '13*, volume 21 of *LIPIcs*, pages 287–302, 2013.

7 S. Winkler, H. Zankl, and A. Middeldorp. Ordinals and Knuth-Bendix orders. In *Proc. LPAR '11*, volume 7180 of *LNCS Advanced Research in Computing and Software Science*, pages 420–434, 2012.

8 A. Yamada, K. Kusakari, and T. Sakabe. Unifying the Knuth-Bendix, recursive path and polynomial orders. In *Proc. PPDP '13*, 2013. to appear.

9 H. Zankl, N. Hirokawa, and A. Middeldorp. KBO orientability. *Journal of Automated Reasoning*, 43(2):173–201, 2009.