10th International Workshop on Termination (WST 2009)

Leipzig, June 3–5, 2009

Alfons Geser and Johannes Waldmann (editors)

Hochschule für Technik, Wirtschaft und Kultur
Leipzig, Germany

Preface

This volume contains the proceedings of the *Tenth International Workshop on Termination (WST 09)*, to be held June 3-5, 2009 in Leipzig, Germany.

The workshop brings together researchers and practitioners that are interested in methods and techniques related to termination of computations. This includes all models and fields of computation, like rewriting systems, programming paradigms and languages, transitions systems, etc. Workshop contributions report on basic research as well as on applications and implementations.

The Workshop on Termination series started in 1993 in St. Andrews, and continued biannually (more or less). In recent years, the workshop had been attached to conferences (FLOC, RDP), but WST09 is held as a separate event again, returning to the original tradition. The intention is to have more time for presentations and discussions.

The first day of the workshop focuses on the Termination Competition and includes a keynote talk by Morgan Deters (UPC, Barcelona, Spain) on *The SMT∗ Platform: Design, Implementation, and Experience*.

There were a surprising 27 submissions to WST 09, from Austria, Belgium, The Czech Republic, Denmark, France, Germany, The Netherlands, Spain, Japan, and the USA. The submissions were reviewed by the program committee consisting of

Frederic Blanqui	INRIA, France, and Tsinghua U, China,
Byron Cook	Microsoft Research, UK,
Alfons Geser (chair)	HTWK Leipzig, Germany,
Michael Hanus	CAU Kiel, Germany,
Janis Voigtländer	TU Dresden, Germany.

We would like to take this opportunity to thank *Medienstiftung der Sparkasse Leipzig* for providing the facilities at *Mediencampus Villa Ida*, and the members of the program committee for their excellent and conscientious job of refereeing all the papers. Paper submission, refereeing, author notification was all managed by Andrei Voronkov's EasyChair system.

Leipzig, Germany	Alfons Geser and Johannes Waldmann
May, 2009	PC chair and local organizer

Contents

Proving Termination of Programs with Second-Order Recursion

Markus Aderhold
Technische Universität Darmstadt
Darmstadt, Germany
`aderhold@pm.tu-darmstadt.de`

1 Introduction

Many algorithms on data structures such as *terms* (finitely branching trees) are naturally implemented by second-order recursion: A first-order procedure f passes itself as an argument to a second-order procedure like *map*, *every*, *foldl*, *foldr*, etc. to recursively apply f to the direct subterms of a term.

Verifying total correctness of a program involves (i) proving termination of the program and (ii) verifying partial correctness (i. e., proving that the program computes results that are correct wrt. a specification). Consequently, verification tools usually try to prove termination of a given program first and then synthesize induction axioms from terminating procedures to facilitate subsequent inductive proofs.

Automated termination analysis for programs with second-order recursion is a non-trivial problem: In the higher-order theorem provers Isabelle [6, 7] and PVS [8] the user needs to assist the system to prove termination in these cases. The method we propose solves typical termination problems automatically.

Our approach extends the method of *argument-bounded functions* [10, 12] that is used, for instance, in the semi-automated verifier √`eriFun` [11].[1] This method automates termination analysis of first-order programs and allows for the synthesis of suitable induction axioms based on the insight gained by termination analysis [9]. We extend this approach in two respects: Firstly, an extended notion of *argument-boundedness* also considers *components* of types. Secondly, the novel notion of *call-boundedness* facilitates automated termination analysis of procedures with second-order recursion.

2 Programming Language

We consider an extension of √`eriFun`'s functional programming language $\mathscr{L}$ 1.0 that offers definition principles for freely generated polymorphic data types, for first-order and second-order procedures, and for statements about the data types and procedures. The language roughly corresponds to the second-order fragment of Haskell with strict evaluation [2].

A *base type* is a type variable $@A$ or an expression of the form $str[\tau_1,\ldots,\tau_k]$, where $\tau_1,\ldots,\tau_k$ are base types and *str* is a k-ary type constructor ($k \geq 0$). A *type* is a base type or a *function type* of the form $\tau_1 \times \ldots \times \tau_k \to \tau$ for base types $\tau_1,\ldots,\tau_k,\tau$. *Type constructors* are defined by expressions of the form "`structure` $str[@A_1,\ldots,@A_k]$ <= $\ldots$, $cons(sel_1 : \tau_1,\ldots,sel_n : \tau_n)$, $\ldots$" for base types τ_j. Each *cons* is called a *data constructor* and the sel_j are called *selectors*. Expressions of the form $?cons(t)$ are used as shorthand notation for $t = cons(sel_1(t),\ldots,sel_n(t))$. We address *type symbols* (i. e., type constructors and type variables) in a base type by their *position* $\pi \in Pos(\tau) \subseteq \mathbb{N}^*$: $@A|_\varepsilon := @A$, $str[\tau_1,\ldots,\tau_k]|_\varepsilon := str$, and $str[\tau_1,\ldots,\tau_k]|_{h\pi'} := \tau_h|_{\pi'}$ for $h \in \{1,\ldots,k\}$.

Example 1. Figure 1(a) defines data constructors 0, $^+(\ldots)$ (denoting the successor function), ø (denoting the empty list), and "::". Selector $^-(\ldots)$ denotes the predecessor function. Procedure *last* contains a so-called *context requirement* $k \neq ø$ that needs to be satisfied for each function call. In Fig. 1(b), *every* is a second-order procedure that checks if $p(x)$ is satisfied for all elements x of list k. Procedure *groundterm* uses *second-order recursion* to check if a term t does not contain any variables. ◇

The operational semantics of a program P is defined by a partial mapping $eval_P : T(\Sigma(P)) \mapsto \mathbb{V}(P)$. This call-by-value interpreter maps ground terms $t \in T(\Sigma(P))$ over the signature $\Sigma(P)$ of program P to values $q \in \mathbb{V}(P)$, see [2]. For a ground base type τ, $\mathbb{V}(P)_\tau$ consists of all constructor ground terms of

[1] See `http://www.verifun.org` for further information on √`eriFun`.

```
(a) structure $\mathbb{N}$ <= 0, $^{+}(^{-}:\mathbb{N})$
    structure $list[@A]$ <= $\emptyset$, $::(hd : @A,\ tl : list[@A])$
    procedure $last(k : list[@A]) : @A$ <=
      assume $k \neq \emptyset$; $if\ tl(k) = \emptyset\ then\ hd(k)\ else\ last(tl(k))\ end$
(b) structure $variable.symbol$ <= $variable(varID : \mathbb{N})$
    structure $function.symbol$ <= $func(funcID : \mathbb{N})$
    structure $term$ <=
      $var(vsym : variable.symbol)$,
      $apply(fsym : function.symbol,\ args : list[term])$
    procedure $every(p : @A \to bool,\ k : list[@A]) : bool$ <=
      $if\ k = \emptyset\ then\ true\ else\ if\ p(hd(k))\ then\ every(p,\ tl(k))\ else\ false\ end\ end$
    procedure $groundterm(t : term) : bool$ <=
      $if\ ?var(t)\ then\ false\ else\ every(groundterm,\ args(t))\ end$
```

Figure 1: A functional program with second-order recursion in procedure *groundterm*

type τ. For a ground function type τ, $\mathbb{V}(P)_\tau$ consists of all closed λ-expressions of type τ that contain only calls of terminating functions. A function f *terminates* iff $eval_P(f(q_1,\dots,q_n))$ is defined for all values $q_1,\dots,q_n \in \mathbb{V}(P)$. Program P terminates iff all functions f defined in P terminate.

3 Termination Analysis

A procedure `procedure` $f(x\!:\!\tau) : \tau'$ terminates if parameter x gets structurally smaller in each recursive call. For instance, procedure *every* terminates, because list $tl(k)$ is structurally smaller than list k. Formally, function $tl : list[@A] \to list[@A]$ is *argument-bounded*, because the size of $tl(k)$ is bounded by the size of argument k. The structural size of a value is computed by the *size measure*:

Size measure. The *size measure* $\#_\tau : \mathbb{V}(P)_\tau \times Pos(\tau) \to \mathbb{N}$ is uniformly defined for each ground base type $\tau = str[\tau'_1,\dots,\tau'_k]$. For a constructor ground term t and some $\pi \in Pos(\tau)$, $\#_\tau(t,\pi)$ counts how often the data constructors of the type constructor at position π in τ occur in term t. Data constructors are counted with a certain weight, see [2] for details and the formal definition.

Example 2. For $\tau := list[\mathbb{N}]$, $\#_{list[\mathbb{N}]}(t,\varepsilon)$ counts the occurrences of *list*-constructors "$\emptyset$" and "::" in t, whereas $\#_{list[\mathbb{N}]}(t,1)$ counts the occurrences of $\mathbb{N}$-constructors 0 and $^{+}(\dots)$ in t; if $t = x_1 :: \dots :: x_n :: \emptyset$, then $\#_{list[\mathbb{N}]}(t,1) = \#_{\mathbb{N}}(x_1,\varepsilon) + \dots + \#_{\mathbb{N}}(x_n,\varepsilon)$. For $\tau := pair[\mathbb{N},\mathbb{N}]$ (cf. Fig. 2), $\#_{pair[\mathbb{N},\mathbb{N}]}(t,1)$ counts the occurrences of $\mathbb{N}$-constructors in $fst(t)$, whereas $\#_{pair[\mathbb{N},\mathbb{N}]}(t,2)$ counts the occurrences of $\mathbb{N}$-constructors in $snd(t)$. For $\tau := term$, $\#_{term}(t,\varepsilon)$ counts the occurrences of *term*-constructors *var* and *apply* in t. $\diamondsuit$

3.1 Termination Proofs via Estimation Proofs

Argument-bounded functions. A function $f : \tau \to \tau'$ with context requirement c_f is (π,ρ)*-argument-bounded* for positions $\pi \in Pos(\tau)$ and $\rho \in Pos(\tau')$ iff (i) τ is a base type with $\tau|_\pi = \tau'|_\rho$ and (ii) $\#(q,\pi) \geq \#(eval_P(f(q)),\rho)$ for all values $q \in \mathbb{V}(P)$ with $eval_P(c_f[q]) = true$.

Example 3. $^{-}(\dots) : \mathbb{N} \to \mathbb{N}$ is $(\varepsilon,\varepsilon)$-argument-bounded. $hd : list[@A] \to @A$ is $(1,\varepsilon)$-argument-bounded: The size of the first element of a non-empty list k is bounded by the sum of the sizes of all elements in k. Similarly, procedure *last* is $(1,\varepsilon)$-argument-bounded. $tl : list[@A] \to list[@A]$ is $(\varepsilon,\varepsilon)$-argument-bounded, as $tl(k)$ contains fewer *list*-constructors "::" than k. tl is also $(1,1)$-argument-bounded, as $tl(k)$ contains a subset of the elements in k. Finally, $args : term \to list[term]$ is $(\varepsilon,1)$-argument-bounded. $\diamondsuit$

```
structure $pair[@A, @B] <= mkpair(fst : @A,\ snd : @B)$
procedure $split(k\!:\!list[@A]) : pair[list[@A], list[@A]] <= \ldots$
procedure $merge(k,l\!:\!list[\mathbb{N}]) : list[\mathbb{N}] <= \ldots$
procedure $msort(k\!:\!list[\mathbb{N}]) : list[\mathbb{N}] <=$
  $if\ k{=}\emptyset\ then\ \emptyset\ else\ if\ tl(k){=}\emptyset\ then\ k$
  $else\ merge(msort(fst(split(k))), msort(snd(split(k))))\ end\ end$
```

Figure 2: Implementation of Mergesort (sketch)

For a (π,ρ)-argument-bounded function $f : \tau \to \tau'$ with context requirement c_f, the (π,ρ)-*difference function* $\Delta_f^{\pi,\rho} : \tau \to bool$ for f returns *true* iff the size inequality is strict: For all values $q \in \mathbb{V}(P)$ with $eval_P(c_f[q]) = true$ we thus have $eval_P(\Delta_f^{\pi,\rho}(q)) = true \iff \#(q,\pi) > \#(eval_P(f(q)),\rho)$. For instance, $\Delta_{tl}^{\varepsilon,\varepsilon}(k)$ returns *true* iff $k{\neq}\emptyset$, and $\Delta_{args}^{\varepsilon,1}(t)$ returns *true* iff $?apply(t)$.

Estimation proofs. An inequality $\#(t_1,\pi_1) \geq \#(t_2,\pi_2)$ for terms t_1,t_2 and positions π_1,π_2 can be shown using the *estimation calculus* defined in [2]. Apart from a proof of $\#(t_1,\pi_1) \geq \#(t_2,\pi_2)$, an estimation proof yields a *difference equivalent* Δ, which is a finite set of literals such that $\bigvee\Delta$ is true iff the inequality is strict. The estimation calculus is parameterized by a *call context* C (which is a finite set of literals that may be assumed as true) and a family $\Gamma_{\pi,\rho}$ of (π,ρ)-argument-bounded function symbols. We write $\vdash_{\Gamma,C} \langle\Delta,(t_1,\pi_1) \succcurlyeq (t_2,\pi_2)\rangle$ iff there exists an estimation proof of $\#(t_1,\pi_1) \geq \#(t_2,\pi_2)$ with difference equivalent Δ. The estimation calculus is decidable and is used to prove argument-boundedness of procedures, to synthesize difference procedures, and to prove termination of procedures.

Example 4. Figure 2 sketches an implementation of Mergesort. Procedure *split* splits list k into two lists that are recursively sorted and then merged together. Hence it is $(\varepsilon,1)$- and $(\varepsilon,2)$-argument-bounded, because the resulting lists are not longer than k. Procedure *msort* terminates, because the arguments of the recursive calls are smaller wrt. the size measure: $\vdash_{\Gamma,C} \langle\Delta_1,(k,\varepsilon) \succcurlyeq (fst(split(k)),\varepsilon)\rangle$ for call context $C = \{k{\neq}\emptyset,\ tl(k){\neq}\emptyset\}$ and $\Delta_1 = \{\Delta_{fst}^{1,\varepsilon}(split(k)),\ \Delta_{split}^{\varepsilon,1}(k)\}$. Similarly, $\vdash_{\Gamma,C} \langle\Delta_2,(k,\varepsilon) \succcurlyeq (snd(split(k)),\varepsilon)\rangle$ for $\Delta_2 = \{\Delta_{snd}^{2,\varepsilon}(split(k)),\ \Delta_{split}^{\varepsilon,2}(k)\}$ and the same call context. The proofs of the *termination hypotheses* $\forall k\!:\!list[\mathbb{N}].\ \bigwedge C \to \bigvee\Delta_i$ for $i = 1,2$ are trivial, because $\bigwedge C$ entails that list k contains at least two elements, so the difference procedures $\Delta_{split}^{\varepsilon,i}(k)$ return *true*. Thus the size inequality is indeed strict. $\diamondsuit$

3.2 Analysis of Second-Order Recursion

Call-bounded procedures. We write $g(q_1,\ldots,q_k) \rhd f(q'_1,\ldots,q'_m)$ iff the evaluation of $g(q_1,\ldots,q_k)$ requires the evaluation of $f(q'_1,\ldots,q'_m)$ in the body of procedure g. A procedure $g(f\!:\!\tau' \to \tau'', x\!:\!\tau) : \tau'''$ is (π,ρ)-*call-bounded* for $\pi \in Pos(\tau)$ and $\rho \in Pos(\tau')$ iff τ is a base type with $\tau|_\pi = \tau'|_\rho$ such that $\#(\mathsf{x},\pi) \geq \#(q',\rho)$ for all $\mathsf{f} \in \mathbb{V}(P)_{\tau'\to\tau''}$ and $\mathsf{x} \in \mathbb{V}(P)_\tau$ with $g(\mathsf{f},\mathsf{x}) \rhd h_1(\ldots) \rhd \ldots \rhd h_n(\ldots) \rhd \mathsf{f}(q')$ for some functions h_i and some $n \geq 0$, where $h_i \neq \mathsf{f}$ for all $i = 1,\ldots,n$. Thus a call-bounded procedure g calls its first-order parameter f only with arguments q' that are bounded by the size of x. For example, procedure *every* is $(1,\varepsilon)$-call-bounded, because parameter p is only called with an argument $x\!:\!@A$ with $\#(k,1) \geq \#(x,\varepsilon)$, which can be easily shown by two estimation proofs [2].

Second-order recursion. If a procedure is defined by second-order recursion using a *call-bounded* second-order procedure, termination can be proved as in the following example:

Example 5. Procedure *groundterm* in Fig. 1 terminates, because (i) procedure *every* is $(1,\varepsilon)$-call-bounded, (ii) $\vdash_{\Gamma,C} \langle\{\Delta_{args}^{\varepsilon,1}(t)\},(t,\varepsilon) \succcurlyeq (args(t),1)\rangle$ for call context $C := \{\neg ?var(t)\}$, and (iii) the termination hypothesis $\forall t\!:\!term.\ \neg ?var(t) \to \Delta^{\varepsilon,1}_{args}(t)$ is true. Thus $\#(t,\varepsilon) > \#(args(t),1) \geq \#(x,\varepsilon)$ for each argument $x\!:\!term$ of a recursive call; the $>$-estimation follows from (ii) and (iii), the $\geq$-estimation from (i). $\diamondsuit$

4 Conclusion

Our method has been implemented in an experimental version of √eriFun. It automatically solves the examples of second-order recursion considered in the literature [6, 7, 8], see [2], and maintains the advantage of the original approach from [10, 12] that "optimized" induction axioms are synthesized by analyzing the termination proofs (i. e., irrelevant premises are removed and certain variables are universally quantified in the induction hypotheses [9]).

In related work on termination analysis, size information about functions (such as argument-boundedness) is encoded by type annotations. For instance, the approach by Barthe et al. [3] can show both termination and argument-boundedness of Quicksort; our method automatically proves termination of Quicksort, but cannot show argument-boundness of that procedure. Work by Abel [1] and Hughes et al. [5] considers type systems that can also be used to ensure *productivity* of algorithms that work on coinductive data structures such as infinite streams. However, neither of the approaches in [1, 3, 5] can show termination of Mergesort, because the type systems do not recognize the cases when *split* returns lists that are strictly smaller than the input list (cf. Fig. 2). The concept of difference functions precisely captures the behavior of *split* so our approach easily shows termination of Mergesort. Giesl et al. [4] present an approach based on the *dependency pair* method. It offers increased flexibility wrt. the size measure, because it does not focus on a structural size measure as in [1, 3, 5] and in our approach. The synthesis of induction axioms from termination proofs has not been studied for this method.

References

[1] Andreas Abel. Termination checking with types. *RAIRO – Theoretical Informatics and Applications*, 38(4):277–319, 2004. Special Issue: Fixed Points in Computer Science (FICS'03).

[2] Markus Aderhold. Automated termination analysis for programs with second-order recursion. Technical report, TU Darmstadt, 2009.

[3] Gilles Barthe, Benjamin Grégoire, and Colin Riba. Type-based termination with sized products. In *Proc. of CSL-22*, volume 5213 of *LNCS*, pages 493–507. Springer, 2008.

[4] J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated termination analysis for Haskell: From term rewriting to programming languages. In *Proc. of RTA-17*, pages 297–312. Springer, 2006.

[5] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proc. of Symposium on Principles of Programming Languages*, pages 410–423. ACM, 1996.

[6] Alexander Krauss. Defining recursive functions in Isabelle/HOL 2007. Proc. of the Isabelle Workshop, 2007.

[7] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[8] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, November 2001.

[9] Christoph Walther. Mathematical induction. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 127–228. Oxford University Press, 1994.

[10] Christoph Walther. On proving the termination of algorithms by machine. *Artificial Intelligence*, 71(1):101–157, 1994.

[11] Christoph Walther and Stephan Schweitzer. Verification in the classroom. *J. Autom. Reason.*, 32(1), 2004.

[12] Christoph Walther and Stephan Schweitzer. Automated termination analysis for incompletely defined programs. In *Proc. of LPAR-11*, volume 3452 of *LNAI*, pages 332–346. Springer, 2005.

Non-Collapsing Context-Sensitive Dependency Pairs*

Beatriz Alarcón[1] Fabian Emmes[2] Carsten Fuhs[2] Jürgen Giesl[2] Raúl Gutiérrez[1]
Salvador Lucas[1] Peter Schneider-Kamp[3] René Thiemann[4]

[1] DSIC, Universidad Politécnica de Valencia, Spain
[2] LuFG Informatik 2, RWTH Aachen University, Germany
[3] Dept. of Mathematics & Computer Science, University of Southern Denmark, Odense, Denmark
[4] Institute of Computer Science, University of Innsbruck, Austria

Abstract

Context-sensitive dependency pairs (CS-DPs) are the most powerful method for automated termination analysis of context-sensitive rewriting. However, the CS-DPs introduced in [1] suffer from two main drawbacks: (a) CS-DPs can be *collapsing* which complicates their handling. (b) [1] did not provide a "*DP framework*" for CS-DPs which would allow to apply them in a flexible and modular way. We solve drawback (a) by introducing a new definition of CS-DPs. With our definition, CS-DPs are always non-collapsing and thus, they can be handled like ordinary DPs. This also eases the solution of drawback (b), i.e., we extend the existing DP framework for ordinary DPs to context-sensitive rewriting. We implemented our results in the tool AProVE and successfully evaluated them on a large collection of examples.

1 Introduction

Context-sensitive rewriting [9] uses a *replacement map* μ to specify the argument positions of function symbols where rewriting may take place. Consider the context-sensitive term rewrite system (CS-TRS)

$$\begin{array}{rclcrcl} \mathsf{zeros} & \to & \mathsf{cons}(0,\mathsf{g}(\mathsf{zeros})) & (1) & \mathsf{take}(\mathsf{s}(x),\mathsf{cons}(y,z)) & \to & \mathsf{take}(x,z) \quad (2) \\ \mathsf{take}(x,\mathsf{g}(y)) & \to & \mathsf{take}(x,y) & & \mathsf{take}(0,\mathsf{cons}(y,z)) & \to & y \end{array}$$

with $\mu(\mathsf{cons}) = \{1\}$ and $\mu(f) = \{1,\ldots,\mathrm{arity}(f)\}$ for all other symbols f to model the usual behavior of lazy evaluation on lists. It will turn out that due to μ, this CS-TRS is indeed terminating. In contrast, if one allows arbitrary reductions, then the TRS would be non-terminating because of Rule (1).

There are two approaches to prove termination of context-sensitive rewriting. The first approach transforms CS-TRSs to ordinary TRSs, cf. [5, 10]. But these transformations often generate complicated TRSs where all termination tools fail. Thus, it is more promising to adapt termination techniques from ordinary term rewriting to the CS setting. For the dependency pair [4] method, this was solved first in [1] and implemented in the corresponding tool MU-TERM [2], cf. Sect. 2. But the CS-DP method from [1] still had disadvantages compared to the DP method for ordinary rewriting, since CS-DPs could be *collapsing*. To handle such DPs, [1] imposed strong requirements which made the CS-DP method quite weak and which made it difficult to extend refined termination techniques based on DPs to the CS case. In particular, [1] did not adapt the *DP framework* [7], which is the most powerful formulation of the DP method for ordinary TRSs, to the CS setting. Therefore, we present a new notion of *non-collapsing* CS-DPs in Sect. 3 and show how to adapt the DP framework accordingly in Sect. 4. Sect. 5 evaluates our results empirically.

2 Collapsing CS-Dependency Pairs

Def. 1 introduces the CS-DPs of [1]. Here, "$r \trianglerighteq_\mu t$" means that t is a subterm of r at an active position.

Definition 1 (CS-DPs [1]). *Let $(\mathcal{R},\mu)$ be a CS-TRS. If $\ell \to r \in \mathcal{R}$, $r \trianglerighteq_\mu t$, and* $\mathrm{root}(t)$ *is a defined symbol, then $\ell^\sharp \to t^\sharp$ is an* ordinary dependency pair. *If $\ell \to r \in \mathcal{R}$, $r \trianglerighteq_\mu x$ for a variable x, and $\ell \not\trianglerighteq_\mu x$, then $\ell^\sharp \to x$ is a* collapsing DP. *Let $\mathsf{DP}_o(\mathcal{R},\mu)$ and $\mathsf{DP}_c(\mathcal{R},\mu)$ be the sets of all ordinary resp. collapsing DPs.*

*Supported by the EU (FEDER) and the Spanish MEC/MICINN (grants TIN 2008-68093-C02-02, HA 2006-0007, FPU grant AP2005-3399, TIN 2004-7943-C04-02) and the DAAD and the DFG (grants D/06/12785 and GI 274/5-2).

In our example, we obtain the following CS-DPs.

$\mathsf{TAKE}(x,\mathsf{g}(y)) \to \mathsf{TAKE}(x,y)$ (3) $\quad \mathsf{TAKE}(\mathsf{s}(x),\mathsf{cons}(y,z)) \to \mathsf{TAKE}(x,z)$ (4) $\quad \mathsf{TAKE}(\mathsf{s}(x),\mathsf{cons}(y,z)) \to z$

To prove termination, one has to show that there is no infinite (minimal) *chain* of DPs. For ordinary rewriting, a sequence $s_1 \to t_1, s_2 \to t_2, \ldots$ of DPs is a *minimal chain* if there is a substitution σ such that $t_i\sigma$ is terminating and reduces to $s_{i+1}\sigma$.[1] Due to the collapsing DPs, the notion of "chains" has to be adapted. If $s_i \to t_i$ is a collapsing DP (i.e., if $t_i \in \mathscr{V}$), then instead of $t_i\sigma \hookrightarrow^*_{\mathscr{R},\mu} s_{i+1}\sigma$, one requires that there is a term w_i with $t_i\sigma \trianglerighteq_\mu w_i$, $w_i^\sharp$ is terminating, and $w_i^\sharp \hookrightarrow^*_{\mathscr{R},\mu} s_{i+1}\sigma$. A CS-TRS is terminating iff there is no infinite minimal chain w.r.t. $\mathsf{DP}_o(\mathscr{R},\mu) \cup \mathsf{DP}_c(\mathscr{R},\mu)$, $\mathscr{R}$, and μ [1].

Due to the collapsing CS-DPs (and the new definition of "chains"), it is not easy to extend existing termination techniques to CS-DPs. Therefore, we now introduce a new improved definition of CS-DPs.

3 Non-Collapsing CS-Dependency Pairs

Ordinary DPs only consider active subterms of right-hand sides. So Rule (1) leads to no DP. However, the inactive subterm zeros of the right-hand side of (1) may become active again when applying Rule (2). Therefore, Def. 1 creates a collapsing DP $\ell^\sharp \to x$ whenever a rule $\ell \to r$ has a *migrating variable* x with $r \trianglerighteq_\mu x$, but $\ell \not\trianglerighteq_\mu x$. Our main observation is that collapsing DPs are only needed for certain instantiations. One might be tempted to allow only instantiations of migrating variables by *hidden terms*. We say that t is a *hidden term* if $\mathrm{root}(t)$ is defined and if there exists a rule $\ell \to r \in \mathscr{R}$ with $r \rhd_{\not\mu} t$. In our example, the only hidden term is zeros. But unfortunately, allowing instantiations of migrating variables with only hidden terms would be unsound. We also have to regard the possible contexts a hidden term can occur in.

Definition 2. *The function symbol f* hides position *i if there is a $\ell \to r \in \mathscr{R}$ with $r \rhd_{\not\mu} f(r_1,...,r_i,...,r_n)$, $i \in \mu(f)$, and r_i contains a defined symbol or a variable at an active position. A context C is* hiding *iff $C = \Box$ or C has the form $f(t_1,\ldots,t_{i-1},C',t_{i+1},\ldots,t_n)$ where f hides position i and C' is a* hiding context.

In our example, g hides position 1 due to Rule (1). So the hiding contexts are $\Box, \mathsf{g}(\Box), \mathsf{g}(\mathsf{g}(\Box)), \ldots$ To remove collapsing DPs $s \to x$, we now restrict ourselves to instantiations of x with terms of the form $C[t]$ where C is a hiding context and t is a hidden term. So in the DP $\mathsf{TAKE}(\mathsf{s}(x),\mathsf{cons}(y,z)) \to z$ the variable z should only be instantiated by zeros, g(zeros) etc. To represent these infinitely many instantiations in a finite way, we replace $s \to x$ by new *unhiding* DPs (which "unhide" hidden terms).

Definition 3 (Improved CS-DPs). *For a CS-TRS $(\mathscr{R},\mu)$, if $\mathsf{DP}_c(\mathscr{R},\mu) \neq \varnothing$, we introduce a fresh* unhiding tuple symbol U *with $\mu(\mathsf{U}) = \varnothing$ and the following* unhiding DPs*:*

- *$s \to \mathsf{U}(x)$ for every $s \to x \in \mathsf{DP}_c(\mathscr{R},\mu)$,*
- *$\mathsf{U}(f(x_1,\ldots,x_i,\ldots,x_n)) \to \mathsf{U}(x_i)$ for every function symbol f of any arity n and every $1 \le i \le n$ where f hides position i, and*
- *$\mathsf{U}(t) \to t^\sharp$ for every hidden term t.*

Let $\mathsf{DP}_u(\mathscr{R},\mu)$ be the set of all unhiding DPs (where $\mathsf{DP}_u(\mathscr{R},\mu) = \varnothing$, if $\mathsf{DP}_c(\mathscr{R},\mu) = \varnothing$). Then the set of improved CS-DPs *is $\mathsf{DP}(\mathscr{R},\mu) = \mathsf{DP}_o(\mathscr{R},\mu) \cup \mathsf{DP}_u(\mathscr{R},\mu)$.*

In our example, instead of $\mathsf{TAKE}(\mathsf{s}(x),\mathsf{cons}(y,z)) \to z$ we get the following unhiding DPs.

$$\mathsf{TAKE}(\mathsf{s}(x),\mathsf{cons}(y,z)) \to \mathsf{U}(z) \qquad \mathsf{U}(\mathsf{g}(x)) \to \mathsf{U}(x) \qquad \mathsf{U}(\mathsf{zeros}) \to \mathsf{ZEROS}$$

Clearly, the improved CS-DPs are never collapsing. Thus, now *minimal chains* can be defined as for ordinary rewriting [7]. Our main theorem shows that improved CS-DPs are still sound and complete.

Theorem 4. *$(\mathscr{R},\mu)$ terminates iff there is no infinite minimal chain w.r.t. $\mathsf{DP}(\mathscr{R},\mu)$, $\mathscr{R}$, and μ.*

[1] We always assume that different occurrences of DPs are variable-disjoint and substitutions may have infinite domains.

4 CS Dependency Pair Framework

Due to our new notion of CS-DPs, adapting the DP framework to the CS case now becomes much easier. In the CS-DP framework, termination techniques operate on *CS-DP problems* instead of TRSs. A *CS-DP problem* is a tuple $(\mathscr{P},\mathscr{R},\mu)$, where $\mathscr{P}$ and $\mathscr{R}$ are TRSs, and μ is a replacement map. A *CS-DP problem* $(\mathscr{P},\mathscr{R},\mu)$ is *finite* if there is no infinite chain w.r.t. $\mathscr{P}$, $\mathscr{R}$, and μ. A *CS-DP processor* is a function *Proc* that takes a CS-DP problem as input and returns a possibly empty set of CS-DP problems. The processor *Proc* is *sound* if a CS-DP problem d is finite whenever all problems in $Proc(d)$ are finite. For a CS-TRS $(\mathscr{R},\mu)$, the termination proof starts with the *initial DP problem* $(\mathsf{DP}(\mathscr{R},\mu),\mathscr{R},\mu)$. Then sound DP processors are applied repeatedly. If the final processors return empty sets, then termination is proved. In the following, we adapt two of the most important processors to the CS case.

4.1 Dependency Graph Processor

The first processor decomposes a DP problem into several sub-problems. To this end, one determines which pairs can follow each other in chains by constructing a *dependency graph*. In contrast to related definitions for collapsing CS-DPs in [1], now the definition of dependency graphs is analogous to the corresponding definition for non-CS rewriting. One can prove termination separately for each strongly connected component of the graph. Thus, the following processor modularizes termination proofs.

Theorem 5 (CS-Dependency Graph Processor)**.** *Let $Proc(d) = \{(\mathscr{P}_1,\mathscr{R},\mu),\dots,(\mathscr{P}_n,\mathscr{R},\mu)\}$, for $d = (\mathscr{P},\mathscr{R},\mu)$, where $\mathscr{P}_1,\dots,\mathscr{P}_n$ are the SCCs of the $(\mathscr{P},\mathscr{R},\mu)$-dependency graph. Then Proc is sound.*

As in the non-CS setting, the CS-dependency graph is not computable and has to be approximated. In our example the SCCs of the dependency graph are $\mathscr{P}_1 = \{(3),(4)\}$ and $\mathscr{P}_2 = \{\mathsf{U}(\mathsf{g}(x)) \to \mathsf{U}(x)\}$.

4.2 Reduction Pair Processor

There are several processors to simplify DP problems by applying suitable *well-founded orders*. Due to the absence of collapsing DPs, most of these processors are now straightforward to adapt to the context-sensitive setting. In the following, we present the reduction pair processor with *usable rules*.

To prove that a DP problem is finite, the reduction pair processor generates constraints which should be satisfied by a *μ-reduction pair* $(\succsim,\succ)$ [1]. Here, $\succsim$ is a stable μ-monotonic quasi-order, $\succ$ is a well-founded stable order, and $\succsim$ and $\succ$ are compatible (i.e., $\succ \circ \succsim \subseteq \succ$ or $\succsim \circ \succ \subseteq \succ$). Here, *$\mu$-monotonicity* means that $s_i \succsim t_i$ implies $f(s_1,\dots,s_i,\dots,s_n) \succsim f(s_1,\dots,t_i,\dots,s_n)$ whenever $i \in \mu(f)$. For a DP problem $(\mathscr{P},\mathscr{R},\mu)$, the generated constraints ensure that some rules in $\mathscr{P}$ are strictly decreasing (w.r.t. $\succ$) and all remaining rules in $\mathscr{P}$ and $\mathscr{R}$ are weakly decreasing (w.r.t. $\succsim$). Requiring $\ell \succsim r$ for all $\ell \to r \in \mathscr{R}$ ensures that in a chain $s_1 \to t_1, s_2 \to t_2, \dots$ with $t_i\sigma \hookrightarrow^*_{\mathscr{R},\mu} s_{i+1}\sigma$, we have $t_i\sigma \succsim s_{i+1}\sigma$ for all i. Hence, if a reduction pair satisfies the constraints, then one can delete the strictly decreasing pairs from $\mathscr{P}$.

To improve this idea, it is desirable to require only weak decrease of *certain* instead of *all* rules. In the non-context-sensitive setting, it is sufficient if just the *usable rules* are weakly decreasing, provided that $\succsim$ is *$\mathscr{C}_\varepsilon$-compatible*.[2] Def. 6 adapts the concept of usable rules to the CS setting. Here, $\mathscr{F}^{\not\mu}(\ell)$ are all function symbols at inactive positions in ℓ.

Definition 6 (CS-Usable Rules)**.** *Let $Rls(f) = \{\ell \to r \in \mathscr{R} \mid \mathrm{root}(\ell) = f\}$. For any symbols f,h and CS-TRS $(\mathscr{R},\mu)$, let $f \rhd_{\mathscr{R},\mu} h$ if $f = h$ or if there is a symbol g with $g \rhd_{\mathscr{R},\mu} h$ and a rule $\ell \to r \in Rls(f)$ with $g \in \mathscr{F}^{\not\mu}(\ell) \cup \mathscr{F}(r)$. We define the* usable rules *as:*

$$\mathscr{U}(\mathscr{P},\mathscr{R},\mu) = \bigcup_{s\to t\in\mathscr{P},\, f\in\mathscr{F}^{\not\mu}(s)\cup\mathscr{F}(t),\, f\rhd_{\mathscr{R},\mu}g} Rls(g) \quad\cup\quad \bigcup_{\ell\to r\in\mathscr{R},\, f\in\mathscr{F}^{\not\mu}(r),\, f\rhd_{\mathscr{R},\mu}g} Rls(g)$$

[2]A relation $\succsim$ is $\mathscr{C}_\varepsilon$-compatible if $\mathsf{c}(x,y) \succsim x$ and $\mathsf{c}(x,y) \succsim y$ holds for a fresh function symbol c.

Theorem 7 (CS-Reduction Pair Processor)**.** *Let $(\succsim, \succ)$ be a μ-reduction pair where $\succsim$ is $\mathscr{C}_\varepsilon$-compatible. For a CS-DP Problem $d = (\mathscr{P}, \mathscr{R}, \mu)$, the result of $Proc(d)$ is $\{(\mathscr{P} \setminus \succ, \mathscr{R}, \mu)\}$, if $\mathscr{P} \subseteq (\succ \cup \succsim)$ and $\mathscr{U}(\mathscr{P}, \mathscr{R}, \mu) \subseteq \succsim$, and $\{d\}$ otherwise. Then Proc is sound.*

Thm. 7 significantly improves over previous related processors [1, 8] for collapsing CS-DPs, since these previous CS-reduction pair processors required that the context-sensitive subterm relation is contained in $\succsim$ (i.e., $\trianglerighteq_\mu \subseteq \succsim$) whenever there are collapsing DPs. This is a hard requirement which destroys one of the main advantages of the DP method (i.e., the possibility to filter away arbitrary arguments).

Finding suitable orders for our CS-DP Problems $(\mathscr{P}_1, \mathscr{R}, \mu)$ and $(\mathscr{P}_2, \mathscr{R}, \mu)$ is now easy, since in both cases only the rule (1) is usable. Therefore, our original context-sensitive TRS is terminating.

5 Experiments and Conclusion

We have developed a new notion of context-sensitive dependency pairs which improves significantly over previous notions. There are two main advantages: (1) Easier adaption of termination techniques to CS rewriting, and (2) more powerful termination analysis for CS rewriting. To substantiate Claim (2), we performed extensive experiments. We implemented our new non-collapsing CS-DPs and corresponding DP processors in the termination prover AProVE [6]. In contrast, the prover MU-TERM [2] uses collapsing CS-DPs. While MU-TERM was the most powerful tool for termination analysis of context-sensitive rewriting in the *International Competition of Termination Tools* 2007, due to our new notion of CS-DPs, AProVE is now substantially more powerful. We tested the tools on all 90 context-sensitive TRSs from the *Termination Problem Data Base* that was used in the competition. We used a time limit of 120 seconds for each example. Then MU-TERM can prove termination of 68 examples, whereas the new version of AProVE proves termination of 78 examples. Since 4 examples are known to be non-terminating, at most 8 more of the 90 examples could potentially be detected as terminating. To experiment with our implementation, we refer to `http://aprove.informatik.rwth-aachen.de/eval/CS-DPs/`. A long version of this paper appeared in [3].

References

[1] B. Alarcón, R. Gutiérrez, and S. Lucas. Context-sensitive dependency pairs. In *Proc. FSTTCS'06*, LNCS 4337, pp. 297-308, 2006.

[2] B. Alarcón, R. Gutiérrez, J. Iborra, and S. Lucas. Proving termination of context-sensitive rewriting with MU-TERM. *In Proc. PROLE'06*, ENTCS 188, pp. 105-115, 2007.

[3] B. Alarcón, F. Emmes, C. Fuhs, J. Giesl, R. Gutiérrez, S. Lucas, P. Schneider-Kamp, and R. Thiemann. Improving context-sensitive dependency pairs. In *Proc. LPAR'08*, LNAI 5330, pp. 636 - 651, 2008.

[4] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *TCS*, 236:133-178, 2000.

[5] J. Giesl and A. Middeldorp. Transformation techniques for context-sensitive rewrite systems. *Journal of Functional Programming*, 14(4):379-427, 2004.

[6] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the DP framework. In *Proc. IJCAR'06*, LNAI 4130, pages 281-286, 2006.

[7] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automatic Reasoning*, 37(3):155-203, 2006.

[8] R. Gutiérrez, S. Lucas, and X. Urbain. Usable rules for context-sensitive rewrite systems. In *Proc. RTA'08*, LNCS 5117, pp. 126-141, 2008.

[9] S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1):1-61, 1998.

[10] S. Lucas. Proving termination of context-sensitive rewriting by transformation. *Information and Computation*, 204(12):1782-1846, 2006.

Proving Termination with Matrix Interpretations over the Reals[*]

Beatriz Alarcón Salvador Lucas Rafael Navarro-Marset

DSIC, Universidad Politécnica de Valencia, Spain

`{balarcon,slucas,rnavarro}@dsic.upv.es`

Abstract

Matrix interpretations are a new kind of algebraic interpretations with interesting capabilities for proving termination of (string) rewriting systems. Roughly speaking, a matrix interpretation for a k-ary symbol f is a linear expression $F_1x_1 + \cdots + F_kx_k + F_0$ where the $F_1,\ldots,F_k$ are matrices of $n\times n$ natural numbers and the variables $x_1,\ldots,x_k$ (and also F_0) represent n-tuples of natural numbers. In this paper, we extend this framework to matrices and tuples of *real* numbers. We also compare matrix and polynomial interpretations.

1 Matrix interpretations over the naturals

A matrix intepretation for a k-ary symbol f is a linear expression $F_1x_1 + \cdots + F_kx_k + F_0$ where the $F_1,\ldots,F_k$ are (square) matrices of $n\times n$ natural numbers and the variables $x_1,\ldots,x_k$ (and also the constant term F_0) are n-tuples of natural numbers [1]. An expression like $A\vec{x}$, where A is a matrix and $\vec{x}$ is an n-tuple of numbers, is interpreted as the usual matrix-vector product, i.e., the i-th component y_i of vector $\vec{y} = A\vec{x}$ is $y_i = \sum_{j=1}^{n} A_{ij}x_j$. The following quasiordering $\gtrsim$ on tuples of natural numbers is considered: $\vec{x} = (x_1,\ldots,x_n) \gtrsim (y_1,\ldots,y_n) = \vec{y}$ if $x_i \geq_{\mathbb{N}} y_i$ for all $1 \leq i \leq n$. The following (strict) ordering $>$ on tuples of natural numbers is considered: $\vec{x} = (x_1,\ldots,x_n) > (y_1,\ldots,y_n) = \vec{y}$ if $x_1 >_{\mathbb{N}} y_1$ and $(x_2,\ldots,x_n) \gtrsim (y_2,\ldots,y_n)$. An (extended) ordered algebra $\mathscr{A} = (\mathbf{A}, \mathscr{F}_A, \gtrsim, >)$ is obtained by defining $\mathbf{A} = \mathbb{N}^n$ for some $n \in \mathbb{N}_{>0}$, and giving a matrix interpretation $[f] = F_1x_1 + \cdots + F_kx_k + F_0$ to every k-ary symbol $f \in \mathscr{F}$. The algebra $\mathscr{A}$ is *monotonic* if we further require that for all k-ary symbols $f \in \mathscr{F}$ and matrix coefficients F_i for $1 \leq i \leq k$, we have $(F_i)_{11} \geq 1$. The term ordering induced by a monotonic ordered algebra in the usual way is a *reduction* ordering.

2 Matrix interpretations over the reals

Since it is based on a well-founded ordering $>_{\mathbb{N}}$, it is easy to see that the strict ordering $>$ on n-tuples of natural numbers is *well-founded* too. Our goal is to extend matrix interpretations to real numbers. The previous definitions and orderings are properly generalized by just changing $\mathbb{N}$ by $\mathbb{R}$. Unfortunately, since the strict ordering $>_{\mathbb{R}}$ over the reals is *not* well-founded (think of the infinite decreasing sequence $1 >_{\mathbb{R}} \frac{1}{2} >_{\mathbb{R}} \frac{1}{3} >_{\mathbb{R}} \cdots$, for instance), we have to be careful. In [5] a different ordering over the reals is used to 'achieve' well-foundedness. Given $\delta > 0$, the (strict) ordering $>_{\mathbb{R},\delta}$ over the reals given by $\forall x,y \in \mathbb{R}$, $x >_{\mathbb{R},\delta} y$ if $x - y \geq_{\mathbb{R}} \delta$ is well-founded on subsets $A \subseteq \mathbb{R}$ which are *bounded from below*, i.e., $A \subseteq [\alpha, +\infty)$ for some $\alpha \in \mathbb{R}$ [5, Theorem 1]. Now, the quasiordering $\gtrsim$ on tuples of real numbers is considered: $\vec{x} = (x_1,\ldots,x_n) \gtrsim (y_1,\ldots,y_n) = \vec{y}$ if $x_i \geq_{\mathbb{R}} y_i$ for all $1 \leq i \leq n$. And the following (strict) ordering $>_\delta$ on tuples of real numbers is used: $\vec{x} = (x_1,\ldots,x_n) >_\delta (y_1,\ldots,y_n) = \vec{y}$ if $x_1 >_{\mathbb{R},\delta} y_1$ and $(x_2,\ldots,x_n) \gtrsim (y_2,\ldots,y_n)$. Since we want to have a well-founded ordering on tuples of real numbers, we restrict the attention to the set of *non-negative* real numbers $\mathbb{R}_0 = [0, +\infty)$, which is bounded from below. The following result transforms a non-negativeness test on the output of a matrix function into a non-negativeness test on the matrices composing this function. It also shows that only matrices with non-negative entries can be used to define matrix algebras with a domain of tuples of non-negative numbers.

[*]Work partially supported by the EU (FEDER) and the Spanish MEC/MICINN, under grant TIN 2007-68093-C02-02. Beatriz Alarcón was partially supported by the Spanish MEC/MICINN under FPU grant AP2005-3399. Rafael Navarro-Marset was partially supported by the Spanish MEC/MICINN under FPU grant AP2006-026.

Proposition 1. *Let $F = F_1x_1 + \cdots + F_mx_m + F_0$ be a matrix function such that the entries in the matrices F_i are arbitrary real numbers for all i, $0 \leq i \leq m$. Then, for all $\vec{x}_1,\ldots,\vec{x}_m \in \mathbb{R}^n_0$, $F(\vec{x}_1,\ldots,\vec{x}_m) \geq \vec{0}$ if and only if for all $i, 0 \leq i \leq m$, $F_i \geq 0$.*

Despite the discussion above, the following (strict) ordering $>$ on tuples of real numbers is also used later: $\vec{x} = (x_1,\ldots,x_n) > (y_1,\ldots,y_n) = \vec{y}$ if $x_1 >_{\mathbb{R}} y_1$ and $(x_2,\ldots,x_n) \gtrsim (y_2,\ldots,y_n)$. This ordering on tuples is the 'natural' extension of the usual ordering over tuples of natural numbers and it does not refer to any positive real number δ in its definition. Although it is *not* well-founded, we have the following:

Proposition 2. *Let $F = F_1x_1 + \cdots + F_mx_m + F_0$ be a matrix function such that the entries in the matrices F_i are arbitrary real numbers for all i, $0 \leq i \leq m$. The following statements are equivalent*

1. *for all $\vec{x}_1,\ldots,\vec{x}_m \in \mathbb{R}^n_0$, $F(\vec{x}_1,\ldots,\vec{x}_m) > \vec{0}$*
2. *there is $\delta > 0$ such that for all $\vec{x}_1,\ldots,\vec{x}_m \in \mathbb{R}^n_0$, $F(\vec{x}_1,\ldots,\vec{x}_m) >_\delta \vec{0}$.*
3. *there is $\delta > 0$ such that $\forall i, 1 \leq i \leq m$, $F_i \geq 0$ and $F_0 >_\delta \vec{0}$.*

The following result *characterizes* monotonicity properties of matrix functions w.r.t. the orderings considered above. We assume that the matrix functions deal with tuples in $\mathbb{R}^n_0$.

Proposition 3 (Monotonicity w.r.t. $\gtrsim$, $>$ and $>_\delta$)**.** *Let $F(x_1,\ldots,x_m) = F_1x_1 + \cdots + F_mx_m + F_0$ be such that F_i are $n \times n$-matrices, for all i, $1 \leq i \leq m$. Let $i \in \{1,\ldots,m\}$ and $\delta > 0$. Then,*

1. *F is i-monotonic with respect to $\gtrsim$ if and only if $F_i \geq 0$.*
2. *F is i-monotonic with respect to $>$ if and only if $F_i \geq 0$ and $(F_i)_{11} >_{\mathbb{R}} 0$.*
3. *F is i-monotonic with respect to $>_\delta$ if and only if $F_i \geq 0$ and $(F_i)_{11} \geq_{\mathbb{R}} 1$.*

The matrix algebras over the reals that we use here are as follows:

Definition 1 (Matrix algebra over the reals)**.** *Let $\mathscr{F}$ be a signature and $\delta > 0$. An (extended) ordered matrix $\mathscr{F}$-algebra over the reals is a quadruple $\mathscr{A} = (\mathbf{A}, \mathscr{F}_A, \gtrsim, >_\delta)$ where $\mathbf{A} = \mathbb{R}^n_0$ and some $n \in \mathbb{N}_{>0}$, $\mathscr{F}_A$ consists of matrix interpretations $[f]$ for each $f \in \mathscr{F}$, i.e., $[f] = F_1x_1 + \cdots + F_kx_k + F_0$, where $F_1,\ldots,F_k$ are $n \times n$-matrices of non-negative real numbers, F_0 is an n-tuple of non-negative real numbers, and $\gtrsim$ and $>_\delta$ are given as above.*

As discussed in [5], in practice we do not need to make any δ *explicit* in our proofs of termination. This is because, in practice, we are faced to solve a *finite* number of strict symbolic constraints $s \sqsupset t$ on terms s and t. From each such constraint we obtain a matrix function $[s] - [t]$ which is compared to $\vec{0}$, with $>$. By Proposition 2 each successful strict comparison $[s] - [t] > \vec{0}$ with $>$ actually corresponds to a successful comparison with $>_{\delta(s,t)}$ for some $\delta(s,t)$ which depends on terms s and t only. Since we have a finite number of strict comparisons, we can take δ to be the *least* of all such $\delta(s,t)$. Then, $>_\delta$ is the well-founded ordering underlying all strict comparisons with $>$. Thus, obtaining the value of such δ for a particular finite set of strict constraints is very easy.

Remark 1. *The monotonicity requirements for using matrix algebras over the reals depend on their particular applications (see [5] for a deeper discussion on this). Proposition 3 can be used for this purpose. For instance, in proofs of termination of rewriting using reduction orderings (as done in the examples of this paper), we need that $>_\delta$ is a* reduction ordering*, hence monotonic. By Proposition 3, we have to impose $([f]_i)_{11} \geq 1$ for all symbols $f \in \mathscr{F}$ and i, $1 \leq i \leq ar(f)$.*

3 Using matrix interpretations over the reals

In [6], it is proved that polynomial interpretations over the rationals are strictly more powerful than polynomial interpretations over the naturals. In [2], syntactic conditions on the shape of the rules of the rewrite system, which tell us that polynomial interpretations with real coefficients should be used for addressing the corresponding termination problem, were given for the first time. Roughly speaking, the conditions investigated in [2] have two parts: (1) if the left-hand side l of a rule $l \to r$ is strictly embedded into the right-hand side r of this rule, then we need to use coefficients strictly below 1 in (some parts) of the linear interpretation corresponding to a symbol f which occurs in r but not in l [2, Theorem 9]. On the other hand, (2) if a term s containing f has to be also to be compared with a variable x contained in s (e.g., $s \succeq x$), then we need to use *positive* coefficients in the linear interpretation for f [2, Theorem 11]. When both conditions simultaneously hold, we know that a real coefficient in $(0,1)$ must be used. Let $\mathscr{E}mb(\mathscr{F})$ be a set of projection rules for a signature $\mathscr{F}$, such that

$$\mathscr{E}mb(\mathscr{F}) = \{f(x_1,\dots,x_i,\dots,x_n) \to x_i \mid f \in \mathscr{F}^{(n)}, 1 \leq i \leq n\}$$

Given a subset $\mathscr{E} \subseteq \mathscr{E}mb(\mathscr{F})$ and terms $s,t \in \mathscr{T}(\mathscr{F},\mathscr{X})$, we write $s \trianglerighteq_{\mathscr{E}} t$ if $s \to^*_{\mathscr{E}} t$ (we write $s \triangleright_{\mathscr{E}} t$ if $s \trianglerighteq_{\mathscr{E}} t$ and $s \neq t$). We also denote as $\mathscr{F}(\mathscr{E})$ the set of symbols with projection rules in $\mathscr{E}$: $\mathscr{F}(\mathscr{E}) = \{root(l) \mid l \to r \in \mathscr{E}\}$. Regarding matrices, we have the following result concerning *diagonal* entries:

Proposition 4. *Let $\mathscr{F}$ be a signature, $\mathscr{E} \subseteq \mathscr{E}mb(\mathscr{F})$ and let $s,t \in \mathscr{T}(\mathscr{F},\mathscr{X})$ be such that $t \trianglerighteq_{\mathscr{E}} s$. Let $(\mathbf{A}, \mathscr{F}_A)$ be an $n \times n$-matrix interpretation over the reals. If $[s] > [t]$, then there is $f \in \mathscr{F}(\mathscr{E})$ with $[f] = F_1x_1 + \cdots + F_kx_k + F_0$, i such that $f(X_1,\dots,X_i,\dots,X_k) \to X_i \in \mathscr{E}$ and j, $1 \leq j \leq n$ such that $(F_i)_{jj} < 1$.*

Example 1. *Consider the following TRS $\mathscr{R}$ [1]:*

$$f(f(X)) \quad \to \quad f(g(f(X)))$$

The following matrix interpretation:

$$[f](x) \;=\; \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} x + \begin{pmatrix} 1 \\ 1 \end{pmatrix} \qquad [g](x) \;=\; \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

can be used to prove the termination of $\mathscr{R}$. Note the null diagonal coefficient in the matrix accompanying the variable in the matrix function for g. According to Proposition 4, this coefficient cannot be given any positive (natural) number.

Unfortunately, the second part of the work in [2] does not extend to matrices.

Example 2. *Consider the TRS in Example 1 together with the following rule:*

$$f(g(f(X))) \quad \to \quad X$$

The matrix interpretation in Example 1 is (strictly) compatible with this rule. In particular, we have:

$$[f(g(f(X)))] - [X] = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} X + \begin{pmatrix} 2 \\ 2 \end{pmatrix} - \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} X + \begin{pmatrix} 2 \\ 2 \end{pmatrix} > \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

We have some experimental evidence of the advantage of using rational numbers in matrix entries. For instance, using our preliminary implementation of matrix interpretations with rational entries in MU-TERM, we can prove termination of `secret06/jambox/10.trs` (in the TPDB) by using 2×2-matrix interpretations with rational entries in $\{0, \frac{1}{2}, 1\}$. We are not aware of any termination proof for this system which uses 2×2-matrix interpretations with *integer* entries. According to the records of the 2008 termination competition (see `http://termcomp.uibk.ac.at/termcomp/home.seam`), though, the system is proved terminating by Jambox, using 3×3-matrices with integer entries; by AProVE, using *arctic* matrix interpretations; and by TTT2, using match-bounds. This suggests that rational entries in matrix interpretations can avoid the use of bigger matrices in some cases. We are investigating more specific criteria for using matrix interpretations with rational entries.

4 Matrix interpretations and polynomial intepretations

Linear polynomial interpretations are (strictly) subsumed by matrix interpretations: every linear polynomial interpretation can be seen as a matrix interpretation (where $n = 1$). The TRS $\mathcal{R}$ in Example 1 indirectly shows that matrix interpretations can be successful when polynomial interpretations fail: it is not difficult to see that no polynomial ordering based on polynomial interpretations over the naturals of any degree can be used to prove termination of $\mathcal{R}$. However, we should not quickly conclude that matrix interpretations are more powerful than polynomial interpretations. The following example shows that this is *not* the case.

Example 3. *The following TRS $\mathcal{R}$:*

$$\begin{array}{rcl@{\qquad}rcl} add(0,X) & \to & X & mul(0,X) & \to & 0 \\ add(s(X),Y) & \to & s(add(X,Y)) & mul(s(X),Y) & \to & add(Y,mul(X,Y)) \end{array}$$

is proved terminating by the polynomial ordering induced by the following nonlinear *interpretation:*

$$\begin{array}{rcl@{\qquad}rcl} [0] & = & 0 & [s](x) & = & x+2 \\ [add](x,y) & = & 2x+y+2 & [mul](x,y) & = & 2xy+2x+2y+2 \end{array}$$

However, it cannot be proved terminating by using a reduction ordering based on a matrix interpretation.

5 Conclusion

We have extended Endrullis et al.'s matrix interpretations over the naturals [1] to matrices with *nonnegative real* numbers in the entries. In [3], matrix interpretations with noninteger entries were considered for proving termination of SRSs. As in our proposal for TRSs, the ordering $>_\delta$ over the (non-negative) reals was used for strict comparisons. However, in [3], string rewriting systems are not treated as TRSs (as done, e.g., in [1, Section 6]). Thus, only *constant* symbols are interpreted, and all of them are given a square matrix. The interpretation of a string l or r is again a square matrix (which is obtained by multiplication of the matrices which correspond to the symbols in the string) and the strict comparisons $l > r$ are then performed between *matrices* (instead of tuples as in our case, see Proposition 2). A deeper comparison between our framework (translated to SRSs) and Gerbhardt el al.'s one is an interesting subject for future work (for matrices over the naturals *à la Endrullis et al.'s*, see [1, Section 6] in connection with [4], which originated [3]).

References

[1] J. Endrullis, J. Waldmann, and H. Zantema. Matrix Interpretations for Proving Termination of Term Rewriting. *Journal of Automated Reasoning* 40(2-3):195-220, 2008.

[2] C. Fuhs, R. Navarro-Marset, C. Otto, J. Giesl, S. Lucas, and P. Schneider-Kamp. Search Techniques for Rational Polynomial Orders. In *Proc. of AISC'08*, LNAI 5144:109-124, Springer-Verlag, Berlin, 2008.

[3] A. Gebhardt, D. Hofbauer and J. Waldmann. Matrix Evolutions. In *Proc. of WST'07.*

[4] D. Hofbauer and J. Waldmann. Termination of String Rewriting Systems with Matrix Interpretations. In *Proc. of RTA'06*, LNCS 4098:328-342, Springer Verlag, Berlin, 2006.

[5] S. Lucas. Polynomials over the reals in proofs of termination: from theory to practice. *RAIRO Theoretical Informatics and Applications*, 39(3):547-586, 2005.

[6] S. Lucas. On the relative power of polynomials with real, rational, and integer coefficients in proofs of termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 17(1):49-73, 2006.

Polynomial Path Orders and the Rules of Predicative Recursion with Parameter Substitution*

Martin Avanzini
Institute of Computer Science
University of Innsbruck, Austria
martin.avanzini@uibk.ac.at

Georg Moser
Institute of Computer Science
University of Innsbruck, Austria
georg.moser@uibk.ac.at

Abstract

In earlier work we introduced a restriction of the multiset path order, called *polynomial path order*, that induces polynomial runtime complexity. In this note, we present an extension that accounts for *predicative recursion with parameter substitution*. As confirmed by our implementation, the analytical power of polynomial path orders is significantly increased.

1 Introduction

Bellantoni and Cook [6] characterise the polytime computable functions as the least class of functions containing certain initial functions and which is closed under the schemes of *predicative recursion* and *composition*. Unlike the classical recursion-theoretic characterisation given by Cobham [7], this alternative characterisation does not rely on any externally imposed resource bounds. Instead, to break the strength of primitive recursion, the predicative schemes make use of a syntactic separation of arguments into *safe* and *normal* ones. To highlight this separation, we write $f(\vec{x};\vec{y})$ instead of $f(\vec{x},\vec{y})$ for normal arguments $\vec{x}$ and safe arguments $\vec{y}$. For previously defined functions g, h_1 and h_2, a new function f is defined by predicative recursion (on notation) via the equations

$$\begin{aligned} f(0,\vec{x};\vec{y}) &= g(\vec{x};\vec{y}) \\ f(2z+i,\vec{x};\vec{y}) &= h_i(z,\vec{x};\vec{y},f(z,\vec{x};\vec{y})),\ i \in \{1,2\}\ . \end{aligned} \tag{1}$$

For previously defined functions h, $\vec{r}$ and $\vec{s}$, a function f is defined by predicative composition by

$$f(\vec{x};\vec{y}) = h(\vec{r}(\vec{x};);\vec{s}(\vec{x};\vec{y}))\ . \tag{2}$$

Note that recursion is performed on a normal argument, whereas the recursively computed result $f(z,\vec{x};\vec{y})$ is substituted into a safe argument position of the stepping function h_i. The composition scheme guarantees that safe arguments cannot influence normal ones. Effectively, recursion on recursively computed results is prevented. The scheme of *predicative recursion with parameter substitution* generalises the scheme of predicative recursion depicted in (1). Here, the safe arguments in the recursive call may be altered additionally. Formally, a new function f is defined by predicative recursion with parameter substitution via the equations

$$\begin{aligned} f(0,\vec{x};\vec{y}) &= g(\vec{x};\vec{y}) \\ f(2z+i,\vec{x};\vec{y}) &= h_i(z,\vec{x};\vec{y},f(z,\vec{x};p_1(z,\vec{x};\vec{y}),\dots,p_m(z,\vec{x};\vec{y}))),\ i \in \{1,2\} \end{aligned} \tag{3}$$

for previously defined functions g, h_1, h_2 and $p_1,\dots,p_m$. As Bellantoni has shown in his thesis [5], the polytime-computable functions are closed under predicative recursion with parameter substitution.

Based on the schemes (1) and (2), we present in [2] a restriction of multiset path orders (MPOs), called *polynomial path orders* (POP* for short), that induce *polynomial innermost runtime complexities*. More precisely, for a TRS R define the *innermost runtime complexity function* $\mathrm{rc}^{\mathsf{i}}_{\mathsf{R}}$ as the function that relates the maximal length of an innermost R-derivation to the size of the initial term, whenever the set of initial terms is restricted to *basic* terms. Here a term is called basic, if all subterms are *values*

*This research is partially supported by FWF (Austrian Science Fund) projects P20133.

formed from constructors. Compatibility of a TRS R with some polynomial path order certifies that $\mathsf{rc}^{\mathsf{i}}_{\mathsf{R}}$ is bounded polynomially. Noteworthy, it can be shown that the polytime computable functions exactly correspond to those functions expressed by (syntactically restricted, cf. Theorem 2) TRSs compatible with polynomial path orders.

However, although the order is complete in the above sense, its application is limited to TRSs where recursion follows the specific form of (1). In particular, functions defined by *tail recursion* are excluded. Consider the TRS $\mathsf{R}_{\mathsf{rev}}$ defining the reversal of lists in a tail recursive fashion:

$$\mathsf{rev}(xs) \to \mathsf{rev}_{\mathsf{tl}}(xs, []) \qquad \mathsf{rev}_{\mathsf{tl}}([], ys) \to ys \qquad \mathsf{rev}_{\mathsf{tl}}(x : xs, ys) \to \mathsf{rev}_{\mathsf{tl}}(xs, x : ys)\ .$$

Even MPOs fail on the above TRS due to the definition of $\mathsf{rev}_{\mathsf{tl}}$, thus any application of POP* is doomed to fail as well. On the other hand, notice that the definition of $\mathsf{rev}_{\mathsf{tl}}$ is strongly reminiscent of predicative recursion with parameter substitution (3).

In this note we introduce an extension of POP* that goes beyond MPO and captures the schema (3). On a conceptual level, this is related to an alternative characterisation of the primitive recursive functions given in [12], where a related extension of MPO is employed. The resulting order, dubbed *polynomial path order with parameter substitution* or POP$^*_{\mathsf{PS}}$ for short, is introduced in the next section.

2 The Polynomial Path Order with Parameter Substitution

Throughout the following, we follow the notions of [2, 4]. We fix a finite but else arbitrary signature F, partitioned into defined symbols D and constructors C. We use $\succsim = \succ \uplus \approx$ to denote an *admissible* quasi-precedence, i.e. a precedence where constructors are minimal. The separation of safe and normal argument positions is taken into account by the notion of *safe mapping*. A safe mapping safe is a function that associates with every n-ary function symbol f the set of *safe argument positions*. For constructors $f \in \mathsf{C}$ we set all argument positions safe. The argument positions not included in $\mathsf{safe}(f)$ are called *normal* and denoted by $\mathsf{nrm}(f)$. We use $\approx_{\mathsf{s}}$ to denote term equivalence as induced by $\succsim$. (Moreover, we assume $\approx_{\mathsf{s}}$ respects the separation of argument positions, cf. [4].)

The *polynomial path order* $>_{\mathsf{pps}*}$ *with parameter substitution* is based on an auxiliary order $>_{\mathsf{pps}}$ inductively defined as follows: $s = f(s_1, \dots, s_n) >_{\mathsf{pps}} t$ if either

(i) $s_i \gtrsim_{\mathsf{pps}} t$ for some $i \in \{1, \dots, n\}$, and if $f \in \mathsf{D}$ then $i \in \mathsf{nrm}(f)$, or

(ii) $t = g(t_1, \dots, t_m)$ with $f \succ g$, $f \in \mathsf{D}$ and $s >_{\mathsf{pps}} t_j$ for all $j \in \{1, \dots, m\}$.

Here $\gtrsim_{\mathsf{pps}} := >_{\mathsf{pps}} \cup \approx_{\mathsf{s}}$. The split into two orders is necessary, as we must enforce the special shape of predicative composition (2) in the definition of $>_{\mathsf{pps}*}$ below. (Note that due to the restrictive definition of case (i), one can show $f(\vec{x};\vec{y}) >_{\mathsf{pps}} r_i(\vec{x};)$, but one cannot show $f(\vec{x};\vec{y}) >_{\mathsf{pps}} s_i(\vec{x};\vec{y})$.)

Based on $>_{\mathsf{pps}}$, the polynomial path order $>_{\mathsf{pps}*}$ with parameter substitution is defined as follows. Here we write $\mathsf{F}^{\prec f}$ for the restriction of F to symbols ranked below f in the precedence, i.e., $\mathsf{F}^{\prec f} := \{g \mid f \succ g \land f \in \mathsf{F}\}$. Further, we write $\{\!\{a_1, \dots, a_n\}\!\}$ for the multiset with elements $a_1, \dots, a_n$. We set $\gtrsim_{\mathsf{pps}*} := >_{\mathsf{pps}*} \cup \approx_{\mathsf{s}}$ and refer with $>^{\mathsf{mul}}_{\mathsf{pps}*}$ to the strict order contained in the multiset extension of $\gtrsim_{\mathsf{pps}*}$. We inductively define $>_{\mathsf{pps}*}$ such that: $s = f(s_1, \dots, s_n) >_{\mathsf{pps}*} t$ if

(i) $s_i \gtrsim_{\mathsf{pps}*} t$ for some $i \in \{1, \dots, n\}$, or

(ii) $t = g(t_1, \dots, t_m)$ with $f \succ g$, $f \in \mathsf{D}$, and

  (a) $s >_{\mathsf{pps}} t_j$ for all $j \in \mathsf{nrm}(g)$ and $s >_{\mathsf{pps}*} t_j$ for all $j \in \mathsf{safe}(g)$, and

  (b) there exists $j_0 \in \mathsf{safe}(g)$ such that $t_j \in \mathsf{T}(\mathsf{F}^{\prec f}, \mathsf{V})$ for all $j \neq j_0$, or

(iii) $t = g(t_1,\dots,t_m)$ with $f \approx g$ and

  (a) $\{\!\{s_{i_1},\dots,s_{i_p}\}\!\} >^{\mathsf{mul}}_{\mathsf{pps}*} \{\!\{t_{j_1},\dots,t_{j_q}\}\!\}$ where $\mathsf{nrm}(f) = \{i_1,\dots,i_p\}$ and $\mathsf{nrm}(g) = \{j_1,\dots,j_q\}$, and

  (b) both $s >_{\mathsf{pps}*} t_j$ and $t_j \in \mathsf{T}(\mathsf{F}^{\prec f},\mathsf{V})$ for all $j \in \mathsf{safe}(g)$.

Case (ii) basically accounts for predicative composition. The condition (ii.b) is used to guarantee that at most one recursively computed results is substituted into a safe argument position j_o. Case (iii) reflects the scheme of predicative recursion with parameter substitution (3). The additional condition (iii.b) is essential, it prohibits for instance the orientation of $\mathsf{f}(\mathsf{s}(;x)\,;y) \to \mathsf{f}(x;\mathsf{f}(x;y))$ that gives rise to exponentially long (innermost) derivations.

Opposed to *recursive path orders* like MPO, $>_{\mathsf{pps}*}$ is neither closed under contexts nor closed under substitutions. Still, following the pattern of the proof given in [2], we derive our main theorem.

Theorem 1. *Let* R *be a finite constructor TRS. If* R *is compatible with* $>_{\mathsf{pps}*}$*, i.e.,* $\mathsf{R} \subseteq >_{\mathsf{pps}*}$*, then the innermost runtime complexity* $\mathrm{rc}^{\mathsf{i}}_{\mathsf{R}}$ *induced is polynomially bounded.*

Notice that $>_{\mathsf{pps}*}$ is applicable to the TRS $\mathsf{R}_{\mathsf{rev}}$ from above: define $\mathsf{safe}(\mathsf{rev}) = \varnothing$ and $\mathsf{safe}(\mathsf{rev}_{\mathsf{tl}}) = \{2\}$. Moreover, set $\mathsf{rev} \succ \mathsf{rev}_{\mathsf{tl}} \succ (:) \succ []$ in the precedence. Compatibility $\mathsf{R}_{\mathsf{rev}} \subseteq >_{\mathsf{pps}*}$ is now straight forward to verify. By Theorem 1, we conclude that the innermost runtime complexity of $\mathsf{R}_{\mathsf{rev}}$ is polynomially bounded.

We stress that every TRS compatible with some instance of POP* is also compatible with some instance of POP$^*_{\mathsf{PS}}$. As demonstrated by the TRS $\mathsf{R}_{\mathsf{rev}}$, the reverse direction does not hold. Consequently, Theorem 1 is strictly more powerful than the main theorem given in [2].

In [2] we have shown that POP* gives rise to an alternative characterisation of the polytime computable functions. The same observation carries over to POP$^*_{\mathsf{PS}}$. The next theorem establishes that POP$^*_{\mathsf{PS}}$ induces polytime computability of the function described through the analysed TRS. The proof of the theorem follows exactly the proof of the corresponding theorem given in [3].

Theorem 2. *Suppose* R *is a finite, orthogonal and sorted constructor TRS based on a simple signature. If* R *is compatible with* $>_{\mathsf{pps}*}$ *then the functions computed by* R *are polytime computable and vice verse, each polytime computable function is computable by such a TRS that is compatible with* $>_{\mathsf{pps}*}$.

Here *simple* signature [11] essentially means that the size of any constructor term depends only polynomially on its depth. The restriction is responsible for the introduction of sorts, compare [2, 11]. Simple signatures allow the definition of enumerated and inductive datatypes like lists and words, but prohibit for instance the definition of tree structures.

3 Experimental Results

We have implemented the here described technique in the experimental version of the *Tyrolean Complexity Tool* T$_\mathsf{C}$T, an open source complexity analyser[1] All experiments were conducted on a machine that is identical to the official competition server (8 AMD Opteron® 885 dual-core processors with 2.8GHz, 8x8 GB memory). As timeout we use 5 seconds. Experiments were performed on a subset of the 1394 examples from the Termination Problem Database Version 5.0.2 that were used in the runtime-complexity category of the termination competition 2008[2], where we filtered out all non-constructor TRSs. The restricted testbed consists of 638 TRSs.

[1]The stable version is available at `http://cl-informatik.uibk.ac.at/software/tct`. A preliminary build and sources of the experimental version can be found at `http://cl-informatik.uibk.ac.at/~zini/wst09`.

[2]See `http://termcomp.uibk.ac.at`.

Table 1 compares the application of $\mathrm{POP}^*_{\mathrm{PS}}$ to the application of POP^* from [2], where we highlight the total on yes-, maybe- and timeout-instances. Furthermore, we annotate average times in seconds.[3] To check compatibility we encode the constraints on precedence and so forth in *propositional logic* (cf. [1] for details), employing MiniSat [8] for finding satisfying assignments. Table 1 reflects that the here proposed extension significantly increases the analytical power of polynomial path orders.

Table 1: Experimental Results

	Yes	Maybe	Timeout
POP^*	40 / **0.03**	598 / **0.05**	0 / **0.00**
$\mathrm{POP}^*_{\mathrm{PS}}$	51 / **0.05**	585 / **0.14**	2 / **5.02**

4 Conclusion

In this paper we study the runtime complexity of rewrite systems. We extend polynomial path orders with the scheme of predicative recursion with parameter substitution, resulting in a strictly more powerful technique. For constructor TRSs compatible with some instance of $\mathrm{POP}^*_{\mathrm{PS}}$, we conclude a *polynomial* bound on the *innermost runtime complexity* of the studied term rewrite system. Moreover, we obtain an alternative characterisation of the *polytime computable functions*. We have implemented the technique and experimental evidence clearly indicates the power and in particular the efficiency of the new method. Although not presented, following [4] we can lift the restriction that the TRS under consideration is a constructor TRS. Also worthy of note, the here described technique allows an integration into the *dependency pair framework for complexity analysis* as put forward in [9, 10], compare [4].

References

[1] Martin Avanzini. Automation of polynomial path orders. Master's thesis, University of Innsbruck, Faculty for Computer Science., 2009. Available at `http://cl-informatik.uibk.ac.at/~zini/MT.pdf`.

[2] Martin Avanzini and Georg Moser. Complexity analysis by rewriting. In *Proc. 9th FLOPS*, volume 4989 of *LNCS*, pages 130–146, 2008.

[3] Martin Avanzini and Georg Moser. Complexity analysis by rewriting. Technical report, University of Innsbruck, Faculty for Computer Science, 2008.

[4] Martin Avanzini and Georg Moser. Dependency pairs and polynomial path orders. To appear. In *Proc. 20th RTA*, LNCS, 2009.

[5] Stephen Bellantoni. *Predicative recursion and computational complexity*. PhD thesis, University of Torronto, Faculty for Computer Science, 1992.

[6] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. *CC*, 2(2):97–110, 1992.

[7] Alan Cobham. The intrinsic computational difficulty of functions. In *Proc. 1964 LMPS*, pages 24–30, 1964.

[8] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proc. 6th SAT*, volume 2919 of *LNCS*, pages 502–518, 2003.

[9] Nao Hirokawa and Georg Moser. Automated complexity analysis based on the dependency pair method. In *Proc. 4th IJCAR*, volume 5195 of *LNCS*, pages 364–380, 2008.

[10] Nao Hirokawa and Georg Moser. Complexity, graphs, and the dependency pair method. In *Proc. 15th LPAR*, volume 5330 of *LNCS*, pages 667–681, 2008.

[11] Jean-Yves Marion. Analysing the implicit complexity of programs. *IC*, 183:2–18, 2003.

[12] Andreas Weiermann. A termination ordering for primitive recursive schemata. Preprint.

[3]See `http://cl-informatik.uibk.ac.at/~zini/wst09` for extended results.

Polynomial constraint solving using SMT[*]

Cristina Borralleras
Universitat de Vic, Spain

Salvador Lucas   Rafael Navarro-Marset
DSIC, Universitat Politècnica de València, Spain

Enric Rodríguez-Carbonell   Albert Rubio
LSI, Universitat Politècnica de Catalunya, Spain

Abstract

Polynomial constraint-solving has a long and successful history in the development of tools for proving termination of programs. A few years ago, only Diophantine constraints were generated and solved by state-of-the-art termination tools when proofs of termination of rewrite systems were attempted. A number of recent works, however, show that polynomial constraints *over the reals* can be used to improve the performance of such tools. Well-known and very efficient techniques, like SAT algorithms and tools, have been recently proposed and used for implementing polynomial constraint solving algorithms through appropriate encodings. Recently, we have shown that the use of an SMT (SAT modulo theories) approach for *linear arithmetic constraints* (over the rationals) leads to a more powerful framework which outperforms the best existing solvers and provides a new and powerful approach for implementing better and more general solvers for termination provers. In this paper, we report on these achievements.

1 Introduction

Polynomial (non-linear) constraints are present in many application domains, like program analysis or the analysis of hybrid systems. In particular, polynomial constraint solving is an essential technique in the development of tools for proving termination of symbolic programs as well as rewrite systems. There have been some recent works providing new implementations of solvers for polynomial constraints that outperform previous existing solvers [3]. These new solvers are implemented by translating the problem into SAT [6, 7] or into CSP [10]. Following the success of the translation into SAT, it is reasonable to consider whether there is a better target language than propositional logic to keep as much as possible the arithmetic structure of the source language[1]. In this line, we propose a simple method for solving non-linear polynomial constraints over the integers and the reals by considering finite subdomains of integers and rational numbers and translating the constraints into *SAT modulo linear (real or integer) arithmetic* [5], i.e. satisfiability of quantifier-free boolean combinations of linear equalities, inequalities and disequalities. An interesting feature of this approach (in contrast to the SAT-based treatment of [6, 7]) is that we can handle domains with negative values for free. The resulting problem can be solved by taking off-the-shelf any of the state-of-art solvers for *SAT Modulo Theories (SMT)* that handles linear (real or integer) arithmetic (e.g., Barcelogic [1], Yices [4] or Z3 [11]). The efficiency of these tools is one of the keys for the success of our approach. The resulting solvers we obtain are faster and more flexible in handling different solution domains than all their predecessors.

[*]This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN, under grant TIN 2007-68093-C02. Rafael Navarro-Marset was partially supported by the Spanish MEC/MICINN under FPU grant AP2006-026.

[1]An obvious possibility is the use of the *first-order theory of real closed fields* which was proved decidable by Tarski. In practice, though, this is unfeasible due to the complexity of the related algorithms.

2 Translation from Non-linear to Linear Constraints

A *non-linear monomial* is an expression $v_1^{k_1} \dots v_p^{k_p}$ with $k_i > 0$ for all $i \in \{1 \dots p\}$ and $v_i \neq v_j$ for all $i, j \in \{1 \dots p\}$. A *non-linear arithmetic formula* is a propositional formula, where atoms are of the form

$$\Sigma_{1 \le i \le n} c_i \cdot M_i \bowtie k$$

where $\bowtie \in \{=, \geq, >, \leq, <\}$, k is an integer or a rational number, every M_i is a non-linear monomial and every c_i is an integer or a rational number. In the following, we assume that we have a fresh set of variables $\mathscr{X}_{\mathscr{M}}$ containing a variable x_M for every monomial M of the form $v_1^{p_{i_1}} \cdot \ldots \cdot v_m^{p_{i_m}}$ that can be built out of the variables $v_1 \dots v_m$.

In [2], we describe two transformations from non-linear to linear constraints that correspond to two kinds of domains: integer intervals and finite sets of rational numbers. By lack of space, in this report we only consider solution domains consisting of integers intervals, i.e. integers in the domain $\{\mathscr{B}_1, \dots, \mathscr{B}_2\}$ for some lower bound $\mathscr{B}_1$ and upper bound $\mathscr{B}_2$. One particular case that we will often use is when $\mathscr{B}_1$ is 0. Before providing a formal definition, let's consider an example:

Example 1. *Consider the atom:*

$$2a^3b - 5cd^2e \geq 0$$

with $\mathscr{B}_1 = 0$ *and* $\mathscr{B}_2 = 3$*. Then, the translation is done by adding variables* x_{a^3b}*,* x_{cd^2e} *and* y_{d^2e}*, which represent*

$$x_{a^3b} = a^3 \cdot b \qquad x_{cd^2e} = c \cdot y_{d^2e} \qquad y_{d^2e} = d^2 \cdot e$$

Linearizing, we obtain the following equisatisfiable constraint:

$$\begin{array}{lll}
2x_{a^3b} - 5x_{cd^2e} \geq 0 & & \\
a = 0 \rightarrow x_{a^3b} = 0 & c = 0 \rightarrow x_{cd^2e} = 0 & d = 0 \rightarrow y_{d^2e} = 0 \\
a = 1 \rightarrow x_{a^3b} = b & c = 1 \rightarrow x_{cd^2e} = y_{d^2e} & d = 1 \rightarrow y_{d^2e} = e \\
a = 2 \rightarrow x_{a^3b} = 8b & c = 2 \rightarrow x_{cd^2e} = 2y_{d^2e} & d = 2 \rightarrow y_{d^2e} = 4e \\
a = 3 \rightarrow x_{a^3b} = 27b & c = 3 \rightarrow x_{cd^2e} = 3y_{d^2e} & d = 3 \rightarrow y_{d^2e} = 9e \\
0 \leq a \leq 3 & 0 \leq c \leq 3 & 0 \leq d \leq 3 \\
0 \leq b \leq 3 & 0 \leq e \leq 3 &
\end{array}$$

The following definition formalizes our translation procedure.

Definition 1. *Let C be a constraint. The transformation rules are the following:*

Abstraction:
$C[c \cdot M]_p \Longrightarrow C[c \cdot x_M]_p \wedge x_M = M$ — *if M is not linear and c is a constant value*

Linearization 1:
$C \wedge x = v_i^{p_i} \Longrightarrow C \wedge \bigwedge_{\alpha=\mathscr{B}_1}^{\mathscr{B}_2} v_i = \alpha \rightarrow x = \alpha^{p_i} \wedge \mathscr{B}_1 \leq v_i \leq \mathscr{B}_2$ — *if* $p_i > 1$

Linearization 2:
$C \wedge x = v_i^{p_i} \cdot v_j \Longrightarrow C \wedge \bigwedge_{\alpha=\mathscr{B}_1}^{\mathscr{B}_2} v_i = \alpha \rightarrow x = \alpha^{p_i} \cdot v_j \wedge \mathscr{B}_1 \leq v_i \leq \mathscr{B}_2 \wedge \mathscr{B}_1 \leq v_j \leq \mathscr{B}_2$ — *if* $p_i > 0$

Linearization 3:
$C \wedge x = v_i^{p_i} \cdot M \Longrightarrow C \wedge \bigwedge_{\alpha=\mathscr{B}_1}^{\mathscr{B}_2} v_i = \alpha \rightarrow x = \alpha^{p_i} \cdot x_M \wedge \mathscr{B}_1 \leq v_i \leq \mathscr{B}_2 \wedge x_M = M$ — *if M is not linear and* $p_i > 0$

	CSP N4		SMT N4		SMT B4	
	Total	Time	Total	Time	Total	Time
YES	491	424	538	483	539	421
MAYBE	662	3345	880	1599	879	1599
KILLED	351(59)		46(7)		46(6)	

Figure 1: Experimental results with MU-TERM

3 Experiments

We provide a comparison of our solver with the solvers used inside AProVE [8] and MU-TERM [9]. In order to do that, we have used a parametrized version of both that can use different solvers provided by the user. In this way, we have been able to test the performance and the impact of using our solvers instead of the solvers that both systems are currently using. As SMT solver for linear arithmetic, we use Yices 1.0.16, since it has shown the best performance on the constraints we are producing. We have also tried Barcelogic and Z3.

For our experiments we have considered the benchmarks included in the Termination Problems Data Base (TPDB; `www.lri.fr/~marche/tpdb/`), version 4.0, in the category of TRSs, but adding the secret problems considered in the 2007 Termination Competition and removing the directories "TRCSR1", "TRCSR/inn", and all examples that consider special kinds of rewriting that cannot be handled by the version of AProVE we are using (basically, rewriting modulo an equational theory and conditional, relative and context sensitive rewriting) and for which the answer is MAYBE regardless of the solver in use. Altogether, there are 1465 problems. We have performed experiments on a 2GHz 2GB Intel Core Duo with a time limit of 60 seconds (as in the last Termination Competition). Here, we only consider domains of integer coefficients (see [2] for a complete report including experiments with rationals coefficients too).

The tables we have included split, for every experiment, the results depending on whether the answer is YES (then we have a termination proof), MAYBE (we cannot prove termination) or KILLED (we have exceeded the time limit). Besides the number of KILLED problems we add, between parentheses, the number of these problems that are known to be YES for the given tool and solution domain.

3.1 Experiments Using MU-TERM

We have used MU-TERM to generate the polynomial constraints which are used to solve the termination problems considered in our benchmarks. The symbolic constraints for each termination problem are generated according to the dependency pairs technique and solved by using *linear* polynomial interpretations. For integers, we have tried the domain $N4 = \{0,1,2,4\}$ using our SMT-based solver (with disjunction of values) and the CSP-based solver of MU-TERM. To show that even when enlarging the domain, using integer intervals instead of disjunction of values, our solver is faster, we have also considered the integer interval domain $B4 = N4 \cup \{3\}$, for which more examples are solved in less time. As can be seen in Figure 1, the results are far better using our solver than using the CSP solver[2].

3.2 Experiments Using AProVE

We have been provided with a *simplified* version of the AProVE tool for checking termination of term rewrite systems using polynomial interpretations over the integers and the rational numbers, which allows us to focus on the performance of the constraint solver. The system is parametrized by the solver. Basically, the system tries to decompose the dependency graph into smaller problems which are proved

[2]The accumulate of some *'Total'* columns is lesser than 1465 examples due to *segmentation fault* problems when loading some of them.

	BOUND 4				BOUND 6			
	SAT		SMT-LIA		SAT		SMT-LIA	
	Total	Time	Total	Time	Total	Time	Total	Time
YES	598	1176	600	971	598	1256	600	1067
MAYBE	840	2320	843	2005	828	2676	838	2148
KILLED	26(4)		21(2)		38(5)		26(3)	

	BOUND 12				BOUND 32			
	SAT		SMT-LIA		SAT		SMT-LIA	
	Total	Time	Total	Time	Total	Time	Total	Time
YES	589	1359	597	1251	575	1848	586	1425
MAYBE	774	4531	820	2815	570	4545	767	3804
KILLED	100(14)		47(6)		317(28)		111(17)	

Figure 2: Experiments with integers in APROVE (times are in seconds)

by generating polynomial constraints (as for MU-TERM). The constraints are sent to the solver which in turn returns a solution if one is found.

The results in Figure 2 show that, in general, our solver is always faster than the SAT-based APROVE solver, starting about a 20% faster (without counting time outs) and increasing as the domain grows. This improvement is more significant if we take into account that there is an important part of the process that is common (namely the generation of constraints) independently of the solver. Moreover, the number of KILLED problems is always the lowest in our solver and the difference grows as the domain is enlarged.

Acknowledgments: we thank Jürgen Giesl, Carsten Fuhs and Karsten Behrmann for their very quick reaction in providing us with a version of APROVE to be used in our experiments, and specially to Carsten for some interesting and helpful discussion on possible useful extensions of the solver.

References

[1] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonella and A. Rubio. The Barcelogic SMT Solver. *Proc. of 20th CAV*, LNCS 5123:294-298, 2008.

[2] C. Borralleras, S. Lucas, R. Navarro-Marset, E. Rodríguez-Carbonell, and A. Rubio. Solving Non-linear Polynomial Arithmetic via SAT Modulo Linear Arithmetic. In *Proc. of CADE'09*, LNAI *to appear*, 2009.

[3] E. Contejean, C. Marché, A.-P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. Journal of Automated Reasoning, 34(4):325-363, 2006.

[4] B. Dutertre and L. de Moura. The Yices SMT solver. System description report. `http://yices.csl.sri.com/`

[5] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). *Proc. of 18th CAV*, LNCS 4144:81-94, 2006.

[6] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT Solving for Termination Analysis with Polynomial Interpretations. *Proc. of 10th SAT*, LNCS 4501:340-354, 2007.

[7] C. Fuhs, R. Navarro-Marset, C. Otto, J. Giesl, S. Lucas, and P. Schneider-Kamp. Search Techniques for Rational Polynomial Orders. In *Proc. of 9th AISC*, LNAI 5144:109-124, Springer-Verlag, Berlin, 2008.

[8] J. Giesl, P. Schneider-Kamp, and R. Thiemann. APROVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. *3rd Int. Joint Conference on Automated Reasoning* , LNAI 4130:281-286, 2006.

[9] S. Lucas. MU-TERM: A Tool for Proving Termination of Context-Sensitive Rewriting. In *Proc. of RTA'04*, LNCS 3091:200-209, 2004.

[10] S. Lucas. Practical use of polynomials over the reals in proofs of termination. *Proc. of 9th PPDP*, pages 39-50, ACM Press, 2007.

[11] L. de Moura and N. Bjorner.Z3: An Efficient SMT Solver. *Proc. of 14th TACAS*, LNCS 4963:337-340, 2008.

Getting to the $\perp$

Jan Christiansen
Christian-Albrechts-University
Kiel, Germany
jac@informatik.uni-kiel.de

Abstract

Even in programming languages with call-by-name or call-by-need semantics functions can be more strict than necessary. This has undesirable effects from the termination point of view and from the software engineering point of view. We present the basis for a tool that supports programmers in defining functions as non-strict as possible. This way programmers can use the advantages of these programming languages to full capacity.

1 Introduction

Call-by-name is known to be an optimal evaluation strategy with respect to termination. In practice you can only benefit from this fact if functions are not unnecessarily strict. Often it is not trivial to implement a function in a least strict way. In the following we use the lazy functional programming language Haskell [5] for examples. Our approach does not distinguish between call-by-name and call-by-need because we do not consider sharing. We define Peano Numbers and the multiplication of two numbers by means of *plus*. A Peano Number is either zero or the successor of another Peano Number.

```
data Peano = Zero | Succ Peano

plus :: Peano → Peano → Peano
plus Zero y = Zero
plus (Succ x) y = Succ (plus x y)

mult :: Peano → Peano → Peano
mult Zero y = Zero
mult (Succ x) y = plus y (mult x y)
```

Furthermore we define an infinite Peano number which is an infinite sequence of successors.

```
infinity :: Peano
infinity = Succ infinity
```

Thanks to non-strictness the evaluation of *mult Zero infinity* yields *Zero*. However the evaluation of *mult infinity Zero* does not terminate. We can improve *mult* with respect to strictness and, therefore, also with respect to termination. Consider the following implementation of the multiplication.

```
mult' :: Peano → Peano → Peano
mult' Zero y = Zero
mult' (Succ _) Zero = Zero
mult' (Succ x) y = plus y (mult' x y)
```

For all total arguments *mult* and *mult'* yield the same results. But the evaluation of *mult' infinity Zero* yields *Zero*. Note that even this information is not sufficient to check whether *mult'* is less strict than *mult*. Because the less strict order is a partial order the two functions can still be incomparable.

This is not an artificial example. The paper "Declaring Numbers" [1] introduces lazy natural numbers by a binary representation. As a motivating example they present a multiplication of Peano Numbers which is implemented in the exact same manner as *mult*. Furthermore the library "Numbers"[1] provides an implementation of Peano Numbers which implements the multiplication in the same manner as *mult*. Paper and library care about least strict implementations. This demonstrates that it is even not obvious how to implement a simple function like the multiplication of Peano Numbers in a least strict way.

Besides termination there are even more benefits of least strict functions. In the paper "Why functional programming matters" [4] John Hughes argues that non-strictness can contribute greatly to the modularity of programs. You can compose two functions f and g without bothering too much about the size of the intermediate data structures. Although the total memory that is allocated can be very large the total amount of memory that is occupied at one moment is very limited. If a function is not least strict you cannot rely on this behaviour.

Furthermore there is a close connection between least-strictness and the size of the search space in a lazy functional logic programming language like Curry [3]. We have discovered that we can improve the multiplication of binary natural numbers presented in [1] with respect to least-strictness. When we use this improved multiplication in the enumeration of Pythagorean triples its performance is improved by a factor of 4.

Olaf Chitil first presented the idea of least-strictness and implemented a tool called StrictCheck [2]. The tool was supposed to check whether a function is least strict. But it has some shortcomings. In particular it does not consider sequentiality. That is, it suggests improvements that are not implementable in a sequential language. In Section 3 we show the condition of the original tool and its problems in detail.

2 Least-Strictness

By $n\rfloor$ we denote the set $\{1,\ldots,n\}$. Let Σ be a set of constructors with associated type. By $e :: \tau$ we state that the function and the constructor e respectively has type τ. In the following we consider monomorphic first order types only. We use the following naming conventions: $\tau,\tau',\tau_1,\ldots,\tau_n$ are first order types, C are constructors, $c,c_1,\ldots,c_n$ are contexts, f, g are functions, pv are partial values and v are values. We use curried and uncurried representations of functions interchangeably. That is, we use $f :: \langle\tau_1,\ldots,\tau_n\rangle \to \tau$ and $f :: \tau_1,\to \cdots \to \tau_n \to \tau$, $f\langle c_1,\ldots,c_n\rangle$ and $f\ c_1\ \ldots\ c_n$ interchangeably.

Definition 2.1 (Contexts)**.** The set $\mathrm{Cxt}(\tau)$ denotes the set of contexts with type τ with arbitrary many holes. A context is a term over Σ and the additional symbol $[]$ called hole. Let $c \in \mathrm{Cxt}(\tau)$ be a context with n holes. Then $c[c_1,\ldots,c_n]$ denotes the context c where the i-th hole is filled in with the context c_i.

We do not distinguish semantic and syntactic values in the following. For example, by $\perp$ we denote both, a non terminating calculation and the least semantic element.

Definition 2.2 (Values and Partial Values)**.** The set of values $\mathrm{Val}(\tau) \subset \mathrm{Cxt}(\tau)$ is the set of contexts with no holes. The set of partial values $\mathrm{PVal}(\tau)$ is the set of contexts where all holes are filled in with $\perp$. By $\sqsubseteq$ we denote the standard partial order on partial and total values. By $\bigsqcap$ we denote the corresponding infimum.

Definition 2.3 (Position)**.** A position is a sequence of natural numbers that uniquely identifies a sub-context of a context. If c is a context and p a position than $c|_p$ denotes the sub-context at position p. For example $\{\}$ denotes the root position and $\{1,3\}$ denotes the third subterm of the first subterm.

We define a partial order on functions. Two functions f and g are only comparable if they have the same type. A function f is less or equally strict than a function g if f is equally or more defined than g for all partial arguments.

[1]available via hackage (`http://hackage.haskell.org`)

Definition 2.4 (Less strict)**.** Let $f :: \tau \to \tau'$ and $g :: \tau \to \tau'$.

$$f \preceq g :\Longleftrightarrow \forall pv \in \mathrm{PVal}(\tau) : [\![f\ pv]\!] \sqsupseteq [\![g\ pv]\!]$$

If $f \prec g$ we say that f is less strict than g. The relation $\preceq$ is indeed a partial order because $\sqsupseteq$ is one.

A function that raises an exception should be incomparable to a function that yields a value instead. For example, we do not want to suggest improving the division by replacing the division by zero exception by a value. Therefore, if we employ a semantics with exceptions we have to keep this in mind.

Example 2.5. Consider the Boolean conjunction like it is implemented in Haskell and the strict counterpart of this function.

```
andL :: Bool → Bool → Bool
andL False _ = False
andL True x = x

and :: Bool → Bool → Bool
and False False = False
and False True = False
and True False = False
and True True = True
```

For the partial argument $\langle False, \perp \rangle$ we have $[\![andL\ False\ \perp]\!] = False$ and $[\![and\ False\ \perp]\!] = \perp$. Therefore *andL* is less strict than *and*, that is, $andL \prec and$. There is a third possibility to implement the Boolean conjunction in Haskell.

```
andR :: Bool → Bool → Bool
andR _ False = False
andR x True = x
```

Altogether we have $andL \prec and$ and $andR \prec and$ but $andR \not\preceq andL$ and $andL \not\preceq andR$. This demonstrates that we are looking for a minimal element with respect to $\preceq$ and not for a least one.

3 Checking Least-Strictness

We know that for all functions f and all partial values pv we have $[\![f\ pv]\!] \sqsubseteq \mathrm{inf}_f(pv)$ where $\mathrm{inf}_f(pv) = \bigsqcap\{[\![f\ v]\!] | v \in \mathrm{Val}(\tau), pv \sqsubset v\}$. The StrictCheck tool presented in [2] uses this statement to check whether a function is least strict. It checks the following condition for a function f of type $\tau \to \tau'$.

$$f \text{ is least strict} \Longleftrightarrow \forall pv \in \mathrm{PVal}(\tau) : [\![f\ pv]\!] = \mathrm{inf}_f(pv)$$

The tool optimizes this by only checking partial values that contain a single $\perp$.

Because of sequentiality this condition is too keen. For example the condition above declares *andL* to be not least strict because $[\![andL\ \perp\ False]\!] = \perp$ while $\mathrm{inf}_{andL}(\langle \perp, False \rangle) = False$. In fact, for a function f the tool checks whether there is a function g such that $g \prec f$. But it does not guarantee that g is a sequential function. In a programming language like Haskell you can only define sequential functions.

3.1 Sequentiality

For the definition of sequentiality we refer to the definition of Vuillemin [6] who has considered sequentiality for PCF in the context of full abstraction. PCF is a call-by-name functional language based on the typed λ-calculus. The definition that we present here is customized for our purposes.

Definition 3.1. A PCF-like language is sequential if the following holds. Let c be a context with n holes such that $[\![c[\perp,\dots,\perp]]\!] = \perp$. Then there exists $i \in n\rfloor$ such that

$$[\![c[c_1,\dots,c_{i-1},\perp,c_{i+1},\dots,c_n]]\!] = \perp$$

for all contexts $c_1,\dots,c_n$ of appropriate type.

Functional languages like Haskell are sequential. On basis of this definition we define sequentiality of a function.

Lemma 3.2. *In a sequential language for every function f the following condition holds. Let c be a context with n holes and p a position such that $[\![f\ c[\perp,\dots,\perp]]\!]|_p = \perp$. Then there exists $i \in n\rfloor$ such that*

$$[\![f\ c[c_1,\dots,c_{i-1},\perp,c_{i+1},\dots,c_n]]\!]|_p = \perp$$

for all contexts $c_1,\dots,c_n$ of appropriate type.

This lemma follows directly from the previous one by choosing $c = sel\ (f\ c)$ where sel is a projection to position p. We call a function that fulfills this condition sequential.

Example 3.3. Because we have $[\![andL \perp \perp]\!]|_{\{\}} = \perp$ we know that $\forall pv \in \mathrm{PVal}(Bool) : [\![andL\ pv \perp]\!]|_{\{\}} = \perp$ or $\forall pv \in \mathrm{PVal}(Bool) : [\![andL \perp pv]\!]|_{\{\}} = \perp$. Because we have $[\![andL\ False \perp]\!]|_{\{\}} \neq \perp$ we know that $\forall pv \in \mathrm{PVal}(Bool) : [\![andL \perp pv]\!]|_{\{\}} = \perp$. Therefore we may not demand anything from $andL$ for arguments of the form $\langle pv, \perp\rangle$. This is a contradiction to the original demands of StrictCheck.

That is, to check whether a function f is least strict we have to check whether there is a sequential function g such that $g \prec f$.

References

[1] Bernd Brassel, Sebastian Fischer, and Frank Huch. Declaring numbers. In *Proc. of the 16th International Workshop on Functional and (Constraint) Logic Programming WFLP 2007*, 2007.

[2] Olaf Chitil. Promoting non-strict programming. In *Draft Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages, IFL 2006*, pages 512–516, Budapest, Hungary, September 2006. Eotvos Lorand University.

[3] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://curry-language.org`, 2006.

[4] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[5] Simon P. Jones. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, May 2003.

[6] Jean Etienne Vuillemin. *Proof-techniques for recursive programs.* PhD thesis, Stanford, CA, USA, 1974.

Derivational complexity of $\{\mathtt{aa} \to \mathtt{bc}, \mathtt{bb} \to \mathtt{ac}, \mathtt{cc} \to \mathtt{ab}\}$

Karel Chvalovský[*]
Department of Logic, Faculty of Arts
Charles University, Prague, Czech Republic

Institute of Computer Science,
Academy of Sciences of the Czech Republic, Czech Republic
karel@chvalovsky.cz

We study the derivational complexity of Zantema's string rewriting system over the alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ with the set of rules $\{\mathtt{aa} \to \mathtt{bc}, \mathtt{bb} \to \mathtt{ac}, \mathtt{cc} \to \mathtt{ab}\}$. It was proved [1] that this rewriting system terminates, but the question of derivational complexity remained open.

For simplicity of notation, we use $\mathtt{x}^n$ as an abbreviation for a string with n consecutive occurences of symbol $\mathtt{x}$. The class of strings with the longest possible derivations is reduced by an easy observation.

Lemma 1. *All strings with the longest possible derivations are of the form* $\mathtt{c}^i\mathtt{b}^j\mathtt{a}^k$*, where* $i, j, k \geq 0$.

Let $l(\mathtt{s})$ be the length of the longest derivations starting from a string $\mathtt{s}$, e.g. $l(\mathtt{aa}) = 2$. Let m be a function from strings to strings, which reverts an input string and simultaneously changes all occurrences of $\mathtt{a}$ to $\mathtt{c}$ and vice versa. We can show the following evident lemmas.

Lemma 2. *For any string* $\mathtt{s}$ *we have* $l(\mathtt{s}) = l(m(\mathtt{s}))$.

Lemma 3 (separation)**.** *For any strings* $\mathtt{s}_1$ *and* $\mathtt{s}_2$*, where the last symbol of* $\mathtt{s}_1$ *is not* $\mathtt{a}$ *and the first symbol of* $\mathtt{s}_2$ *is not* $\mathtt{c}$*, we have* $l(\mathtt{s}_1\mathtt{ac}\mathtt{s}_2) = l(\mathtt{s}_1) + l(\mathtt{s}_2) - 1$.

From now on, we emphasize the previous situation by the notation $\mathtt{s}_1|\mathtt{ac}|\mathtt{s}_2$.

We first compute the exact derivational complexity for two classes of strings with long possible derivations.

Lemma 4. *The derivational complexity of strings* $\mathtt{b}^k\mathtt{aa}$ *and* $\mathtt{ccb}^k$*,* $k \geq 0$*, is*

$$l(\mathtt{b}^k\mathtt{aa}) = l(\mathtt{ccb}^k) = \begin{cases} \frac{3}{2}k^2 + k + \frac{5}{2} & \text{if } k \text{ is odd,} \\ \frac{3}{2}k^2 + k + 2 & \text{if } k \text{ is even.} \end{cases}$$

Idea of proof. Due to Lemma 2 we can study only one case. Let us have some string $\ldots\mathtt{bbbbaa}$. We argue mainly by separation lemma that the following derivation is the longest possible up to different step order. We start by

$$\ldots\mathtt{bbbb\underline{aa}} \rightsquigarrow \ldots\mathtt{bbb\underline{bb}c} \rightsquigarrow \ldots\mathtt{bbba\underline{cc}} \rightsquigarrow \ldots\mathtt{bbbaab}$$

and continue by pushing $\mathtt{aa}$ forward with the same arguments and finally obtain $\mathtt{aabbbb}\ldots$. Very useful observation is that any derivation with

$$\ldots\mathtt{bbaa\underline{bb}} \rightsquigarrow \ldots\mathtt{bba\underline{aa}c} \rightsquigarrow \ldots\mathtt{bbabcc},$$

can be simulated by the longer derivation:

$$\ldots\mathtt{bb\underline{aa}bb} \rightsquigarrow \ldots\mathtt{b\underline{bb}cbb} \rightsquigarrow \ldots\mathtt{ba\underline{cc}bb} \rightsquigarrow \ldots\mathtt{baa\underline{bb}b} \rightsquigarrow \ldots\mathtt{ba\underline{aa}cb} \rightsquigarrow \ldots\mathtt{bab\underline{cc}b}$$
$$\rightsquigarrow \ldots\mathtt{baba\underline{bb}} \rightsquigarrow \ldots\mathtt{bab\underline{aa}c} \rightsquigarrow \ldots\mathtt{ba\underline{bb}cc} \rightsquigarrow \ldots\mathtt{b\underline{aa}ccc} \rightsquigarrow \ldots\mathtt{bb\underline{cc}cc} \rightsquigarrow \ldots\mathtt{bbabcc}.$$

[*]Supported by grant 401/09/H007 of GA ČR.

The derivation continues as follows

$$\texttt{aa}\underline{\texttt{bb}}\texttt{bb}\ldots \rightsquigarrow \texttt{a}\underline{\texttt{aa}}\texttt{cbb}\ldots \rightsquigarrow \texttt{ab}\underline{\texttt{cc}}\texttt{bb}\ldots \rightsquigarrow \texttt{aba}\underline{\texttt{bb}}\texttt{b}\ldots \rightsquigarrow \texttt{ab}\underline{\texttt{aa}}\texttt{cb}\ldots \rightsquigarrow \texttt{abb}\underline{\texttt{cc}}\texttt{b}\ldots \rightsquigarrow \texttt{abbabb}\ldots,$$

where all rewriting steps are necessary by separation lemma or other alternatives lead only to variants of our derivation. From $\texttt{abbabb}\ldots$ we can continue by pushing the last $\texttt{a}$ forward. Other alternatives like

$$\texttt{a}\underline{\texttt{bb}}\texttt{abb}\ldots \rightsquigarrow \underline{\texttt{aa}}\texttt{cabb}\ldots \rightsquigarrow \texttt{b}\underline{\texttt{cc}}\texttt{abb}\ldots \rightsquigarrow \texttt{baba}\underline{\texttt{bb}}\ldots \rightsquigarrow \texttt{bab}\underline{\texttt{aa}}\texttt{c}\ldots \rightsquigarrow \texttt{babb}\underline{\texttt{cc}}\ldots \rightsquigarrow \texttt{babbab},$$

are only variants of our derivation or shorter.

By pushing $\texttt{a}$ right we obtain $\texttt{abb}\ldots\texttt{b}|\texttt{ab}$, where $\texttt{ab}$ is separated and $\texttt{abb}\ldots\texttt{b}$ is an immediate rewriting successor of $\texttt{ccb}\ldots\texttt{b}$ and $l(\texttt{ccb}\ldots\texttt{b}) = l(\texttt{b}\ldots\texttt{baa})$ by lemma 2.

We know $l(\texttt{baa}) = 5$ and for $k > 1$ we obtain $l(\texttt{b}^k\texttt{aa}) = 6k - 3 + (l(\texttt{b}^{k-2}\texttt{aa}) - 1)$ which immediately proves the lemma. □

By analysing other classes of strings from Lemma 1 we show in the following lemmas that these classes are of lower derivational complexity. We say that a string s is *maximal* if for every string s' of the same length $l(s') \le l(s)$.

Lemma 5. *For $k > 2$ strings $\texttt{a}^k, \texttt{b}^k$ and $\texttt{c}^k$ are not maximal.*

Lemma 6. *For any $k \ge 0$, $m > 0$ and nonzero $n \ne 2$ a string $\texttt{c}^k\texttt{b}^m\texttt{a}^n$ is not maximal.*

Idea of proof. For $n = 1$ it is obvious and for $n = 3$ we must sooner or later rewrite some $\texttt{aa}$, but in case of $\texttt{c}^k\texttt{b}^m\texttt{bc}|\texttt{a}$ is $\texttt{a}$ out of further rewriting and in case of $\texttt{c}^k\texttt{b}^m|\texttt{abc}$ even $\texttt{abc}$.

For $n \ge 4$ the only reasonable option is sooner or later the following or equivalent rewriting

$$\ldots\texttt{baa}\underline{\texttt{aa}}\ldots \rightsquigarrow \ldots\texttt{b}\underline{\texttt{aa}}\texttt{bc}\ldots \rightsquigarrow \texttt{bbcbc}\ldots,$$

which has a longer derivation:

$$\begin{aligned}\ldots\underline{\texttt{cc}}\texttt{bbb}\ldots &\rightsquigarrow \ldots\texttt{a}\underline{\texttt{bb}}\texttt{bb}\ldots \rightsquigarrow \ldots\underline{\texttt{aa}}\texttt{cbb}\ldots \rightsquigarrow \ldots\texttt{b}\underline{\texttt{cc}}\texttt{bb}\ldots \rightsquigarrow \ldots\texttt{ba}\underline{\texttt{bb}}\texttt{b}\ldots\\ &\rightsquigarrow \ldots\texttt{b}\underline{\texttt{aa}}\texttt{cb}\ldots \rightsquigarrow \ldots\texttt{bb}\underline{\texttt{cc}}\texttt{b}\ldots \rightsquigarrow \ldots\texttt{bba}\underline{\texttt{bb}}\ldots \rightsquigarrow \ldots\texttt{bb}\underline{\texttt{aa}}\texttt{c}\ldots \rightsquigarrow \ldots\texttt{b}\underline{\texttt{bb}}\texttt{cc}\ldots\\ &\rightsquigarrow \ldots\texttt{ba}\underline{\texttt{cc}}\texttt{c}\ldots \rightsquigarrow \ldots\texttt{b}\underline{\texttt{aa}}\texttt{bc}\ldots \rightsquigarrow \ldots\texttt{bbcbc}\ldots\end{aligned}$$

□

Lemma 7. *For any nonzero m and n a string $\texttt{c}^m\texttt{a}^n$ is not maximal.*

Idea of proof. Lemma 2 reduces cases which must be studied significantly. Moreover, the only interesting case is when m and n are both even, because otherwise there is at least one $\texttt{a}$ or $\texttt{c}$ which is out of any rewriting. Let m and n be even and nonzero. It can be shown that the only reasonable strategy is to push $\texttt{aa}$ left and $\texttt{cc}$ right. Then we can only rewrite each $\texttt{aa}$ and $\texttt{cc}$ which gives $l(\texttt{c}^m\texttt{a}^n) = \frac{3}{4}mn + \frac{1}{2}(m+n) + 1$, for even nonzero m and n. □

Lemma 8. *For any $k \ge 1$ a string $\texttt{ccb}^k\texttt{aa}$ is not maximal.*

Idea of proof. By computation we can show that $l(\texttt{ccbaa}) < l(\texttt{bbbaa})$. Let $k \ge 2$. It can be argued by methods similar to previous proofs that the longest possible derivations have the following or equivalent form:

$$\underline{\texttt{cc}}\texttt{bb}^{k-1}\texttt{aa} \rightsquigarrow \texttt{a}\underline{\texttt{bb}}\texttt{b}^{k-1}\texttt{aa} \rightsquigarrow \underline{\texttt{aa}}\texttt{cb}^{k-1}\texttt{aa} \rightsquigarrow \texttt{bccb}^{k-1}\texttt{aa} \rightsquigarrow \ldots \rightsquigarrow \texttt{bb}^{k-1}\texttt{ccaa} \rightsquigarrow \ldots \rightsquigarrow \texttt{bb}^{k-1}\texttt{aacc}$$

We push $\texttt{cc}$ right and then switch $\texttt{cc}$ with $\texttt{aa}$ (or conversely) and now can be shown that the most effective derivations from $\texttt{b}^k\texttt{aacc}$ separate $\texttt{b}^k\texttt{aa}$ and $\texttt{cc}$. Therefore $l(\texttt{ccb}^k\texttt{aa}) = l(\texttt{ccb}^k) + 3k + 3 + 1$. □

Due to Lemma 2 we considered all classes of strings from Lemma 1 and we immediately obtain the following corollary.

Corollary 1. *The derivational complexity of a string* $\mathtt{s}$ *is at most quadratic in its length.*

References

[1] D. Hofbauer & J. Waldmann. Termination of $\{aa \rightarrow bc, bb \rightarrow ac, cc \rightarrow ab\}$. *Information Processing Letters*, 98(4), 156–158, 2006.

Matrix interpretations revisited [*]

Pierre Courtieu Gladys Gbedo Olivier Pons
CÉDRIC, CNAM, Paris, France

1 Introduction

The property of termination, well-known to be undecidable, is fundamental by many aspects of computer sciences and logic. Many heuristics have been proposed to provide automation for termination proofs. Almost all of them require, possibly after several transformations of the initial termination problem, to search a well-founded orderings satisfying some properties. Among the different kinds of orderings, polynomial interpretations [11, 2, 4] and recursive path ordering [5] are the most used. Matrix interpretation is a recent powerful kind of well-founded ordering based on term interpretation into vectors, introduced by Endrullis, Waldmann and Zantema in [9]. This interpretation associates to each symbol a linear mapping with matrix coefficients. This paper presents a generalization of this method, which allows for more matrices and more orderings. In particular it allows for more systems to be proved to be terminating without increasing the bounds for coefficients or the size of matrices.

Due to the monotonicity requirement for interpretations, the original matrix interpretations are restricted to matrices with a strictly positive upper left coefficient, and the associated strict ordering only considers the upper coefficient on vectors. We propose in this paper a weaker limitation still preserving monotonicity. We require for each matrix to have a fixed sub-matrix with no null columns. The strict ordering will consider coefficients corresponding to this sub-matrix. Note that matrix interpretations of [9] is a particular case of our approach where the sub-matrix is reduced to the upper left coefficients. Note that similar ideas are used in [8] in the context of string rewriting.

Section 2 recalls preliminary notions on orderings by interpretation. Section 3 presents the extension we propose and the proof of its correctness. Sections 3.3 and 4 illustrate our method on an example and summarizes some experiments on the *termination problems database (TPDB)*. Finally we present future work and conclude in Section 5.

2 Preliminaries

We assume that the reader is familiar with basic concepts of term rewriting [6, 1, 7, 5] and termination. The typical approach of an (automated) termination prover is to transform recursively a well foundation problem to an equivalent set of simpler problems until we get problems that are proved directly using a well-founded ordering (pair). There are two kinds of desirable orderings: weakly monotonic (used for example to order dependency pairs) and strictly monotonic (rewriting rules) ones. We can use *interpretations* to produce such orderings. We recall notions of *ordering pairs* and *homomorphic interpretations*.

Ordering pairs. An ordering pair is a pair $(>, \geq)$ of relations over $T(\mathscr{F}, X)$ such that: 1) $\geq$ is a quasi-ordering, i.e. reflexive and transitive, 2) $>$ is a strict ordering, i.e. irreflexive and transitive, and 3) $\geq \cdot > \subseteq >$ (notice that $>$ is *not* the strict part of $\geq$). An ordering pair $(>, \geq)$ is *well-founded* if there is no infinite strictly decreasing sequence $t_1 > t_2 > \dots$. An ordering pair $(>, \geq)$ is *weakly monotonic* if for all terms t and u and any symbol f, $t \geq u \rightarrow f(\dots t \dots) \geq f(\dots u \dots)$. An ordering pair $(>, \geq)$ is *strictly monotonic* if for all terms t and u and any symbol f, $t > u \rightarrow f(\dots t \dots) > f(\dots u \dots)$.

Orderings by Interpretation. In the sequel, we suppose a non empty set D (domain), a quasi-ordering $\geq_D$ on D, and $>_D = \geq_D - \leq_D$. Therefore $(>, \geq)$ is an ordering pair.

[*] Work partially supported by A3PAT project of the French ANR (ANR-05-BLAN-0146-01).

Definition 2.1. *A* homomorphic interpretation *is a function φ that takes a symbol f and returns a valuation function $[f]_\varphi : D^n \to D$, where n is the arity of f. We define the homomorphic interpretation $\varphi(t)$ of a (possibly non closed) term t as a function from valuation functions to D (i.e. $\varphi(t) : (X \to D) \to D$) by induction on t as follows: $\varphi(x)(v) = v(x)$ and $\varphi(f(t_1,\dots,t_n))(v) = [f]_\varphi(\varphi(t_1)(v),\dots,\varphi(t_n)(v))$*

Definition 2.2. *We define the ordering pair $(\succeq_\varphi, \succ_\varphi)$ on terms by: $s \succeq_\varphi t$ iff $\forall v \in (X \to D), \varphi(s)(v) \geq_D \varphi(t)(v)$ and $s \succ_\varphi t$ iff $\forall v \in (X \to D), \varphi(s)(v) >_D \varphi(t)(v)$.*

Theorem 2.3. *Let φ be a homomorphic interpretation on D, then*

- *$(\succeq_\varphi, \succ_\varphi)$ is stable by substitution. It is also well-founded if $(\geq_D, >_D)$ is.*
- *if the following property (strict monotonicity on each argument) holds for all symbol f: $\forall x,y \in D, x >_D y \Rightarrow [f]_\varphi(\dots x \dots) >_D [f]_\varphi(\dots y \dots)$, then $(\succeq_\varphi, \succ_\varphi)$ is strictly monotonic.*
- *if the following property (weak monotonicity on each argument) holds for all symbol f of arity n: $\forall x,y \in D, x \geq_D y \Rightarrow [f]_\varphi(\dots x \dots) \geq_D [f]_\varphi(\dots y \dots)$, then $(\succeq_\varphi, \succ_\varphi)$ is weakly monotonic.*

Matrix interpretation. The main idea of matrix interpretation in [9] is to define homomorphic interpretations using linear mappings represented by matrices. The ordering pair on $\mathbb{N}^d$, which we note $(\geq_{\mathbb{N}^d}, >_{\mathbb{N}^d})$ is defined as follows: $(u_i) \geq_{\mathbb{N}^d} (v_i)$ iff $\forall i, u_i \geq_\mathbb{N} v_i$ and $(u_i) >_{\mathbb{N}^d} (v_i)$ iff $\forall i, u_i \geq_\mathbb{N} v_i$ and $u_1 >_\mathbb{N} v_1$. As homomorphic interpretations defined by polynomials over matrices may not be monotonic, [9] proposes a restriction on the form of vectors and matrices to ensure strict monotonicity: the upper-left coefficient of vectors and matrices must be strictly positive.

3 Generalized matrix interpretation

We define a family of interpretations parametrized by the set E of column and line indexes used by the strict ordering. We use polynomials with *matrix* constants instead of vectors ($D = \mathbb{N}^{d\times d}$). This corresponds to the usual notion of polynomials over matrices. Results of [9] can be considered as consequences of what follows by forcing adequate columns to be null.

3.1 Ordering and interpretation

Definition 3.1. *Let $m, m' \in \mathbb{N}^{d\times d}$ and $E \subseteq \{1,...,d\}$, We define $\geq_{\mathbb{N}^{d\times d}}$ and $>^E_{\mathbb{N}^{d\times d}}$ on $\mathbb{N}^{d\times d}$ as: $m \geq_{\mathbb{N}^{d\times d}} m'$ iff $\forall i,j \in [1..d], m_{ij} \geq_\mathbb{N} m'_{ij}$, and $m >^E_{\mathbb{N}^{d\times d}} m'$ iff $\forall i,j \in [1..d], m_{ij} \geq_\mathbb{N} m'_{ij} \wedge \exists i,j \in E, m_{ij} >_\mathbb{N} m'_{ij}$.*

Lemma 3.2. *$(\geq_{\mathbb{N}^{d\times d}}, >^E_{\mathbb{N}^{d\times d}})$ is a well-founded ordering pair.*

Definition 3.3. *Let $E \subseteq \{1,...,d\}$, we call an E-position in a matrix $m \in \mathbb{N}^{d\times d}$ a position m_{ij} where $i \in E$ and $j \in E$. We also call E-columns the E-positions belonging to the same column. We say that m is* E-compatible *iff each E-column is non null, i.e. at least one E-position on each E-column is non null.*

For example the following matrix A is $\{1,3\}$-compatible whereas B is not: $A = \begin{pmatrix} \underline{\mathbf{0}} & 1 & \underline{\mathbf{0}} \\ 0 & 0 & 0 \\ \underline{\mathbf{2}} & 3 & \underline{\mathbf{1}} \end{pmatrix}, B = \begin{pmatrix} \underline{\mathbf{1}} & 1 & \underline{\mathbf{0}} \\ 0 & 0 & 0 \\ \underline{\mathbf{2}} & 3 & \underline{\mathbf{0}} \end{pmatrix}$.

Definition 3.4. *Given a signature Σ and a dimension $d \in \mathbb{N}$, a* matrix interpretation *φ is a homomorphic interpretation that takes a symbol f of arity n and returns a valuation function of the form: $[f]_\varphi(m_1,\dots,m_n) = F_1 m_1 + \cdots + F_n m_n + F_{n+1}$ where $F_i \in \mathbb{N}^{d\times d}$ and $m_1,\dots,m_n$ take their values in $\mathbb{N}^{d\times d}$. If the matrices F_i, **except** F_{n+1}, are E-compatible then the interpretation is said to be E-compatible and we call it an E-*interpretation.

Definition 3.5. *$(\succeq_\varphi, \succ^E_\varphi)$ is defined from $(\geq_{\mathbb{N}^{d\times d}}, >^E_{\mathbb{N}^{d\times d}})$ as in Definitions 2.2 (with $D = \mathbb{N}^{d\times d}$).*

Lemma 3.6. *Let φ be a matrix interpretation, then $(\succeq_\varphi, \succ^E_\varphi)$ is 1) stable wrt substitution, 2) well-founded, 3) weakly monotonic and 4) strictly monotonic if moreover φ is E-compatible.*

3.2 Proving termination

To prove termination we need to compare matrix interpretations of terms with $\succ_\varphi$ and $\succeq_\varphi$. Interpretations can be computed by developing and comparing (linear) polynomials, which is decidable as stated below:

Lemma 3.7. *Let φ be a matrix interpretation and t a term with free variables $x_1 \dots x_n$, then there exist $n+1$ matrices $M_1 \dots M_{n+1}$ such that $\varphi(t)(v) = M_1 v(x_1) + \cdots + M_n v(x_n) + M_{n+1}$. If moreover φ is E-compatible, then $M_1 \dots M_n$ are E-compatible.*

Lemma 3.8. *Let t and u be terms such that $\varphi(t)(v) = L_1 v(x_1) + \cdots + L_k v(x_k) + L_{k+1}$ and $\varphi(u)(v) = R_1 v(x_1) + \cdots + R_k v(x_k) + R_{k+1}$. If $\forall 1 \leq i \leq k+1, L_i \geq_{\mathbb{N}^{d\times d}} R_i$ then $\varphi(t)(v) \geq_{\mathbb{N}^{d\times d}} \varphi(u)(v)$ for any valuation $\alpha : \mathbb{N}^k \to \mathbb{N}$. If moreover $L_{k+1} >^E_{\mathbb{N}^{d\times d}} R_{k+1}$, then $\varphi(t)(v) >^E_{\mathbb{N}^{d\times d}} \varphi(u)(v)$ for any valuation $\alpha : \mathbb{N}^k \to \mathbb{N}$.*

3.3 Example

In this example matrix coefficients are forced to be 0 or 1. Within this bounds there exists a $\{1,2\}$-interpretation ($[.]_\varphi$ below) which allows to prove termination of the following system by Manna-Ness criterion, but no $\{1\}$-interpretations. It is worth noticing that this example can be solved by a $\{1\}$-interpretation with a higher bound, but at the price of a greater search space.

$$h(0) \to 0 \qquad h(s(s(x))) \to s(h(x)) \qquad l(s(0)) \to 0 \qquad l(s(s(x))) \to s(l(s(h(x))))$$

$$[h]_\varphi(x) = \begin{pmatrix} 10 \\ 01 \end{pmatrix} x + \begin{pmatrix} 01 \\ 00 \end{pmatrix}, [l]_\varphi(x) = \begin{pmatrix} 11 \\ 01 \end{pmatrix} x + \begin{pmatrix} 00 \\ 00 \end{pmatrix}, [s]_\varphi(x) = \begin{pmatrix} 10 \\ 01 \end{pmatrix} x + \begin{pmatrix} 00 \\ 11 \end{pmatrix}, [0]_\varphi = \begin{pmatrix} 00 \\ 00 \end{pmatrix}$$

4 Experiments

The proof search is an adaptation of the method of [9] for generating termination proofs. The main differences are the choice of an E, the treatment of E-compatibility and the ordering constraints using E. *Remark*: Given a set of constraints S, if there is a E-interpretation φ that solves S then any *simultaneous* permutation of lines and columns of all matrices of φ is a E'-interpretation for some E' which also solves S. Therefore it is enough to try only sets E of the form $\{1,\dots,n\}$ where $1 \leq n \leq d$.

The benchmarks were made on the Tpdb database with C*i*ME. The solving part of the criterion (MN=Manna-Ness, DP=Dependency pairs, LEX=Lexicographic combination, DPG=graph refinement of DP) were made by translation to the SAT solver `minisat2`. Figure 1 highlights differences depending on the size of E. Notice that, for the sake of comparison, inside one criterion *we only compare problems that did not timeout for any $|E|$*. Therefore this results do not compare the solving times.

We see that for strictly monotonic orderings the sets of problems solved are not included in each other as shown by line $|E| = 1$ or 2, therefore both sizes for E are worth trying. This is not the case for weakly monotonic orderings of DP criterion where the maximum size of E is the best. Finally we see that the choice of $|E|$ is not pertinent when considering graph refinement of dependency pairs.

5 Conclusion and Future work

As shown in previous section our approach generalizes the original matrix interpretations. It also allows for more generalization. First it should naturally extend to other refinement of matrix interpretations

Criterion	MN			LEX			DP			DPG		
coef. bound	1	2	3	1	2	3	1	2	3	1	2	3
$\|E\| = 1$	115	232	250	343	386	380	342	451	471	578	590	594
$\|E\| = 2$	167	236	252	296	369	377	469	467	480	578	590	594
$\|E\| = 1$ or 2	170	252	259	354	393	387	469	467	480	578	590	594

Figure 1: Experiments with 2×2 matrices

such as arctic interpretations (where the usual plus/times operations are generalized to an arbitrary semi-ring [10]). Second our approach using true polynomials over matrices, instead of mixing matrices and vectors, should allow for *non linear polynomials over matrices*.

Our matrix interpretations have been implemented in an early C*i*ME-3 prototype. Interpretations are found by solving linear constraints over matrix coefficients. We produce SAT constraints for each size of E, which forces us to call SAT solving once per size of E. It should be possible to build efficient SAT constraints corresponding to all possible sizes of E.

Another important point is that our implementation *produces proof traces*. We are currently adapting the translation of these traces into *proof certificates* [3] for verification by the Coq proof assistant.

References

[1] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[2] Ahlem Ben Cherifa and Pierre Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9(2):137–159, 1987.

[3] Évelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Certification of automated termination proofs. In Boris Konev and Frank Wolter, editors, *6th International Symposium on Frontiers of Combining Systems (FroCos 07)*, volume 4720 of *Lecture Notes in Artificial Intelligence*, pages 148–162, Liverpool,UK, September 2007. Springer-Verlag.

[4] Évelyne Contejean, Claude Marché, Ana Paula Tomás, and Xavier Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):325–363, 2005.

[5] Nachum Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, March 1982.

[6] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. North-Holland, 1990.

[7] Jürgen Giesl, editor. *Term Rewriting and Applications*, volume 3467 of *Lecture Notes in Computer Science*, Nara, Japan, April 2005. Springer-Verlag.

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

[9] J. Waldmann J. Endrullis and H. Zantema. Matrix interpretations for proving termination of term rewriting. *Journal of Automated Reasoning*, 2007.

[10] Adam Koprowski and Johannes Waldmann. Arctic termination ...below zero. In Andrei Voronkov, editor, *RTA*, volume 5117 of *Lecture Notes in Computer Science*, pages 202–216, Hagenberg, Austria, July 2008. Springer-Verlag.

[11] Dallas S. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Mathematics Department, Louisiana Tech. Univ., 1979. Available at `http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html`.

Type Systems for the Termination of Mobile Processes

Romain Demangeon
ENS Lyon, Université de Lyon, CNRS, INRIA – France

In this abstract, we address termination of concurrent systems, and especially systems allowing the definition of dynamically evolving structures: typically, new servers can be spawned at run-time, names newly created somewhere are sent elsewhere. Deciding termination for such systems is a challenging task.

We present several type systems for termination in the π-calculus. The soundness of such systems ensures that a typable term is terminating. We describe several results coming from joint work with D. Hirschkoff, D. Sangiorgi and N. Kobayashi [DHKS07, DHS08, DHS09]; these studies rely on [DS06] as starting point.

1 Weight-based Type Systems

We use as a formalism the standard polyadic π-calculus with only replicated inputs. We denote a polyadic input action by $a(\widetilde{x})$, a polyadic output action by $\overline{a}\langle\widetilde{v}\rangle$ and a replicated input by $!a(\widetilde{x}).P$. When we examine the semantics of the π-calculus, we notice that the size of a process performing a communication involving non replicated terms, like, e.g., $\overline{a}\langle\widetilde{v}\rangle.P \mid a(\widetilde{x}).Q \to P \mid Q[\widetilde{v}/\widetilde{x}]$ strictly decreases. Actually, the replication is the sole source of divergence for π processes: in a replicated communication $\overline{a}\langle\widetilde{v}\rangle.P \mid !a(\widetilde{x}).Q \to P \mid Q[\widetilde{v}/\widetilde{x}] \mid !a(\widetilde{x}).Q$, the persistence of $!a(\widetilde{x}).Q$, which allows us to model the behavior of a server, can let the size of the whole process increase. Terms P_0 and P_1 of Fig. 1 represent the kind of processes we want to reject, as they contain dangerous replications, which lead to diverging behaviours.

The restriction operator ν, allowing the dynamical creation of new names, contributes importantly to the expressiveness of the π-calculus. However, it is the source of several technical complications in the analyses we describe below. For the sake of clarity, we will not address here name restriction.

The first type system we present, called S1, is introduced in [DS06]. The main idea is to assign a level (a natural number) to every name used in the process. The main typing rule, to control replicated processes, is as follows (here and below, we give a simplified presentation of the typing rules, to ease readability):

$$\frac{\Gamma, \widetilde{x} : \widetilde{T} \vdash P : m \qquad m < n}{\Gamma \vdash !a^n(\widetilde{x}).P : 0} \qquad \text{(S1)}$$

For a process to be type-checked, the levels of the names appearing in output position in the continuation (P) must be smaller than the level of a. Here a^n means that a is given level n. The judgment $\Gamma \vdash Q : l$ means that Q is typable in the context Γ, and that the outputs in Q which do not appear under a replication are of level at most l.

Soundness of the type system is established as follows: take as a measure on processes the multiset of levels of all names appearing in output position not under a replication. One proves that for a type-checked process, this measure strictly decreases at each reduction step: this is insured by the above typing rule.

Complexity of inference. It is easy to see that the inference for S1 is polynomial, as it boils down to searching for cycles in a graph of domination (an edge between a, b represents the constraint $a > b$ between levels assigned to names). This algorithm is implemented in [Kob07], and in [Bou08], where a more expressive variant of S1, described in [DHS08] is also implemented.

$P_0 = !a(x).\overline{a}\langle x\rangle \qquad P_1 = !a(x).\overline{b}\langle x\rangle \mid !b(y).\overline{a}\langle y\rangle \qquad P_2 = !a(x).b(y).\overline{a}\langle x\rangle \qquad P_3 = !a(x,y).x.(\overline{a}\langle x,y\rangle \mid \overline{y})$

$P_4 = !a(x,y,z).x.(\overline{a}\langle x,y,z\rangle \mid \overline{y} \mid \overline{z}) \qquad P_5 = a(X).(\overline{a}\langle X\rangle \mid X) \qquad P_6 = \overline{a}\langle X \mapsto (X \mid X)\rangle \mid a(F).b(Y).F\lfloor Y\rfloor$

Figure 1: Motivating examples.

2 Allowing Forms of Recursion

The main disadvantage of S1, from the point of view of expressiveness, is that we reject a process as soon as it contains a recursion: a replicated input $!a(\widetilde{x}).P$ whose continuation contains an output on a cannot be type-checked, For example, the process P_2 of Fig. 1 is rejected by our type system (the level of a should be strictly greater than itself), although this process is not intrinsically divergent (it requires an input on a and an input on b to produce a single output on a). Several approaches have been explored to increase expressiveness by allowing some controlled recursions (processes like P_0 should still be rejected).

Multisets of names. The paper [DS06] proposes a more complex system, obtained by considering (replicated) input sequences as a whole: to type-check a process of the form $!a_1(\widetilde{x_1}).\dots.a_k(\widetilde{x_k}).P$ we compare the levels of the multiset of names $\{a_1,\dots,a_k\}$ (called the weight of $\{a_1,\dots,a_k\}$) with the weight the multiset of names appearing in output position (again, not under a replication) in P .

The main typing rule of this system, that we call S2, is as follows:

$$\frac{\Gamma, \widetilde{x_1} : \widetilde{T_1}, \dots, \widetilde{x_k} : \widetilde{T_k} \vdash P : M \qquad \{n_1,\dots,n_k\} >_{mul} M}{\Gamma \vdash !a_1^{n_1}(\widetilde{x_1}).\dots.a_k^{n_k}(\widetilde{x_k}).P : \emptyset} \qquad \text{(S2)}$$

Here $>_{mul}$ is a multiset comparison between multisets of integers, and the judgment $\Gamma \vdash Q : N$ means that Q is typable in the context Γ and that N is the multiset of the levels of all outputs not appearing under a replication in Q.

Complexity of inference. We prove in [DHKS07] that the type inference problem for this system is NP-complete. The fact that we are comparing two multisets of integers leads to a combinatorial number of possible level assignments and allows us to reduce the problem **1in3 SAT** to the problem of type inference.

We also give in [DHKS07] a variant of system S2, at least as expressive, which uses algebraic comparisons between multisets of names, instead of multiset comparisons. Type inference for this variant is polynomial.

Partial orders. However system S2 is still not expressive enough to type complex processes mimicking the behavior of list-like (a *symbol table* example is detailed in [DS06]) or tree-like (we give an example in [DHS08]) data structures. The action of propagating a request in such data structures is modelled in the π-calculus by processes like P_3 and P_4 respectively (Fig. 1 – in these examples, we omit the arguments received and sent on this node, for the sake of clarity): a firing of a replication trades an input on a name x (modelling the request on a node of the structure) for outputs on names y, z (y modelling a request on its successor in the list structure, and y,z modelling requests on its children in the tree structure). In a π-calculus encoding of such a dynamically evolving structure, the types of x,y and z are necessarily the same, and so are their levels.

In the case of the list sructure, in such a replication, the weight consumed is equal to the weight released. This motivates the definition of a more refined type system, S3, in [DS06]. System S3 uses a well-founded partial order between names. In a typed replication, either we have $\{n_1,\dots,n_k\} > M$ (as in S2), or $\{n_1,\dots,n_k\} = M$ and the partial order between names decreases: we are trading inputs for outputs of the same level, but going down in the partial order. In the case of P_3, the partial order will state that the name a dominates the name b.

In [DHS08], we define a type system which is expressive enough to type-check processes modelling the behaviour of the tree structure. In that case, the type system has to be modified in a non trivial way, since the weight associated to the process can *increase* when a replication is fired. This introduces some technicalities, in particular in relation with the control of the restriction operator.

3 Higher-order Mobile Calculi

In [DHS09], we adapt the ideas exposed above to the higher-order paradigm, where the form of recursion is different. The principles of the previous type system can no longer be applied, as they are based on controlling the replication operator; in higher-order calculi like HOPI$_2$, a version of the π-calculus where values carried on channels are processes, this operator is not required to obtain divergence. This is illustrated, e.g., by the diverging process $Q_5 = \overline{a}\langle P_5\rangle \mid P_5$ (P_5 is defined in Fig. 1).

We prove in [DHS09] that if we forbid recursive outputs, that is, the possibility for a channel a to carry processes containing outputs on a (Q_5 contains such a recursive output), we obtain a terminating system. Indeed, we can assign levels to names like in S1 and forbid the presence of outputs on levels higher than l in a message emitted on a name of level l. Here is a simplified version of the main typing rule of this system:

$$\frac{\Gamma \vdash P : k \qquad \Gamma \vdash Q : m \qquad k < n}{\Gamma \vdash \overline{a}^n\langle P\rangle.Q : \max(m,n)} \qquad \text{(S4)}$$

We write the judgment $\Gamma \vdash Q : l$ when Q is typable in the context Γ and the names in output position in Q are at most of level l.

Termination is enforced as there exists a measure (the multiset of levels of all names at top-level, i.e., not appearing in a process in message position) which decreases at every reduction step: the several copies of the message process spawned contain only outputs whose levels are smaller than the level output consumed.

We notice some similarities between this typing rule for output actions and the typing rule for replicated processes in S1. Indeed, we prove in [DHS09] that nearly every typable HOPi$_2$ process (precisely, every process which does not contain a subprocess of the form $a(X).P$ with an output $\overline{b}\langle X\rangle$ in P) is encoded (using the standard encoding from [SW01]) into a first order π-calculus process, which is typable in system S1.

We are able to adapt system S4 to more complex calculi, such as HOPi$_\omega$, where values carried on channels can be higher-order functions, whose codomain is the set of processes. An example is given by process P_6 from Fig. 1, where a function that duplicates a process is transmitted on channel a. Other extensions of S4, to calculi combining higher-order features and concurrency, are currently being studied.

References

[Bou08] P. Boutillier. Implementation of a hybrid type system for termination in the π-calculus. Training period report, ENS Lyon, 2008.

[DHKS07] R. Demangeon, D. Hirschkoff, N. Kobayashi, and D. Sangiorgi. On the complexity of termination inference for processes. In *Proc. of TGC'07*, volume 4912 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2007.

[DHS08] R. Demangeon, D. Hirschkoff, and D. Sangiorgi. Static and dynamic typing for the termination of mobile processes. In *Proc. of TCS'08*. Springer Verlag, 2008.

[DHS09] R. Demangeon, D. Hirschkoff, and D. Sangiorgi. Termination in Higher-Order Concurrent Calculi. In *Proc. of FSEN'09*, 2009. to appear.

[DS06] Y. Deng and D. Sangiorgi. Ensuring Termination by Typability. *Information and Computation*, 204(7):1045–1082, 2006.

[Kob07] N. Kobayashi. TyPiCal: Type-based static analyzer for the Pi-Calculus. available from `http://www.kb.ecei.tohoku.ac.jp/~koba/typical/`, 2007.

[SW01] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

Termination of Integer Term Rewriting[*]

C. Fuhs, J. Giesl, M. Plücker
LuFG Informatik 2
RWTH Aachen University
Aachen, Germany

P. Schneider-Kamp
Dept. of Mathematics & CS
University of Southern Denmark
Odense, Denmark

S. Falke
CS Department
University of New Mexico
Albuquerque, NM, USA

Abstract

When using rewrite techniques for termination analysis of programs, a main problem are pre-defined data types like integers. We extend term rewriting by built-in integers and adapt the dependency pair framework to prove termination of *integer term rewriting* automatically.

1 Introduction

Rewrite techniques and tools have been successfully applied to prove termination automatically for different programming languages. The advantage of rewrite techniques is that they are very powerful for algorithms on user-defined data structures, since they can generate well-founded orders comparing arbitrary forms of terms. But in contrast to techniques for termination of imperative programs, rewrite techniques do not support data structures like integers which are pre-defined in most programming languages.

To solve this problem, we extend TRSs by built-in integers and adapt the popular dependency pair (DP) framework for termination of TRSs to integers in Sect. 2. In Sect. 3, we improve the main processor of the adapted DP framework by considering *conditions* and explain how to generate suitable orders for termination proofs of integer TRSs (ITRSs). Sect. 4 evaluates our implementation in AProVE [6].

2 Integer Dependency Pair Framework

We represent each integer by a pre-defined constant of the same name. So the signature is split into two disjoint subsets $\mathcal{F}$ and $\mathcal{F}_{int}$. $\mathcal{F}_{int}$ contains $\mathbb{Z} = \{0, 1, -1, \ldots\}$, $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$, and pre-defined operations $\mathit{ArithOp} = \{+, -, *, /, \%\}$, $\mathit{RelOp} = \{>, \geqslant, <, \leqslant, ==, !=\}$, and $\mathit{BoolOp} = \{\wedge, \Rightarrow\}$. An *ITRS* $\mathcal{R}$ is a (finite) TRS over $\mathcal{F} \uplus \mathcal{F}_{int}$ where for all rules $\ell \to r$, we have $\ell \in \mathcal{T}(\mathcal{F} \cup \mathbb{Z} \cup \mathbb{B}, \mathcal{V})$ and $\ell \notin \mathbb{Z} \cup \mathbb{B}$. The rewrite relation $\hookrightarrow_{\mathcal{R}}$ of an ITRS $\mathcal{R}$ is defined as $\overset{\mathsf{i}}{\to}_{\mathcal{R} \cup PD}$, where PD is an infinite set of rules to evaluate the pre-defined operations. For example, PD contains the rules $2*21 \to 42$, $42 \geqslant 23 \to \mathsf{true}$, and $\mathsf{true} \wedge \mathsf{false} \to \mathsf{false}$. So pre-defined operations can only be evaluated if both their arguments are integers resp. Booleans. For example, consider the ITRSs $\mathcal{R}_1 = \{(1), (2), (3)\}$ where $\mathsf{sum}(x,y)$ computes $\sum_{i=y}^{x} i$.

$$\mathsf{sum}(x,y) \to \mathsf{sif}(x \geqslant y, x, y) \;\; (1) \qquad \mathsf{sif}(\mathsf{true}, x, y) \to y + \mathsf{sum}(x, y+1) \;\; (2) \qquad \mathsf{sif}(\mathsf{false}, x, y) \to 0 \;\; (3)$$

The term $\mathsf{sum}(1,1)$ can be rewritten as follows: $\underline{\mathsf{sum}(1,1)} \hookrightarrow_{\mathcal{R}_1} \mathsf{sif}(\underline{1 \geqslant 1}, 1, 1) \hookrightarrow_{\mathcal{R}_1} \underline{\mathsf{sif}(\mathsf{true}, 1, 1)} \hookrightarrow_{\mathcal{R}_1} 1 + \mathsf{sum}(1, \underline{1+1}) \hookrightarrow_{\mathcal{R}_1} 1 + \underline{\mathsf{sum}(1,2)} \hookrightarrow_{\mathcal{R}_1} 1 + \mathsf{sif}(\underline{1 \geqslant 2}, 1, 2) \hookrightarrow_{\mathcal{R}_1} 1 + \underline{\mathsf{sif}(\mathsf{false}, 1, 2)} \hookrightarrow_{\mathcal{R}_1} \underline{1+0} \hookrightarrow_{\mathcal{R}_1} 1$.

We extend the *DP framework* [1, 5, 7, 8] to ITRSs. The main problem is that proving innermost termination of $\mathcal{R} \cup PD$ *automatically* is not straightforward, as PD is infinite. Therefore, we will not consider the rules PD explicitly, but integrate their handling in the processors of the DP framework. The resulting method should be as powerful as possible for term rewriting on integers, but at the same time it should have the full power of the original DP framework when dealing with other function symbols.

For an ITRS $\mathcal{R}$, the *defined* symbols $\mathcal{D}$ are the root symbols of left-hand sides of rules in $\mathcal{R} \cup PD$, i.e., $\mathcal{D}$ also includes $\mathit{ArithOp} \cup \mathit{RelOp} \cup \mathit{BoolOp}$. However, we ignore these symbols when building DPs.

Definition 1 (DP). *For all $f \in \mathcal{D} \setminus \mathcal{F}_{int}$, we introduce a fresh* tuple symbol *F with the same arity. If $t = f(t_1, ..., t_n)$, let $t^\sharp = F(t_1, ..., t_n)$. If $\ell \to r \in \mathcal{R}$ for an ITRS $\mathcal{R}$ and t is a subterm of r with* $\mathrm{root}(t) \in$ *$\mathcal{D} \setminus \mathcal{F}_{int}$, then $\ell^\sharp \to t^\sharp$ is a* dependency pair *of $\mathcal{R}$. $DP(\mathcal{R})$ is the set of all DPs.*

For example, $DP(\mathcal{R}_1) = \{\mathsf{SUM}(x,y) \to \mathsf{SIF}(x \geqslant y, x, y) \;\; (4),\;\; \mathsf{SIF}(\mathsf{true}, x, y) \to \mathsf{SUM}(x, y+1) \;\; (5)\}$.

[*]Supported by the DFG grant GI 274/5-2 and by the G.I.F. grant 966-116.6.

For any TRS $\mathcal{P}$ and ITRS $\mathcal{R}$, a *$\mathcal{P}$-chain* is a sequence of variable renamed pairs $s_1 \to t_1, s_2 \to t_2, \dots$ from $\mathcal{P}$ such that there is a substitution σ (with possibly infinite domain) where $t_i\sigma \hookrightarrow_{\mathcal{R}}^* s_{i+1}\sigma$ and $s_i\sigma$ is in normal form w.r.t. $\hookrightarrow_{\mathcal{R}}$, for all i. We get the following corollary from the standard results on DPs.

Corollary 2. *An ITRS $\mathcal{R}$ is terminating (w.r.t. $\hookrightarrow_{\mathcal{R}}$) iff there is no infinite DP($\mathcal{R}$)*-chain.

Termination techniques are now called *DP processors* and they operate on sets of DPs (called *DP problems*). A DP processor *Proc* takes a DP problem as input and returns a set of new DP problems which have to be solved instead. *Proc* is *sound* if for all DP problems $\mathcal{P}$ with an infinite $\mathcal{P}$-chain there is also a $\mathcal{P}' \in Proc(\mathcal{P})$ with an infinite $\mathcal{P}'$-chain. Termination proofs start with the initial DP problem $DP(\mathcal{R})$. Then the DP problem is simplified repeatedly by sound DP processors. If all resulting DP problems have been simplified to $\varnothing$, then termination is proved. Many processors (like the *(estimated) dependency graph processor* [1, 5]) do not rely on the rules of the TRS, but just on the DPs and on the defined symbols. Therefore, they can also be directly applied for ITRSs.

But an adaption is non-trivial for one of the most important processors, the *reduction pair processor*. For a DP problem $\mathcal{P}$, this processor generates constraints which should be satisfied by a suitable order on terms. Here, we consider orders based on *max-polynomial interpretations* [3]. The set of *max-polynomials* is the smallest set containing the integers $\mathbb{Z}$, the variables, and $p+q$, $p*q$, and $\max(p,q)$ for all max-polynomials p and q. A *max-polynomial interpretation Pol* maps every n-ary function symbol f to a max-polynomial f_{Pol} over n variables $x_1,\dots,x_n$. This mapping is extended to terms as usual.

Consider the interpretation *Pol* where $\mathsf{SUM}_{Pol} = x_1 - x_2$, $\mathsf{SIF}_{Pol} = x_2 - x_3$, $+_{Pol} = x_1 + x_2$, $n_{Pol} = n$ for all $n \in \mathbb{Z}$, and $\geqslant_{Pol} = \mathsf{true}_{Pol} = \mathsf{false}_{Pol} = 0$. For any term t and position π in t, t is *$\succsim_{Pol}$-dependent* on π iff there exist terms u,v where $t[u]_\pi \not\approx_{Pol} t[v]_\pi$. Here, $\approx_{Pol} = \succsim_{Pol} \cap \precsim_{Pol}$. So in our example, $\mathsf{SIF}(b,x,y)$ is $\succsim_{Pol}$-dependent on 2 and 3, but not on 1. A term t is *$\succsim_{Pol}$-increasing* on π iff $u \succsim_{Pol} v$ implies $t[u]_\pi \succsim_{Pol} t[v]_\pi$ for all terms u,v. So $\mathsf{SIF}(b,x,y)$ is $\succsim_{Pol}$-increasing on 1 and 2, but not on 3.

The reduction pair processor requires that all DPs in $\mathcal{P}$ are strictly or weakly decreasing and all *usable rules* $\mathcal{U}_{\mathcal{R}\cup PD}(\mathcal{P})$ are weakly decreasing. Then one can delete all strictly decreasing DPs. The *usable rules* [1, 7] include all rules that can reduce terms in $\succsim_{Pol}$-dependent positions of $\mathcal{P}$'s right-hand sides when instantiating their variables with normal forms. Moreover, as $\succsim_{Pol}$ is not monotonic in general, we require that defined symbols only occur on $\succsim_{Pol}$-increasing positions of right-hand sides.

When using interpretations into the integers, $\succ_{Pol}$ is not well founded. However, for any bound, there is no infinite $\succ_{Pol}$-decreasing sequence that remains greater than the bound. Hence, the reduction pair processor transforms a DP problem into *two* new problems. As before, the first problem results from removing all strictly decreasing DPs. The second DP problem results from removing all DPs $s \to t$ from $\mathcal{P}$ that are *bounded from below*, i.e., DPs which satisfy the inequality $s \succsim \mathsf{c}$ for a fresh constant c.

However, there are two problems: (i) *PD* is infinite and thus, there are usually infinitely many usable rules, which is a problem for the automation. (ii) Defined symbols like $+$ often occur on non-$\succsim_{Pol}$-increasing positions (e.g., in the right-hand side of (5) when using *Pol* above). To solve these problems, we now restrict ourselves to so-called *I-interpretations* where $n_{Pol} = n$ for all $n \in \mathbb{Z}$, $+_{Pol} = x_1 + x_2$, $-_{Pol} = x_1 - x_2$, $*_{Pol} = x_1 * x_2$, $\%_{Pol} = |x_1|$, and $/_{Pol} = |x_1| - \min(|x_2| - 1, |x_1|)$. We say that an I-interpretation is *proper* for a term t if all defined symbols except $+$, $-$, and $*$ only occur on $\succsim_{Pol}$-increasing positions of t and if symbols from *RelOp* only occur on $\succsim_{Pol}$-independent positions of t.

The concept of *proper* I-interpretations ensures that we can disregard the (infinitely many) usable rules for the symbols from *RelOp* and that the symbols "/" and "%" only have to be estimated "upwards". Moreover, we may allow $+$, $-$, and $*$ on arbitrary positions and we only have to regard the usable rules w.r.t. $\mathcal{R} \cup BO$. Here, *BO* are the (finitely many) rules for the symbols $\wedge$ and $\Rightarrow$ in *BoolOp*.

Theorem 3 (Reduction Pair Processor for ITRSs)**.** *Let $\mathcal{R}$ be an ITRS, Pol be an I-interpretation, and $\mathcal{P}_{bound} = \{s \to t \in \mathcal{P} \mid s \succsim_{Pol} \mathsf{c}\}$ for a fresh constant* c*. Then the following processor Proc is sound.*

$$Proc(\mathcal{P}) = \begin{cases} \{\mathcal{P}\setminus\succ_{Pol},\ \mathcal{P}\setminus\mathcal{P}_{bound}\}, & \textit{if } \mathcal{P}\subseteq\succsim_{Pol}\cup\succ_{Pol},\ \mathcal{U}_{\mathcal{R}\cup BO}(\mathcal{P})\subseteq\succsim_{Pol}, \\ & \textit{and Pol is proper for all right-hand sides of } \mathcal{P}\cup\mathcal{U}_{\mathcal{R}}(\mathcal{P}) \\ \{\mathcal{P}\}, & \textit{otherwise} \end{cases}$$

To solve the DP problem $\mathcal{P} = \{(4),(5)\}$, we use an I-interpretation *Pol* where $\mathsf{SUM}_{Pol} = x_1 - x_2$ and $\mathsf{SIF}_{Pol} = x_2 - x_3$. We have $\mathcal{U}_{\mathcal{R}\cup BO}(\mathcal{P}) = \varnothing$, as the $+$- and $\geqslant$-rules are not included in $\mathcal{R}\cup BO$. The DP (5) is strictly decreasing, but no DP is bounded, since $\mathsf{SUM}(x,y) \not\succsim_{Pol} \mathsf{c}$ and $\mathsf{SIF}(\mathsf{true},x,y) \not\succsim_{Pol} \mathsf{c}$ for any value of c_{Pol}. Thus, the processor returns the problems $\{(4)\}$ and $\{(4),(5)\}$, i.e., it does not simplify $\mathcal{P}$.

3 Conditional Constraints and Generation of I-Interpretations

To solve this problem, we consider *conditions* for inequalities like $s \underset{(\sim)}{\succ} t$ or $s \succsim \mathsf{c}$. So to include (4) in $\mathcal{P}_{bound}$, we do not demand $\mathsf{SUM}(x,y) \succsim \mathsf{c}$ for *all* x and y. It suffices to require the inequality only for those instantiations of x and y which can be used in chains. So we require $\mathsf{SUM}(x,y) \succsim \mathsf{c}$ only for instantiations σ where (4)'s instantiated right-hand side $\mathsf{SIF}(x \geqslant y,x,y)\sigma$ reduces to an instantiated left-hand side $u\sigma$ for some DP $u \to v$ where $u\sigma$ is in normal form. Here, $u \to v$ should again be variable renamed. As our DP problem contains two DPs (4) and (5), we get the following two *conditional constraints* (by considering all $u \to v \in \{(4),(5)\}$). We include (4) in $\mathcal{P}_{bound}$ if both constraints are satisfied.

$$\mathsf{SIF}(x \geqslant y,x,y) = \mathsf{SUM}(x',y') \;\Rightarrow\; \mathsf{SUM}(x,y) \succsim \mathsf{c} \quad (6) \qquad \mathsf{SIF}(x \geqslant y,x,y) = \mathsf{SIF}(\mathsf{true},x',y') \;\Rightarrow\; \mathsf{SUM}(x,y) \succsim \mathsf{c} \quad (7)$$

To check whether conditional constraints are valid requires reasoning about reachability w.r.t. TRSs with infinitely many rules. To this end, we developed rules to simplify conditional constraints. These rules detect that (6)'s premise is unsatisfiable and hence, (6) is valid. Moreover, they transform (7) into

$$x \succsim y \qquad \Rightarrow \qquad \mathsf{SUM}(x,y) \succsim \mathsf{c} \tag{8}$$

To automate the reduction pair processor, one has to generate an I-interpretation satisfying a given conditional constraint. One starts with an *abstract* I-interpretation. It maps each function symbol to a max-polynomial with *abstract* coefficients. So we could use an abstract I-interpretation *Pol* where $\mathsf{SUM}_{Pol} = a_0 + a_1 x_1 + a_2 x_2$, $\mathsf{SIF}_{Pol} = b_0 + b_1 x_1 + b_2 x_2 + b_3 x_3$, and $\mathsf{c}_{Pol} = c_0$. Of course, the interpretation for the symbols in $\mathbb{Z} \cup \mathit{ArithOp}$ is fixed as for any I-interpretation (i.e., $+_{Pol} = x_1 + x_2$, etc.).

Then we transform the conditional constraint into an *inequality constraint* by replacing all atomic constraints "$s \succsim t$" by "$[s]_{Pol} \geqslant [t]_{Pol}$" and "$s \succ t$" by "$[s]_{Pol} \geqslant [t]_{Pol} + 1$". So "$\mathsf{SUM}(x,y) \succsim \mathsf{c}$" is transformed into "$a_0 + a_1 x + a_2 y \geqslant c_0$". Here, the abstract coefficients a_0, a_1, a_2, c_0 are implicitly existentially quantified and the variables $x, y \in \mathcal{V}$ are universally quantified. So (8) is transformed into

$$\forall x \in \mathbb{Z}, y \in \mathbb{Z} \quad (\, x \geqslant y \quad \Rightarrow \quad a_0 + a_1 x + a_2 y \geqslant c_0\,) \tag{9}$$

Now we remove universally quantified variables from such constraints. Rule (A) handles conditions "$x \geqslant p$" or "$p \geqslant x$" for a polynomial p without x. So (9) is transformed to $\forall y \in \mathbb{Z}, z \in \mathbb{N}$ $a_0 + a_1(y+z) + a_2 y \geqslant c_0$ (10).

A. **Eliminating Conditions**

$$\frac{\forall x \in \mathbb{Z},\ldots \quad (x \geqslant p \wedge \varphi \Rightarrow \psi)}{\forall z \in \mathbb{N},\ldots \quad (\varphi[x/p+z] \Rightarrow \psi[x/p+z])} \qquad \frac{\forall x \in \mathbb{Z},\ldots \quad (p \geqslant x \wedge \varphi \Rightarrow \psi)}{\forall z \in \mathbb{N},\ldots \quad (\varphi[x/p-z] \Rightarrow \psi[x/p-z])}$$

To replace all remaining quantifiers over $\mathbb{Z}$ by quantifiers over $\mathbb{N}$, Rule (B) splits the inequality constraint φ into the cases where y is positive resp. negative. Thus, (10) is transformed into the conjunction of (11) and (12).

B. **Split**

$$\frac{\forall y \in \mathbb{Z} \quad \varphi}{\forall y \in \mathbb{N} \quad \varphi \qquad \wedge \qquad \forall y \in \mathbb{N} \quad \varphi[y/-y]}$$

$$\forall y \in \mathbb{N}, z \in \mathbb{N} \quad a_0 + a_1(y+z) + a_2 y \geqslant c_0 \quad (11) \qquad \forall y \in \mathbb{N}, z \in \mathbb{N} \quad a_0 + a_1(-y+z) - a_2 y \geqslant c_0 \quad (12)$$

Note that (11) can be reformulated as "$\forall y \in \mathbb{N}, z \in \mathbb{N} \quad (a_1 + a_2)y + a_1 z + (a_0 - c_0) \geqslant 0$". So we now have to ensure non-negativeness of "polynomials" over variables like y and z ranging over $\mathbb{N}$, where the "coefficients" are polynomials like "$a_1 + a_2$" over the abstract variables. To this end, it suffices to require that these "coefficients" are $\geqslant 0$ [9]. In other words, now one can eliminate all universally quantified variables like y, z and transform (11) into the *Diophantine constraint* "$a_1 + a_2 \geqslant 0 \wedge a_1 \geqslant 0 \wedge a_0 - c_0 \geqslant 0$".

To search for abstract coefficients that satisfy the resulting Diophantine constraints, one fixes upper and lower bounds for their values. Then one can translate such Diophantine constraints into a SAT problem which can be handled by SAT solvers efficiently [2]. The constraints resulting from the initial inequality constraint (9) are for example satisfied by $a_0 = 0$, $a_1 = 1$, $a_2 = -1$, and $c_0 = 0$. With these values, the abstract interpretation $a_0 + a_1 x_1 + a_2 x_2$ for SUM is turned into the concrete interpretation $x_1 - x_2$. With the resulting concrete I-interpretation *Pol*, we would have $\mathscr{P}_{\succ} = \{(5)\}$ and $\mathscr{P}_{bound} = \{(4)\}$. The reduction pair processor of Thm. 3 would therefore transform the initial DP problem $\mathscr{P} = \{(4),(5)\}$ into the two problems $\mathscr{P} \setminus \mathscr{P}_{\succ} = \{(4)\}$ and $\mathscr{P} \setminus \mathscr{P}_{bound} = \{(5)\}$. Both are easy to solve.

4 Experiments and Conclusion

We adapted the DP framework to ITRSs. To evaluate our approach, we implemented it in AProVE [6] and tested it on a data base of 117 ITRSs containing also numerous examples from papers on termination of imperative programs. With a timeout of 1 minute per example, the new version of AProVE proves termination of 104 examples (88.9 %). We also tested the previous version of AProVE (AProVE08) and the tool $\mathsf{T_TT_2}$ [10] that do not support built-in integers. Here, we converted integers into terms constructed with 0, s, pos, and neg (e.g., -1 is represented as "neg(s(0))") and we added rules for pre-defined operations on integers in this representation. Although AProVE08 won the last *Termination Competition* 2008 for term rewriting and $\mathsf{T_TT_2}$ was second, AProVE08 resp. $\mathsf{T_TT_2}$ only proved termination of 24 (20.5 %) resp. 6 examples (5.1 %). This clearly shows the benefits of built-in integers in term rewriting. For details on our experiments and to run the new version of AProVE, we refer to `http://aprove.informatik.rwth-aachen.de/eval/Integer/`. A longer version of this paper will appear in [4].

References

[1] T. Arts, J. Giesl. Termination of term rewriting using dependency pairs. *Th. Comp. Sc.*, 236:133-178, 2000.

[2] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In *Proc. SAT'07*, LNCS 4501, pp. 340-354, 2007.

[3] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. Maximal termination. In *Proc. RTA'08*, LNCS 5117, pp. 110-125, 2008.

[4] C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving termination of integer term rewriting. In *Proc. RTA'09*, LNCS, 2009.

[5] J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. LPAR'04*, LNAI 3452, pp. 301-331, 2005.

[6] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. IJCAR'06*, LNAI 4130, pp. 281-286, 2006.

[7] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155-203, 2006.

[8] N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *I & C*, 199(1,2):172-199, 2005.

[9] H. Hong and D. Jakuš. Testing positiveness of polynomials. *J. Aut. Reasoning*, 21(1):23-38, 1998.

[10] M. Korp, C. Sternagel, H. Zankl, A. Middeldorp. Tyrolean Termination Tool 2. *Proc. RTA'09*, LNCS, 2009.

Mechanizing Proofs of Termination with Context-Sensitive Dependency Pairs [*]

Raúl Gutiérrez
DSIC, Universidad Politécnica de Valencia
Valencia, Spain
rgutierrez@dsic.upv.es

Salvador Lucas
DSIC, Universidad Politécnica de Valencia
Valencia, Spain
slucas@dsic.upv.es

Abstract

The dependency pairs approach, one of the most powerful techniques for proving termination of rewriting, has been recently adapted to be used for proving termination of Context-Sensitive Rewriting (CSR). The notion of Context-Sensitive Dependency Pair (CSDP) is different from the standard one in that collapsing dependency pairs (i.e., rules whose right-hand side is a variable) are considered. Although the implementation and practical use of CSDPs lead to a very powerful framework which improves the current state-of-the-art of methods for proving termination of CSR, handling collapsing pairs is not easy and often leads to impose heavy requirements over the base orderings which are used to achieve the proofs. A recent proposal removes collapsing pairs by transforming them into sets of new (standard) pairs. In this way, though, the complexity of the obtained dependency graph is heavily increased and the role of collapsing pairs for modeling context-sensitive computations gets lost. This leads to a less intuitive and accurate description of the termination behavior of the system.

In this paper, we show how to get the best of the two approaches, thus obtaining a more powerful dependency pair framework which hopefully fulfills all practical and theoretical expectations.

1 Introduction

In Context-Sensitive Rewriting (CSR, [6]), a *replacement map* μ satisfying $\mu(f) \subseteq \{1,...,\mathsf{ar}(f)\}$ for every function symbol f in the signature $\mathcal{F}$, is used to discriminate the argument positions on which the rewriting steps are allowed. In this way, a terminating behavior of (context-sensitive) computations with Term Rewriting Systems can be obtained. In [2, 3], Arts and Giesl's dependency pairs approach was adapted to CSR (see [4] for a more recent presentation). In [1], a transformation that replaces the *collapsing dependency pairs* (i.e., pairs whose right hand sides are variables, see [2]) by a new set of pairs that simulate their behavior was introduced. This new set of pairs is used to simplify the definition of context-sensitive dependency chain; but, on the other hand, we loose the intuition of what a collapsing pair means in a context-sensitive rewriting chain, and some processors which are based on them cannot be used anymore (see [4]). Furthermore, understanding the new dependency graph is harder.

Example 1. *Consider the context-sensitive term rewriting system (CS-TRS) in [1]*

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{gt}(\mathsf{0},y) & \to & \mathsf{false} & \mathsf{p}(\mathsf{0}) & \to & \mathsf{0}\\
\mathsf{gt}(\mathsf{s}(x),\mathsf{0}) & \to & \mathsf{true} & \mathsf{p}(\mathsf{s}(x)) & \to & x\\
\mathsf{gt}(\mathsf{s}(x),\mathsf{s}(y)) & \to & \mathsf{gt}(x,y) & \mathsf{minus}(x,y) & \to & \mathsf{if}(\mathsf{gt}(y,\mathsf{0}),\mathsf{minus}(\mathsf{p}(x),\mathsf{p}(y)),x)\\
\mathsf{if}(\mathsf{true},x,y) & \to & x & \mathsf{div}(\mathsf{0},\mathsf{s}(y)) & \to & \mathsf{0}\\
\mathsf{if}(\mathsf{false},x,y) & \to & y & \mathsf{div}(\mathsf{s}(x),\mathsf{s}(y)) & \to & \mathsf{s}(\mathsf{div}(\mathsf{minus}(x,y),\mathsf{s}(y)))
\end{array}$$

with $\mu(\mathsf{if}) = \{1\}$ and $\mu(f) = \{1,\ldots,\mathsf{ar}(f)\}$ for all other symbols f. If we follow the transformational

[*]Supported by the EU (FEDER) and MEC/MICINN projects TIN 2007-68093-C02-02 and TIN 2004-7943-C04-02.

definition in [1], we have the following dependency pairs (a new symbol U *is introduced):*

$\mathsf{GT}(\mathsf{s}(x),\mathsf{s}(y)) \to \mathsf{GT}(x,y)$	(1)	$\mathsf{M}(x,y) \to \mathsf{IF}(\mathsf{gt}(y,0),\mathsf{minus}(\mathsf{p}(x),\mathsf{p}(y)),x)$	(7)
$\mathsf{M}(x,y) \to \mathsf{GT}(y,0)$	(2)	$\mathsf{D}(\mathsf{s}(x),\mathsf{s}(y)) \to \mathsf{D}(\mathsf{minus}(x,y),\mathsf{s}(y))$	(8)
$\mathsf{D}(\mathsf{s}(x),\mathsf{s}(y)) \to \mathsf{M}(x,y)$	(3)	$\mathsf{U}(\mathsf{minus}(\mathsf{p}(x),\mathsf{p}(y))) \to \mathsf{M}(\mathsf{p}(x),\mathsf{p}(y))$	(9)
$\mathsf{IF}(\mathsf{true},x,y) \to \mathsf{U}(x)$	(4)	$\mathsf{U}(\mathsf{p}(x)) \to \mathsf{U}(x)$	(10)
$\mathsf{IF}(\mathsf{false},x,y) \to \mathsf{U}(y)$	(5)	$\mathsf{U}(\mathsf{p}(y)) \to \mathsf{U}(y)$	(11)
$\mathsf{U}(\mathsf{p}(x)) \to \mathsf{P}(x)$	(6)	$\mathsf{U}(\mathsf{minus}(x,y)) \to \mathsf{U}(x)$	(12)
		$\mathsf{U}(\mathsf{minus}(x,y)) \to \mathsf{U}(y)$	(13)

and the dependency graph has the unreadable aspect shown in Figure 1 (left). In contrast, if we consider

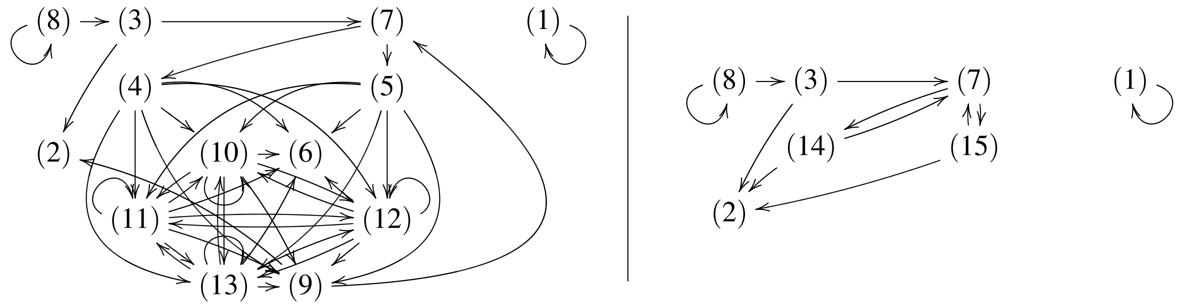


Figure 1: Dependency graph for Example 1 following [1] (left) and [4] (right)

the original definition of CSDP and CSDG in [3, 4], our set of dependency pairs is the following:

$\mathsf{GT}(\mathsf{s}(x),\mathsf{s}(y))$	$\to$	$\mathsf{GT}(x,y)$	(1)	$\mathsf{M}(x,y)$	$\to$	$\mathsf{IF}(\mathsf{gt}(y,0),\mathsf{minus}(\mathsf{p}(x),\mathsf{p}(y)),x)$	(7)
$\mathsf{M}(x,y)$	$\to$	$\mathsf{GT}(y,0)$	(2)	$\mathsf{D}(\mathsf{s}(x),\mathsf{s}(y))$	$\to$	$\mathsf{D}(\mathsf{minus}(x,y),\mathsf{s}(y))$	(8)
$\mathsf{D}(\mathsf{s}(x),\mathsf{s}(y))$	$\to$	$\mathsf{M}(x,y)$	(3)	$\mathsf{IF}(\mathsf{true},x,y)$	$\to$	x	(14)
				$\mathsf{IF}(\mathsf{false},x,y)$	$\to$	y	(15)

and the dependency graph is much more clear, see Figure 1 (right).

The work in [1] was motivated by the fact that mechanizing proofs of termination of CSR according to the results in [2] can be difficult due to the presence of collapsing dependency pairs. In this paper we solve this problem without loosing the modeling power of collapsing pairs.

2 Modeling Infinite Minimal Sequences

As shown in [1, 4], when a minimal infinite μ-rewrite sequence is considered, the minimal non-μ-terminating terms u (whose active strict subterms are μ-terminating, denoted $u \in \mathscr{M}_{\infty,\mu}$), which are introduced by the instantiated migrating variables x of a rule $l \to r$ (which are frozen in r but active in l, i.e., $x \in \mathscr{V}ar^{\mu}(r) \setminus \mathscr{V}ar^{\mu}(l)$) are instances of terms occurring in frozen positions in the right-hand sides of the rules (*hidden terms*) within a given (*hiding*) context. A term $t \in \mathscr{T}(\mathscr{F},\mathscr{X}) \setminus \mathscr{X}$ is a *hidden term* if there is a rule $l \to r \in R$ such that t is a frozen subterm of r. $\mathscr{H}\mathscr{T}(\mathscr{R},\mu)$ (or just $\mathscr{H}\mathscr{T}$, if $\mathscr{R}$ and μ are clear for the context) is the set of all hidden terms in $(\mathscr{R},\mu)$ and $\mathscr{N}\mathscr{H}\mathscr{T}(\mathscr{R},\mu)$ the set of narrowable hidden terms headed by a defined symbol. A function symbol f *hides position* i if $f(r_1,\ldots,r_i,\ldots,r_n) \in \mathscr{H}\mathscr{T}(\mathscr{R},\mu)$, $i \in \mu(f)$, and r_i contains a defined symbol or a variable at an active position (i.e., $\mathscr{P}os^{\mu}_{\mathscr{D}}(r_i) \cup \mathscr{P}os^{\mu}_{\mathscr{X}}(r_i) \neq \varnothing$). A context $C[\square]$ is *hiding* if $C[\square] = \square$ or $C[\square] = f(t_1,\ldots,t_{i-1},C'[\square],t_{i+1},\ldots,t_n)$, where f hides position i and $C'[\square]$ is a hiding context.

Example 2. *Following Example 1, the hidden terms are* $\mathsf{minus}(\mathsf{p}(x),\mathsf{p}(y))$, $\mathsf{p}(x)$ *and* $\mathsf{p}(y)$*, the hidden symbols* minus *and* p*, and* minus *hides positions* 1 *and* 2*, and* p *hides position* 1.

The following theorem shows how these notions are used and combined to model infinite context-sensitive rewrite sequences starting from (strongly) minimal non-μ-terminating terms which are non-μ-terminating terms t whose strict subterms are (all of them) μ-terminating, denoted $t \in \mathscr{T}_{\infty,\mu}$ [4].

Theorem 1 (Minimal sequence [5]). *Let $\mathcal{R} = (\mathcal{F}, R)$ be a TRS and $\mu \in M_{\mathcal{F}}$. For all $t \in \mathcal{T}_{\infty,\mu}$, there is an infinite sequence $t = t_0 \xrightarrow{>\Lambda}{}^{*}_{\mathcal{R},\mu} \sigma_1(l_1) \overset{\Lambda}{\hookrightarrow} \sigma_1(r_1) \trianglerighteq_\mu t_1 \xrightarrow{>\Lambda}{}^{*}_{\mathcal{R},\mu} \sigma_2(l_2) \overset{\Lambda}{\hookrightarrow} \sigma_2(r_2) \trianglerighteq_\mu t_2 \xrightarrow{>\Lambda}{}^{*}_{\mathcal{R},\mu} \cdots$ where, for all $i \geq 1$, $l_i \to r_i \in R$ are rewrite rules, σ_i are substitutions, and terms $t_i \in \mathcal{M}_{\infty,\mu}$ are minimal non-μ-terminating terms such that either*

1. *$t_i = \sigma_i(s_i)$ for some s_i such that $r_i \trianglerighteq_\mu s_i$, or*
2. *$\sigma_i(x_i) = C_i[t_i]$ for some $x_i \in \mathcal{V}ar^{\mu}(r_i) \setminus \mathcal{V}ar^{\mu}(l_i)$ and $C_i[t_i] = \theta_i(C'_i)[\theta_i(t'_i)]$ for some $t'_i \in \mathcal{NHT}(\mathcal{R},\mu)$, some hiding context $C'_i[\Box]$ and substitution θ_i.*

3 Context-Sensitive Dependency Pairs

The following definitions naturally follow from the facts which have been established in Theorem 1.

Definition 1 (CSDPs [4]). *Let $(\mathcal{R},\mu)$ be a CS-TRS. We define $\mathsf{DP}(\mathcal{R},\mu) = \mathsf{DP}_{\mathcal{F}}(\mathcal{R},\mu) \cup \mathsf{DP}_{\mathcal{X}}(\mathcal{R},\mu)$ to be set of* context-sensitive dependency pairs *where:*

$$\begin{array}{rcl} \mathsf{DP}_{\mathcal{F}}(\mathcal{R},\mu) & = & \{\ell^\sharp \to s^\sharp \mid \ell \to r \in R, r \trianglerighteq_\mu s, \mathsf{root}(s) \in \mathcal{D}, \ell \ntriangleright_\mu s, \textsc{Narr}^{\mu}(\textsc{Ren}^{\mu}(s))\} \\ \mathsf{DP}_{\mathcal{X}}(\mathcal{R},\mu) & = & \{\ell^\sharp \to x \mid \ell \to r \in R, x \in \mathcal{V}ar^{\mu}(r) \setminus \mathcal{V}ar^{\mu}(l)\} \end{array}$$

We extend $\mu \in M_{\mathcal{F}}$ into $\mu^\sharp \in M_{\mathcal{F}\cup\mathcal{D}^\sharp}$ by $\mu^\sharp(f) = \mu(f)$ if $f \in \mathcal{F}$ and $\mu^\sharp(f^\sharp) = \mu(f)$ if $f \in \mathcal{D}$.

In contrast to [1], we store the information about hidden terms and hiding contexts as a new TRS.

Definition 2 (Unhiding TRS). *For a CS-TRS $(\mathcal{R},\mu)$, let $\mathsf{unh}(\mathcal{R},\mu)$ be the TRS formed by the rules:*

- *$f(x_1,\ldots,x_i,\ldots,x_n) \to x_i$ for every function symbol f of any arity n and every $1 \leq i \leq n$ where f hides position i, and*
- *$t \to t^\sharp$ for every $t \in \mathcal{NHT}(\mathcal{R},\mu)$.*

Example 3. *The unhiding TRS for the CS-TRS in Example Example 1 is:*

$\mathsf{minus}(\mathsf{p}(x),\mathsf{p}(y)) \to \mathsf{M}(\mathsf{p}(x),\mathsf{p}(y))$	(16)	$\mathsf{p}(x) \to x$	(18)
$\mathsf{p}(x) \to \mathsf{P}(x)$	(17)	$\mathsf{minus}(x,y) \to x$	(19)
		$\mathsf{minus}(x,y) \to y$	(20)

Now we use Definitions 1 and 2 to provide a suitable notion of *chain* which is able to capture minimal infinite μ-rewrite sequences according to the description in Theorem 1.

Definition 3 ($\mathcal{S}$-chain of pairs - Minimal $\mathcal{S}$-chain). *Let $\mathcal{R} = (\mathcal{F}, R)$ and $\mathcal{P} = (\mathcal{G}, P)$ be TRSs and $\mu \in M_{\mathcal{F}\cup\mathcal{G}}$. Let $(\mathcal{S},\mu) = (\mathcal{S}_{\triangleright_\mu} \cup \mathcal{S}_\sharp, \mu)$, where $\mathcal{S}_{\triangleright_\mu}$ are rules of the form $f(x_1,\ldots,x_i,\ldots,x_n) \to x_i \in \mathcal{S}$ for some $f \in \mathcal{F}$ and $i \in \mu(f)$; and $\mathcal{S}_\sharp = \mathcal{S} \setminus \mathcal{S}_{\triangleright_\mu}$. A $(\mathcal{P},\mathcal{R},\mathcal{S},\mu)$-chain is a finite or infinite sequence of pairs $u_i \to v_i \in \mathcal{P}$, together with a substitution σ satisfying that, for all $i \geq 1$,*

1. *if $v_i \notin \mathcal{V}ar(u_i) \setminus \mathcal{V}ar^{\mu}(u_i)$, then $\sigma(v_i) = s_i \hookrightarrow^{*}_{\mathcal{R},\mu} \sigma(u_{i+1})$, and*
2. *if $v_i \in \mathcal{V}ar(u_i) \setminus \mathcal{V}ar^{\mu}(u_i)$, then $\sigma(v_i) \overset{\Lambda}{\hookrightarrow}{}^{*}_{\mathcal{S}_{\triangleright_\mu},\mu} \circ \overset{\Lambda}{\hookrightarrow}_{\mathcal{S}_\sharp,\mu} s_i \hookrightarrow^{*}_{\mathcal{R},\mu} \sigma(u_{i+1})$.*

A $(\mathcal{P},\mathcal{R},\mathcal{S},\mu)$-chain is called minimal *if for all $i \geq 1$ s_i is $(\mathcal{R},\mu)$-terminating.*

Notice that if $\mathcal{S}_{\triangleright_\mu} = \{f(x_1,\ldots,x_n) \to x_i \mid f \in \mathcal{D}, i \in \mu(f)\}$, then we have the notion of chain in [4] and if $\mathcal{S}_\sharp = \{f(x_1,\ldots,x_n) \to f^\sharp(x_1,\ldots,x_n) \mid f \in \mathcal{D}\}$, then we have the original notion of chain from [2].

Theorem 2 (Soundness and Completeness of CSDPs). *Let $\mathcal{R}$ be a TRS and $\mu \in M_{\mathcal{R}}$. A CS-TRS $(\mathcal{R},\mu)$ is terminating iff there is no infinite $(\mathsf{DP}(\mathcal{R},\mu), \mathcal{R}, \mathsf{unh}(\mathcal{R},\mu), \mu)$-chain.*

4 Dependency Pair Framework

We adapt the definition of *CS-termination problem* in an easy way (see also [1, 4]).

Definition 4 (CS-termination problem and processor)**.** *A* CS-termination problem τ *is a tuple* $\tau = (\mathscr{P},\mathscr{R},\mathscr{S},\mu)$*, where* $\mathscr{R} = (\mathscr{F},R)$*,* $\mathscr{P} = (\mathscr{G},P)$ *and* $\mathscr{S} = (\mathscr{F}\cup\mathscr{G},S)$ *are TRSs, and* $\mu \in M_{\mathscr{F}\cup\mathscr{G}}$*. A* CS-termination problem $(\mathscr{P},\mathscr{R},\mathscr{S},\mu)$ *is* finite *if there is no infinite* $(\mathscr{P},\mathscr{R},\mathscr{S},\mu)$*-chain.*

A CS-processor Proc *is a mapping from CS-termination problem into sets of CS-termination problems. A CS-processor* Proc *is* sound *if for all CS-termination problem* τ*,* τ *is finite whenever* τ' *is finite for all* $\tau' \in \mathsf{Proc}(\tau)$*. A CS-processor* Proc *is* complete *if for all CS-termination problem* τ*, whenever* τ *is finite,* τ' *is finite for all* $\tau' \in \mathsf{Proc}(\tau)$*.*

Our initial CS-termination problem is $(\mathsf{DP}(\mathscr{R},\mu),\mathscr{R},\mathsf{unh}(\mathscr{R},\mu),\mu)$ and we repeatedly apply CS-processors until the CS-processors return empty sets, and hence, the μ-termination of $\mathscr{R}$ is proved.

Another advantage of the new definition is that we can easily use all existing processors (see [1, 4]). The following processor integrates the transformation of [1] into the new framework.

Theorem 3 (Collapsing pairs transformation processor)**.** *Let* $\tau = (\mathscr{P},\mathscr{R},\mathscr{S},\mu)$ *be a CS-termination problem. Let* P_{U} *consist of the following rules:*

- $u \to \mathsf{U}(x)$ *for every* $u \to x \in \mathscr{P}_{\mathscr{X}}$*,*
- $\mathsf{U}(f(x_1,\ldots,x_i,\ldots,x_n)) \to \mathsf{U}(x_i)$ *for every* $f(x_1,\ldots,x_i,\ldots,x_n) \to x_i \in \mathscr{S}_{\rhd_\mu}$*, and*
- $\mathsf{U}(s) \to t$ *for every* $s \to t \in \mathscr{S}_{\sharp}$*.*

where U *is a new fresh symbol. Let* $\mathscr{P}' = (\mathscr{G}\cup\{\mathsf{U}\},P')$ *where* $P' = (P \setminus P_{\mathscr{X}}) \cup P_{\mathsf{U}}$*, and* μ' *extends* μ *by* $\mu'(\mathsf{U}) = \varnothing$*. The processor* Proc_{eColl} *given by* $\mathsf{Proc}_{eColl}(\tau) = \{(\mathscr{P}',\mathscr{R},\varnothing,\mu')\}$ *is sound and complete.*

Now, we can then apply all processor from [1] and [4] which did not consider any $\mathscr{S}$ component in termination problems.

References

[1] B. Alarcón, F. Emmes, C. Fuhs, J. Giesl, R. Gutiérrez, S. Lucas, P. Schneider-Kamp, and R. Thiemann. Improving Context-Sensitive Dependency Pairs. In I. Cervesato, H. Veith, and A. Voronkov, editors, *Proc. of 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'08*, volume 5330 of *Lecture Notes in Computer Science*, pages 636–651. Springer-Verlag, November 2008.

[2] B. Alarcón, R. Gutiérrez, and S. Lucas. Context-Sensitive Dependency Pairs. In S. Arun-Kumar and N. Garg, editors, *Proc. of 26th Conference on Foundations of Software Technology and Theoretical Computer Science, FST&TCS'06*, volume 4337 of *Lecture Notes in Computer Science*, pages 297–308. Springer-Verlag, 2006.

[3] B. Alarcón, R. Gutiérrez, and S. Lucas. Improving the Context-Sensitive Dependency Graph. *Electronic Notes in Theoretical Computer Science*, 188:91–103, 2007.

[4] B. Alarcón, R. Gutiérrez, and S. Lucas. Context-sensitive dependency pairs. Technical report, Universidad Politécnica de Valencia, July 2008. Available as Technical Report DSIC-II/10/08.

[5] R. Gutiérrez. Context-Sensitive Dependency Pairs Framework. Master's thesis, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Valencia, Spain, December 2008.

[6] S. Lucas. Context-Sensitive Computations in Functional and Functional Logic Programs. *Journal of Functional and Logic Programming*, 1998(1):1–61, 1998.

The Subterm Criterion in Complexity Analysis[*]

Nao Hirokawa
School of Information Science
Japan Advanced Institute of Science and Technology, Japan
hirokawa@jaist.ac.jp

Georg Moser
Institute of Computer Science
University of Innsbruck, Austria
georg.moser@uibk.ac.at

Abstract

The subterm criterion, an extension of the dependency pair method, invented by the first author and Middeldorp, turned out to be a simple, but yet very powerful tool in the termination analysis of rewrite systems. In this note we study this criterion from the viewpoint of complexity analysis.

1 Introduction

Given a finite term rewrite system R (TRS for short) and a *terminating* term t, it is a natural question to study the *derivation length* of t with respect to the rewrite relation $\rightarrow_{\mathsf{R}}$:

$$\mathsf{dl}(t, \rightarrow_{\mathsf{R}}) := \max\{n \mid t \rightarrow_{\mathsf{R}}^{n} u \text{ for some } u\} \; .$$

Roughly such an analysis amounts to a worst-case complexity analysis of the functions computed by the TRS R. See [7, 2] for early results in this area. In the following we assume familiarity with term rewriting, see [1, 10]. In this note we study the *subterm criterion* (see [4]) in the context of complexity analysis. To prepare our analysis we consider the following variant of the subterm criterion. Note that in the context of termination this form is weaker than its original formulation, compare [4]. Recall that a *simple projection* π is a special argument filtering that projects arguments of dependency pair symbols but leaves other symbols unchanged.

Lemma 1.1. *Let* R *be a TRS and* $\mathsf{P} = \mathsf{DP}(\mathsf{R})$ *the set of dependency pairs. Assume there exists a simple projection* π *such that* $\pi(\mathsf{P}) \subseteq \rhd$. *Then* R *is terminating.*

In proof of this lemma one argues indirectly assuming the existence of a minimal infinite derivation over R such that the relation $\rightarrow_{\mathsf{R}}$ is well-founded on the set $\{u_i \mid i \in \mathbb{N}\}$ of all direct subterms in the studied derivation. Moreover one utilises the fact that the proper superterm relation $\rhd$ commutes over $\rightarrow_{\mathsf{R}}$, i.e., $\rhd \cdot \rightarrow_{\mathsf{R}} \subseteq \rightarrow_{\mathsf{R}} \cdot \rhd$. These two observations suffice to derive the sought contradiction.

2 The Subterm Criterion in Complexity Analysis

To simplify the analysis we momentarily focus on string rewrite systems (SRSs for short), i.e., TRSs whose signature consists of unary function symbols only. We recall the notion of *weak dependency pairs* in the context of SRSs. Let $f \in \mathsf{F}$ denote a function symbol in the signature F; we write $f^{\sharp}$ to denote the corresponding dependency pair symbol. The marked term $t^{\sharp}$ is defined as usual. Suppose $l \rightarrow r \in \mathsf{R}$ and $r = C[u]$ such that C is a maximal context, free of variables or defined symbols. Then the rewrite rule $l^{\sharp} \rightarrow u^{\sharp}$ is called a *weak dependency pair* of R. The set of all weak dependency pairs is denoted by $\mathsf{WDP}(\mathsf{R})$.

We need some additional definitions. We say that a term t is *basic* if all proper subterms of t are constructor terms. The set of all (marked) basic terms is denoted as $\mathsf{T_b}$ ($\mathsf{T_b^{\sharp}}$). The set of usable rules of the set weak dependency pairs P is denoted as $\mathsf{U}(\mathsf{P})$.

[*]This reseach is partly supported by Grant-in-Aid for Young Scientists 20800022 of the Ministry of Education, Culture, Sports, Science and Technology of Japan and by FWF (Austrian Science Fund) project P20133.

Theorem 2.1. *Let* R *be an SRS, let* $t \in \mathsf{T_b}$ *be terminating and* $\mathsf{P} = \mathsf{WDP}(\mathsf{R})$*. Assume there exists a simple projection* π *such that* $\pi(\mathsf{P}) \subseteq \rhd$*. Then* $\mathsf{dl}(t, \to_{\mathsf{R}}) \leqslant K \cdot \mathsf{dl}(t^\sharp, \to_{\mathsf{U(P)}}) + |t|$*, for some constant K. (Here* $|t|$ *denotes the size of t, defined as usual.)*

In proof, one first employs that $\mathsf{dl}(t, \to_{\mathsf{R}}) \leqslant \mathsf{dl}(t^\sharp, \to_{\mathsf{U(P)} \cup \mathsf{P}})$, (see [5, Theorem 17]) to translate any derivation over R into a derivation over $\mathsf{U(P)} \cup \mathsf{P}$. Then one essentially follows the pattern of the proof of Lemma 1.1 to construct a sequence of terms:

$$u_0 \to_{\mathsf{U(P)}} u_1 \to_{\mathsf{U(P)}} u_2 \to_{\mathsf{U(P)}} \cdots \to_{\mathsf{U(P)}} u_k \rhd u_{k+1} \rhd u_{n+2} \rhd \cdots u_{k+l} \ ,$$

such that $\mathsf{dl}(t^\sharp, \to_{\mathsf{U(P)} \cup \mathsf{P}}) = k + l$. Here the fact that all function symbols are at-most unary is used. Thus $k \leqslant \mathsf{dl}(u_0, \to_{\mathsf{U(P)}})$. It remains to bound the number of $\rhd$-steps. To this avail, we observe the following lemma, whose proof follows by a straightforward inductive argument.

Lemma 2.2. *Let* R *denote a TRS and let* $\mathsf{dp}(\mathsf{R}) := \max\{\mathsf{dp}(r) \mid l \to r \in \mathsf{R}\} \leqslant K$ *for some number K. If* $s \to_{\mathsf{R}} t$*, then* $\mathsf{dp}(s) + K \geqslant \mathsf{dp}(t)$*. (Here* $\mathsf{dp}(t)$ *denotes the depth of t defined as usual.)*

It remains to give an explicit criterion to easily verify the derivation length. Recall that the *runtime complexity function* (with respect to $\to_{\mathsf{R}}$) is defined as

$$\mathsf{rc_R}(n) := \max\{\mathsf{dl}(t, \to_{\mathsf{R}}) \mid t \in \mathsf{T_b} \text{ and } |t| \leqslant n\} \ .$$

We utilise matrix interpretations, see [8, 3]. We fix a dimension $d \in \mathbb{N}$ and use the set $\mathbb{N}^d$ as the carrier of an algebra A, together with the following extension of the natural order $>$ on $\mathbb{N}$: $(x_1, x_2, \ldots, x_d) > (y_1, y_2, \ldots, y_d) :\Longleftrightarrow x_1 > y_1 \wedge x_2 \geqslant y_2 \wedge \ldots \wedge x_d \geqslant y_d$. For each n-ary function symbol f, we choose as an interpretation a linear function of the following shape: $f_{\mathsf{A}} : (\mathbb{N}^d)^n \to \mathbb{N}^d : (\vec{v}_1, \ldots, \vec{v}_n) \mapsto F_1\vec{v}_1 + \ldots + F_n\vec{v}_n + \vec{f}$, where $\vec{v}_1, \ldots, \vec{v}_n$ are (column) vectors of variables, $F_1, \ldots, F_n$ are matrices (each of size $d \times d$), and $\vec{f}$ is a vector over $\mathbb{N}$. Moreover, for any i $(1 \leqslant i \leqslant n)$ the top left entry $(F_i)_{1,1}$ is positive. It is easy to see that the algebra forms a well-founded monotone algebra.

An *upper triangular matrix* is a matrix M in $\mathbb{N}^{d \times d}$ such that we have $M_{j,k} = 0$ for all $1 \leqslant k < j \leqslant d$, and $M_{j,j} \leqslant 1$ for all $1 \leqslant j \leqslant d$. We say that a matrix interpretation A is a *triangular matrix interpretation* (*TMI* for short) if A is a matrix interpretation (over $\mathbb{N}$) and all matrices employed are of upper triangular form. Let A be a TMI over domain $\mathbb{N}^d$, let α_0 denotes the assignment mapping any variable to $\vec{0}$, i.e., $\alpha_0(x) = \vec{0}$ for all $x \in \mathsf{V}$, and let t be a term. In the following we write $[t]$, $[t]_j$ as an abbreviation for $[\alpha_0]_{\mathsf{A}}(t)$, or $([\alpha_0]_{\mathsf{A}}(t))_j$, respectively. Let t be a term. Then it is easy to see that there exists a polynomial p such that $[t]_j \leqslant p(|t|)$ holds for all $j = 1, \ldots, d$, compare [9]. This allows the application of TMIs in our context: it suffices to observe that $\mathsf{dl}(u_0, \to_{\mathsf{U(P)}})$ is bounded by the evaluation (with respect to A) of the term u_0.

Corollary 2.3. *Let* R *be a SRS, let t be terminating and* $\mathsf{P} = \mathsf{WDP}(\mathsf{R})$*. Assume there exists a simple projection* π *such that* $\pi(\mathsf{P}) \subseteq \rhd$ *and assume there exists a TMI* A *such that* A *is compatible with* $\mathsf{U(P)}$*. The the runtime complexity with respect to* R *is polynomially bounded.*

3 Extensions beyond String Rewriting

It is tempting to think that Theorem 2.1 ought to be extensible to an arbitrary signature. This is unfortunately no the case. We generalise weak dependency pairs to arbitrary TRSs.

We define the function COM as a mapping from tuples of terms to terms as follows: $\text{COM}(t_1, \ldots, t_n)$ is t_1 if $n = 1$, and $c(t_1, \ldots, t_n)$ otherwise. Here c is a fresh n-ary function symbol called *compound symbol*.

If $l \to r \in \mathsf{R}$ and $r = C[u_1, \dots, u_n]$ then the rewrite rule $l^\sharp \to \text{COM}(u_1^\sharp, \dots, u_n^\sharp)$ is called a weak dependency pair of R. Here the context C fulfils the same conditions as above. Consider the following TRS R

$$1\colon\ \mathsf{d}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{d}(x))) \qquad 2\colon\ \mathsf{f}(\mathsf{s}(x), y) \to \mathsf{f}(x, \mathsf{d}(y))$$

Note that the runtime complexity of R is exponential. On the other hand consider the set of weak dependency pairs $\mathsf{P} = \mathsf{WDP}(\mathsf{R})$.

$$4\colon\ \mathsf{d}^\sharp(\mathsf{s}(x)) \to \mathsf{d}^\sharp(x) \qquad 5\colon\ \mathsf{f}^\sharp(\mathsf{s}(x), y) \to \mathsf{f}^\sharp(x, \mathsf{d}(y))$$

Using the simple projection induced by $\pi(\mathsf{f}^\sharp) = 1$, $\pi(\mathsf{d}^\sharp) = 1$ we observe $\pi(\mathsf{P}) \subseteq \rhd$ and the following polynomial interpretation A is compatible with $\mathsf{U}(\mathsf{P})$: $\mathsf{s}_\mathsf{A}(x) := x + 1$, $\mathsf{d}_\mathsf{A}(x) = 3x$. If the theorem could be extended to TRSs we would have to conclude that the runtime complexity function rc_R is linearly bounded.

However, we can extend the theorem to TRSs where the arity of defined function symbols is at most unary. More formally a TRS R is called *monadic* if R is based on a signature F so that all defined symbols in F have arity 0 or 1. In order to do so, we need some additional notation. The set $\mathsf{T}_\mathsf{c}^\sharp$ is inductively defined as follows (i) $\mathsf{T}^\sharp \cup \mathsf{T} \subseteq \mathsf{T}_\mathsf{c}^\sharp$, where $\mathsf{T}^\sharp = \{t^\sharp \mid t \in \mathsf{T}\}$ and (ii) $c(t_1, \dots, t_n) \in \mathsf{T}_\mathsf{c}^\sharp$, whenever $t_1, \dots, t_n \in \mathsf{T}_\mathsf{c}^\sharp$ and c a compound symbol. Moreover, we define a slight extension $\rhd'$ of the subterm relation over $\mathsf{T}_\mathsf{c}^\sharp$: for $s, t \in \mathsf{T}_\mathsf{c}^\sharp$ let $s \rhd' t$ if either $s, t \in \mathsf{T}^\sharp \cup \mathsf{T}$ and $s \rhd' t$ or $s \in \mathsf{T}^\sharp \cup \mathsf{T}$ and t a nullary compound symbol or dependency pair symbol c. In order to employ the extension, is suffices to check that the $\rhd'$ commutes over $\to_\mathsf{R}$. This follows easily as c is in normal form with respect to $\to_\mathsf{R}$.

We obtain the following result. Let R be a monadic TRS based on $\mathsf{F} = \mathsf{D} \cup \mathsf{C}$, where D (C) denotes the defined (constructor) symbols in the signature. Due to the constraint on the signature, we can fix the simple projection $\pi(f^\sharp) = 1$ for $f \in \mathsf{D}$ if f is unary, and $\pi(c^\sharp) = [\,]$, if c is nullary.

Corollary 3.1. *Let R be a monadic TRS and let $\mathsf{P} = \mathsf{WDP}(\mathsf{R})$. Assume $\pi(\mathsf{P}) \subseteq \rhd'$ and assume there exists a TMI A such that A is compatible with $\mathsf{U}(\mathsf{P})$. Then the runtime complexity with respect to R is polynomially bounded.*

In proof it suffices to observe that the simple projection π can only delete essential rewrite steps if the arity of the *defined* symbols is strictly greater than 1. Note that the above corollary differs from earlier results on the incorporation of the dependency pair method in complexity analysis. In particular in comparison to Corollary 35 in [5] the requirement about *strongly linear interpretations* could be dropped. However here we have to restrict to *monadic* TRSs. Note that for dimension 1 (i.e., if the TMI A amount to a restricted linear polynomial) Corollary 3.1 is subsumed by Corollary 18 in [5].

4 Weak Dependency Graphs

A very well-studied refinement of the dependency pair method are dependency graphs. And it is no coincidence that the original formulation of the subterm criterion is formulated in the context of dependency graphs. For complexity analysis we introduced in [6] the notion of *weak dependency graphs*. Let R be a TRS over a signature F and let P be the set of weak dependency pairs. The nodes of the *weak dependency graph* $\mathsf{WDG}(\mathsf{R})$ are the elements of P and there is an arrow from $s \to t$ to $u \to v$ if and only if there exist a context C and substitutions $\sigma, \tau\colon \mathsf{V} \to \mathsf{T}(\mathsf{F}, \mathsf{V})$ such that $t\sigma \to_\mathsf{R}^* C[u\tau]$.

A graph is called *strongly connected* if any node is connected with every other node by a path. A *strongly connected component* (*SCC* for short) is a maximal strongly connected subgraph. Let $\mathsf{G} = \mathsf{WDG}(\mathsf{R})$, we write $\mathsf{G}/_\equiv$ for the *congruence graph*, where $\equiv$ is the equivalence relation induced by SCCs. Suppose $t \in \mathsf{T}_\mathsf{b}^\sharp$ is terminating and define

$$\mathsf{L}(t) := \max\{\mathsf{dl}(t, \to_{\mathsf{P}_m/\mathsf{S}}) \mid (\mathsf{P}_1, \dots, \mathsf{P}_m) \text{ is a path in } \mathsf{G}/_\equiv \text{ such that } \mathsf{P}_1 \in \mathsf{Src}\}\ ,$$

where Src denotes the set of source nodes in $\mathsf{G}/_{\equiv}$ and $\mathsf{S} = \mathsf{P}_1 \cup \cdots \cup \mathsf{P}_{m-1} \cup \mathsf{U}(\mathsf{P}_1 \cup \cdots \cup \mathsf{P}_m)$. Then there exists a polynomial p (depending only on R) such that $\mathsf{dl}(t, \rightarrow_{\mathsf{P}/\mathsf{U}(\mathsf{P})}) \leqslant p(\mathsf{L}(t))$, cf. [6].

Combining the subterm criterion and weak dependency graphs we obtain the following result.

Corollary 4.1. *Let* R *be a monadic TRS and let* $\mathsf{P} = \mathsf{WDP}(\mathsf{R})$. *For all maximal paths in* $\mathsf{G}/_{\equiv}$

$$(\mathsf{P}_1, \ldots, \mathsf{P}_m)\,, \tag{1}$$

such that $\mathsf{P}_1 \in \mathsf{Src}$ *assume the following: (i)* $\pi(\mathsf{P}_m) \subseteq \rhd'$ *and (ii) there exists a TMI* A *such that* A *is compatible with* $\mathsf{P}_1 \cup \cdots \cup \mathsf{P}_{m-1} \cup \mathsf{U}(\mathsf{P}_1) \cup \cdots \cup \mathsf{U}(\mathsf{P}_m)$. *Then the runtime complexity with respect to* R *is polynomially bounded.*

In proof, observe that the corollary is slightly stronger than the immediate combination of the subterm criterion and weak dependency graphs. Namely for the latter we would have to assume that for each path (1) the usable rules $\mathsf{U}(\mathsf{P}_1 \cup \cdots \cup \mathsf{P}_m)$ are compatible to a strongly linear interpretation, compare [6, Corollary 24]. In the context of the subterm criterion and the restriction to monadic TRSs this is no longer necessary.

5 Conclusion

In this note we study the subterm criterion from the viewpoint of complexity analysis. Our analysis reveals that in the context of polynomial runtime complexity analysis the criterion is only applicable to a restrictive subclass of TRSs, dubbed monadic above.

References

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

[2] E.-A. Cichon and P. Lescanne. Polynomial interpretations and the complexity of algorithms. In *Proc. 11th CADE*, pages 139–147. Springer Verlag, 1992.

[3] J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *JAR*, 40(3):195–220, 2008.

[4] N. Hirokawa and A. Middeldorp. Tyrolean termination tool: Techniques and features. *IC*, 205:474–511, 2007.

[5] N. Hirokawa and G. Moser. Automated complexity analysis based on the dependency pair method. In *Proc. 4th IJCAR*, volume 5195 of *LNCS*, pages 364–379, 2008. Revised version available at `http://cl-informatik.uibk.ac.at/~georg/list.publications.html`.

[6] N. Hirokawa and G. Moser. Complexity, graphs, and the dependency pair method. In *Proc. 15th LPAR*, volume 5330 of *LNCS*, pages 667–681, 2008. Revised version available at `http://cl-informatik.uibk.ac.at/~georg/list.publications.html`.

[7] D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations. In *Proc. 3rd RTA*, volume 355 of *LNCS*, pages 167–177, 1989.

[8] D. Hofbauer and J. Waldmann. Termination of string rewriting with matrix interpretations. In *Proc. 17th RTA*, volume 4098 of *LNCS*, pages 328–342, 2006.

[9] G. Moser, A. Schnabl, and J. Waldmann. Complexity analysis of term rewriting based on matrix and context dependent interpretations. In *Proc. 28th FSTTCS*, pages 304–315. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2008. Creative-Commons-NC-ND licensed.

[10] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

System Description: KnockedForLoops

Dieter Hofbauer
BA Nordhessen
D-34537 Bad Wildungen, Germany
`d.hofbauer@ba-nordhessen.de`

Abstract

An implementation of loop detection for string rewriting based on forward closures is presented.

1 Introduction

Apart from cycles, loops are the most elementary certificates for nontermination of rewriting systems. In this abstract we adress the automatic search for loops in string rewriting, where loops are derivations of the form $x \to_R^+ y$ with x being a factor of y.

2 Algorithmic approach

Due to Geser and Zantema [1], the existence of a loop is equivalent to the existence of a looping forward closure. Our tool (current working title KnockedForLoops) implements a brute-force breadth-first enumeration of forward closures. It is implemented in Java and uses multithreading. Only few straightforward combinatorial optimizations are employed. Essential, however, turned out to be bounding the size of initial strings in forward closures during the enumeration. This allows for detecting longer loops within given time and space resources.

As also shown by Geser and Zantema [1], if there is a loop of length n, then there also is a looping forward closure of length at most n, thus short loops are not lost when restricting to forward closures. Concerning the symmetry between left and right forward closures (expanding prefixes or suffixes respectively), this also shows that the minimal lengths of looping left and right forward closures coincide. Nevertheless, this symmetry is no longer preserved when considering the size of initial strings in forward closures. As an example, consider the one-rule string rewriting system $\{cbaba \to abababcbc\}$ (`SRS/Zantema/z042.srs` in the termination problems data base). Here, the minimal length of initial strings of looping right forward closures is 13, whereas a looping left forward closure with an initial string of size 11 exists. As a consequence of this simple observation, we search for left and right forward closures in parallel.

3 Experimental results

Our prototype implementation seems to be able to detect all looping string rewriting systems in the termination problems data base ever found automatically during runs of the annual termination competition. We expect experimental results performed at the competition platform to be available at the workshop.

References

[1] Alfons Geser, Hans Zantema Non-looping string rewriting. *Informatique Théorique et Applications* 33(3): 279-302, 1999.

Beyond Dependency Graphs[*]

Martin Korp and Aart Middeldorp
Institute of Computer Science
University of Innsbruck, Austria

Abstract

The dependency pair framework is a powerful technique for proving termination of rewrite systems. One of the most frequently used methods within the dependency pair framework is the dependency graph processor. In this note we improve this processor by incorporating right-hand sides of forward closures. In combination with tree automata completion we obtain an efficient processor which can be used instead of the dependency graph approximations that are in common use in termination provers.

1 Introduction

Proving termination of term rewrite systems is a very active research area. Several tools exist that perform this task automatically. The most powerful ones are based on the dependency pair framework. This framework combines a great variety of termination techniques in a modular way by means of dependency pair processors. In this note we are concerned with the dependency graph processor. It is one of the most important processors as it enables the decomposition of termination problems into smaller subproblems. The processor requires the computation of an over-approximation of the dependency graph. In the literature several such approximations are proposed [1, 8, 12, 13]. In this note we return to tree automata techniques. We show that tree automata *completion* is much more effective for approximating dependency graphs than the method based on approximating the underlying rewrite system to ensure regularity preservation proposed in [12]. We further show that by incorporating *right-hand sides of forward closures* [4], a technique that recently became popular in connection with the match-bound technique [6, 11], we can eliminate arcs from the (real) dependency graph.

The remainder of the note is organized as follows. In Section 2 we recall some basic facts about dependency graphs and processors. In Section 3 we employ tree automata completion to approximate dependency graphs and in Section 4 we incorporate right-hand sides of forward closures. Experimental data is presented in Section 5.

2 Preliminaries

Familiarity with term rewriting [2] and tree automata [3] is assumed. Knowledge of the dependency pair framework [7, 14] and the match-bound technique [6, 11] will be helpful. Below we recall important definitions concerning the former needed in the remainder of the note. Throughout this note we assume that TRSs are finite.

Let $\mathcal{R}$ be a term rewrite system (TRS for short). The set of *dependency pairs* of $\mathcal{R}$ is denoted by $\mathsf{DP}(\mathcal{R})$. A *DP problem* is a triple $(\mathcal{P},\mathcal{R},\mathcal{G})$ where $\mathcal{P}$ and $\mathcal{R}$ are two TRSs and $\mathcal{G} \subseteq \mathcal{P} \times \mathcal{P}$ is a directed graph. A DP problem $(\mathcal{P},\mathcal{R},\mathcal{G})$ is called *finite* if there are no infinite rewrite sequences of the form $s_1 \xrightarrow{\epsilon}_{\alpha_1} t_1 \to^*_{\mathcal{R}} s_2 \xrightarrow{\epsilon}_{\alpha_2} t_2 \to^*_{\mathcal{R}} \cdots$ such that all terms t_1, t_2, $\ldots$ are terminating with respect to $\mathcal{R}$ and $(\alpha_i, \alpha_{i+1}) \in \mathcal{G}$ for all $i \geqslant 1$. Such an infinite sequence is said to be *minimal*. The main result underlying the dependency pair approach states that a TRS $\mathcal{R}$ is terminating if and only if the DP problem $(\mathsf{DP}(\mathcal{R}), \mathcal{R}, \mathsf{DP}(\mathcal{R}) \times \mathsf{DP}(\mathcal{R}))$ is finite. The latter is shown by applying functions that take a DP problem as input and return a set of DP problems as output, the so-called *DP processors*. These processors must have the property that a DP problem is finite whenever all DP problems returned by the processor are

[*]A full version of this note will appear in the Proc. of the 22nd International Conference on Automated Deduction, 2009.

finite, which is known as *soundness*. To use DP processors for establishing non-termination, they must additionally be *complete* which means that if one of the DP problems returned by the processor is not finite then the original DP problem is not finite.

Numerous DP processors have been developed. In this note we are concerned with the dependency graph processor. It determines which dependency pairs can follow each other in infinite rewrite sequences.

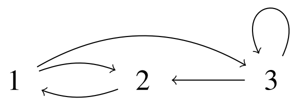
Definition 1. The *dependency graph processor* maps a DP problem $(\mathcal{P},\mathcal{R},\mathcal{G})$ to the set $\{(\mathcal{P},\mathcal{R},\mathcal{G}\cap \mathsf{DG}(\mathcal{P},\mathcal{R}))\}$. Here $\mathsf{DG}(\mathcal{P},\mathcal{R})$ is the *dependency graph* of $\mathcal{P}$ and $\mathcal{R}$, which has the rules in $\mathcal{P}$ as nodes and there is an arc from $s \to t$ to $u \to v$ if and only if there exist substitutions σ and τ such that $t\sigma \to_{\mathcal{R}}^* u\tau$.

It is well-known [1, 9, 14] that the dependency graph processor is sound and complete.

Example 2. Consider the DP problem $(\mathcal{P},\mathcal{R},\mathcal{G})$ with $\mathcal{R}$ consisting of the rewrite rules $\mathsf{f}(\mathsf{g}(x),y) \to \mathsf{g}(\mathsf{h}(x,y))$ and $\mathsf{h}(\mathsf{g}(x),y) \to \mathsf{f}(\mathsf{g}(\mathsf{a}),\mathsf{h}(x,y))$, $\mathcal{P} = \mathsf{DP}(\mathcal{R})$ consisting of

$$1\colon \mathsf{F}(\mathsf{g}(x),y) \to \mathsf{H}(x,y) \qquad 2\colon \mathsf{H}(\mathsf{g}(x),y) \to \mathsf{F}(\mathsf{g}(\mathsf{a}),\mathsf{h}(x,y)) \qquad 3\colon \mathsf{H}(\mathsf{g}(x),y) \to \mathsf{H}(x,y)$$

and $\mathcal{G} = \mathcal{P} \times \mathcal{P}$. Because $\mathsf{H}(\mathsf{g}(x),y)$ is an instance of $\mathsf{H}(x,y)$ and $\mathsf{F}(\mathsf{g}(\mathsf{a}),\mathsf{h}(x,y))$ is an instance of $\mathsf{F}(\mathsf{g}(x),y)$, $\mathsf{DG}(\mathcal{P},\mathcal{R})$ has five arcs:



$$\begin{tikzcd}
1 \arrow[r, bend left] \arrow[rr, bend left=40] & 2 \arrow[l, bend left] & 3 \arrow[l] \arrow[loop above]
\end{tikzcd}$$

The dependency graph processor returns the new DP problem $(\mathcal{P},\mathcal{R},\mathsf{DG}(\mathcal{P},\mathcal{R}))$.

3 Tree Automata Completion

We start by recalling some basic facts and notions. Let $\mathcal{R}$ be a TRS over $\mathcal{F}$. The set $\{t \in \mathcal{T}(\mathcal{F}) \mid s \to_{\mathcal{R}}^* t \text{ for some } s \in L\}$ of descendants of a set $L \subseteq \mathcal{T}(\mathcal{F})$ of ground terms is denoted by $\to_{\mathcal{R}}^*(L)$. We say that a tree automaton $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ is *compatible* with $\mathcal{R}$ and L if $L \subseteq \mathcal{L}(\mathcal{A})$ and for each rewrite rule $l \to r \in \mathcal{R}$ and state substitution $\sigma\colon \mathcal{V}\mathsf{ar}(l) \to Q$ such that $l\sigma \to_{\Delta}^* q$ it holds that $r\sigma \to_{\Delta}^* q$. For left-linear $\mathcal{R}$ it is known that $\to_{\mathcal{R}}^*(L) \subseteq \mathcal{L}(\mathcal{A})$ whenever $\mathcal{A}$ is compatible with $\mathcal{R}$ and L [5].

For two TRSs $\mathcal{P}$ and $\mathcal{R}$ the dependency graph $\mathsf{DG}(\mathcal{P},\mathcal{R})$ contains an arc from a dependency pair α to a dependency pair β if and only if there exist substitutions σ and τ such that $\mathsf{rhs}(\alpha)\sigma \to_{\mathcal{R}}^* \mathsf{lhs}(\beta)\tau$. Without loss of generality we may assume that $\mathsf{rhs}(\alpha)\sigma$ and $\mathsf{lhs}(\beta)\tau$ are ground terms. Hence there is no arc from α to β if and only if $\Sigma(\mathsf{lhs}(\beta)) \cap \to_{\mathcal{R}}^*(\Sigma(\mathsf{rhs}(\alpha))) = \varnothing$. Here $\Sigma(t)$ denotes the set of ground instances of t with respect to the signature consisting of a fresh constant # together with all function symbols that appear in $\mathcal{P} \cup \mathcal{R}$ minus the root symbols of the left- and right-hand sides of $\mathcal{P}$ that do neither occur on positions below the root in $\mathcal{P}$ nor in $\mathcal{R}$. Since $\to_{\mathcal{R}}^*(\Sigma(\mathsf{rhs}(\alpha)))$ is in general not regular, we compute an over-approximation with the help of tree automata completion [5, 10] starting from an automaton that accepts $\Sigma(\mathsf{ren}(\mathsf{rhs}(\alpha)))$. Here ren is the function that linearizes its argument by replacing all occurrences of variables with fresh variables, which is needed to ensure the regularity of $\Sigma(\mathsf{ren}(\mathsf{rhs}(\alpha)))$.

Definition 3. Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs, L a language, and $\alpha,\beta \in \mathcal{P}$. We say that β is *unreachable* from α with respect to L if there is a tree automaton $\mathcal{A}$ compatible with $\mathcal{R}$ and $L \cap \Sigma(\mathsf{ren}(\mathsf{rhs}(\alpha)))$ such that $\Sigma(\mathsf{lhs}(\beta)) \cap \mathcal{L}(\mathcal{A}) = \varnothing$. The nodes of the c-*dependency graph* $\mathsf{DG_c}(\mathcal{P},\mathcal{R})$ are the rewrite rules of $\mathcal{P}$ and there is no arc from α to β if and only if β is unreachable from α with respect to $\Sigma(\mathsf{ren}(\mathsf{rhs}(\alpha)))$.

Lemma 4. *For two left-linear TRSs $\mathcal{P}$ and $\mathcal{R}$, $\mathsf{DG_c}(\mathcal{P},\mathcal{R}) \supseteq \mathsf{DG}(\mathcal{P},\mathcal{R})$.* $\square$

One can extend the above lemma to *arbitrary* TRSs by following the approach in [10], as described in the full version of this note.

4 Incorporating Forward Closures

When proving the termination of a right-linear TRS $\mathcal{R}$ it is sufficient to restrict attention to the set of right-hand sides of forward closures [4]. This set is defined as the closure of the right-hand sides of the rules in $\mathcal{R}$ under narrowing. Formally, given a set L of terms, $\mathsf{RFC}(L,\mathcal{R})$ is the least extension of L such that $t[r]_p\sigma \in \mathsf{RFC}(L,\mathcal{R})$ whenever $t \in \mathsf{RFC}(L,\mathcal{R})$ and there exist a non-variable position p and a fresh variant $l \to r$ of a rewrite rule in $\mathcal{R}$ with σ a most general unifier of $t|_p$ and l. In the sequel we write $\mathsf{RFC}(t,\mathcal{R})$ for $\mathsf{RFC}(\{t\},\mathcal{R})$.

Definition 5. Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. The *improved dependency graph* of $\mathcal{P}$ and $\mathcal{R}$, denoted by $\mathsf{IDG}(\mathcal{P},\mathcal{R})$, has the rules in $\mathcal{P}$ as nodes and there is an arc from $s \to t$ to $u \to v$ if and only if there exist substitutions σ and τ such that $t\sigma \to_{\mathcal{R}}^* u\tau$ and $t\sigma \in \Sigma_\#(\mathsf{RFC}(t,\mathcal{P}\cup\mathcal{R}))$. Here $\Sigma_\#$ is the operation that replaces all variables by the fresh constant #.

Theorem 6. *The* improved dependency graph processor *which maps a DP problem* $(\mathcal{P},\mathcal{R},\mathcal{G})$ *to* $\{(\mathcal{P},\mathcal{R},\mathcal{G}\cap\mathsf{IDG}(\mathcal{P},\mathcal{R}))\}$ *if* $\mathcal{P}\cup\mathcal{R}$ *is right-linear and* $\{(\mathcal{P},\mathcal{R},\mathcal{G}\cap\mathsf{DG}(\mathcal{P},\mathcal{R}))\}$ *otherwise is sound and complete.* □

Example 7. We consider again the DP problem $(\mathcal{P},\mathcal{R},\mathcal{G})$ of Example 2. Let $s = \mathsf{H}(x,y)$ and $t = \mathsf{F}(\mathsf{g}(\mathsf{a}),\mathsf{h}(x,y))$. Because each term in $\Sigma_\#(\mathsf{RFC}(s,\mathcal{P}\cup\mathcal{R}))$ is a ground instance of $\mathsf{F}(\mathsf{g}(\mathsf{a}),x)$ or $\mathsf{H}(\mathsf{a},x)$, or equal to $\mathsf{H}(\#,\#)$ and each term in $\Sigma_\#(\mathsf{RFC}(t,\mathcal{P}\cup\mathcal{R}))$ is a ground instance of $\mathsf{F}(\mathsf{g}(\mathsf{a}),x)$ or $\mathsf{H}(\mathsf{a},x)$, $\mathsf{IDG}(\mathcal{P},\mathcal{R})$ contains an arc from 2 to 1. Further arcs do not exist. So $\mathsf{IDG}(\mathcal{P},\mathcal{R})$ looks as follows:

$$\begin{tikzcd} 1 & 2 \arrow[l] & 3 \end{tikzcd}$$

The resulting DP problem $(\mathcal{P},\mathcal{R},\mathsf{IDG}(\mathcal{P},\mathcal{R}))$ can be easily show to be finite.

Similar to $\mathsf{DG}(\mathcal{P},\mathcal{R})$, $\mathsf{IDG}(\mathcal{P},\mathcal{R})$ is not computable in general. We can however over-approximate $\mathsf{IDG}(\mathcal{P},\mathcal{R})$ by using tree automata completion as described in Section 3.

Definition 8. Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. The nodes of the c-*improved dependency graph* $\mathsf{IDG}_\mathsf{c}(\mathcal{P},\mathcal{R})$ are the rewrite rules of $\mathcal{P}$ and there is no arc from α to β if and only if β is unreachable from α with respect to $\Sigma_\#(\mathsf{RFC}(\mathsf{rhs}(\alpha),\mathcal{P}\cup\mathcal{R}))$.

Lemma 9. *Let* $\mathcal{P}$ *and* $\mathcal{R}$ *be two linear TRSs. Then* $\mathsf{IDG}_\mathsf{c}(\mathcal{P},\mathcal{R}) \supseteq \mathsf{IDG}(\mathcal{P},\mathcal{R})$.

To compute $\mathsf{IDG}_\mathsf{c}(\mathcal{P},\mathcal{R})$ we have to construct a tree automaton that accepts $\mathsf{RFC}(\mathsf{rhs}(\alpha),\mathcal{P}\cup\mathcal{R})$. This can be done by using tree automata completion as described in in [6, 10]. We remark that the above lemma holds also for *right-linear* TRSs (see the description of the full version of this note).

5 Experimental Results

The techniques described in the preceding sections, extended to non-left-linear TRSs as described in the full version of this note, are integrated in the termination prover TTT2. Below we report on the experiments we performed with TTT2 on the 1331 TRSs in version 5.0 of the Termination Problem Data Base[1] that satisfy the variable condition, i.e., $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(l)$ for each rewrite rule $l \to r \in \mathcal{R}$. We used a workstation equipped with an Intel® Pentium™ M processor running at a CPU rate of 2 GHz and 1 GB of system memory. For all experiments we used a 60 seconds time limit. Our results are summarized in Table 1. We list the number of successful termination attempts, the average wall-clock

[1]http://www.termination-portal.org

Table 1: Dependency graph approximations

	e	c	r	$*$
# successes	60	67	176	183
average time	190	390	340	279
# timeouts	2	60	78	2

time needed to compute the graphs (measured in milliseconds), and the number of timeouts. Besides the SCC processor [9, 14] we used the dependency graph processor of Definition 1 and the improved dependency graph processor of Theorem 6 with $\mathsf{IDG_c}(\mathcal{P},\mathcal{R})$ ($\mathsf{DG_c}(\mathcal{P},\mathcal{R})$) for (non-)right-linear $\mathcal{P} \cup \mathcal{R}$ (r for short). To approximate dependency graphs we used the estimation described in [8] (abbreviated by e) and $\mathsf{DG_c}(\mathcal{P},\mathcal{R})$ of Definition 3 (indicated by c).

The power of the new processors is apparent, although the difference with e decreases when other DP processors are in place. An obvious disadvantage of the new processors is the large number of timeouts. This is mostly due to the unbounded number of new states to resolve compatibility violations during tree automata completion. Since the processors c and r seem to be quite fast when they terminate, an obvious idea to avoid timeouts is to equip each computation of a compatible tree automaton with a small time limit. This is shown in the column labeled $*$, which denotes the composition of e, c and r with a time limit of 500 milliseconds each for the latter two.

References

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

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

[3] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available from `www.grappa.univ-lille3.fr/tata`, 2002.

[4] N. Dershowitz. Termination of linear rewriting systems (preliminary version). In *Proc. 8th ICALP*, volume 115 of *LNCS*, pages 448–458, 1981.

[5] T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In *Proc. 9th RTA*, volume 1379 of *LNCS*, pages 151–165, 1998.

[6] A. Geser, D. Hofbauer, J. Waldmann, and H. Zantema. On tree automata that certify termination of left-linear term rewriting systems. *I&C*, 205(4):512–534, 2007.

[7] J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. 11th LPAR*, volume 3425 of *LNAI*, pages 301–331, 2004.

[8] J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proc. 5th FroCoS*, volume 3717 of *LNAI*, pages 216–231, 2005.

[9] N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *I&C*, 199(1-2):172–199, 2005.

[10] M. Korp and A. Middeldorp. Proving termination of rewrite systems using bounds. In *Proc. 18th RTA*, volume 4533 of *LNCS*, pages 273–287, 2007.

[11] M. Korp and A. Middeldorp. Match-bounds revisited. *I&C*, 2009. To appear.

[12] A. Middeldorp. Approximating dependency graphs using tree automata techniques. In *Proc. 1st IJCAR*, volume 2083 of *LNAI*, pages 593–610, 2001.

[13] A. Middeldorp. Approximations for strategies and termination. In *Proc. 2nd WRS*, volume 70 of *ENTCS*, pages 1–20, 2002.

[14] R. Thiemann. *The DP Framework for Proving Termination of Term Rewriting*. PhD thesis, RWTH Aachen, 2007. Available as technical report AIB-2007-17.

A Mechanized Proof Reconstruction for SCNP Termination

Alexander Krauss[*] Armin Heller

Institut für Informatik
Technische Universität München

1 Introduction

Ben-Amram and Codish described SCNP [2], a subclass of the size-change termination criterion SCT [8], which permits efficient certificate checking. Termination problems in this class have a global ranking function of a certain form, which can be found using SAT solving.

This note describes an automated proof reconstruction for this certificate scheme, implemented in the theorem prover Isabelle/HOL [9]. In previous work [6], we have shown how to use the full size-change principle for termination proofs of recursive function definitions in Isabelle. Although the certificate-based approach is less powerful in theory, it has practical advantages:

- The transitive closure computation in [8] is an efficiency bottleneck, and optimizing it is hard since the code must be proved correct and executed within the logical system.
- Certificates can be stored, which makes proof checking easier when the proof script is re-run.
- Much less logical infrastructure is necessary. In particular, no formalization of Ramsey's theorem is required, which makes the approach portable to theorem provers with a constructive foundation, such as Coq [3].

Our method is included in the recent release of Isabelle 2009 (`http://isabelle.in.tum.de`).

2 Termination Goals

Isabelle termination goals arise from recursive function definitions and express the wellfoundedness of the call relation extracted from the definition. The induction principle encoded in this relation is then used internally to construct an explicit model of the function.

Termination goals in Isabelle have the following form:

$$wf\ (\{(r_1, l_1) \mid \Gamma_1\} \cup \ldots \cup \{(r_n, l_n) \mid \Gamma_n\})$$

Here, each recursive call in the function definition is expressed as a relation comprehension $\{(r_i, l_i) \mid \Gamma_i\}$, where l_i is the argument from the left hand side of the definition, r_i is the argument of the recursive call, and Γ_i is the condition that leads to the recursive call. The goal is to prove that the union of these relations is wellfounded.[1]

Example. The function *perm* below is taken from [8]. The corresponding termination goal consists of two calls:

$$perm\ (m, n, r) = (\text{if } 0 < r \text{ then } perm\ (m, r-1, n) \text{ else if } 0 < n \text{ then } perm\ (r, n-1, m) \text{ else } m)$$

$$wf\ (\{((m, r-1, n), (m, n, r)) \mid 0 < r\} \cup \{((r, n-1, m), (m, n, r)) \mid \neg\, 0 < r \wedge 0 < n\})$$

This kind of proof obligation is typical for the tradition of modelling programs as functions in higher-order logic. In this shallow embedding, where the syntax of programs is not modelled explicitly, our setting differs from that of the CoLoR [4] and A3PAT [5] projects and the recent CeTA tool [10]. However, our proof reconstruction should be applicable in that context, too.

For this presentation, we ignore many issues that tend to complicate things, like mutual and nested recursion, and just focus on the reconstruction.

[*] `http://www.in.tum.de/~krauss`

[1] The wellfoundedness predicate *wf* is defined in such a way that the "smaller" element is written left, contrary to conventions in rewriting.

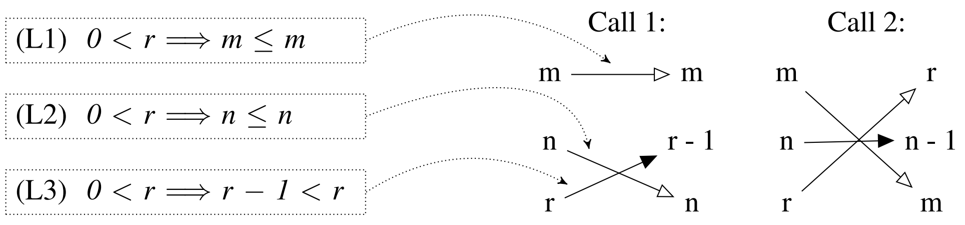

(L1) $0 < r \Longrightarrow m \le m$
(L2) $0 < r \Longrightarrow n \le n$
(L3) $0 < r \Longrightarrow r - 1 < r$
Call 1:
m
m
n
r - 1
r
n
Call 2:
m
r
n
n - 1
r
m

Figure 1: The size-change graphs generated for *perm* and the underlying theorems

3 SCNP = Size-Change Termination in NP

The SCNP criterion uses the abstraction known from size-change termination: A termination problem is represented as an abstract control-flow graph, whose edges carry information about the relative sizes of data in the transitions, depicted as size-change graphs. In Isabelle, this information is derived using off-the-shelf theorem proving methods, producing a theorem for each edge in a size-change graph. Figure 1 shows the size-change graphs generated for the function *perm* and the underlying theorems.

Instead of computing the transitive closure of the set of size-change graphs, the SCNP approach is based on orderings. It uses the following extensions of wellfounded relations to finite multisets:

Definition 1 (Max-, min-, and multiset extension).
$\mathit{max\text{-}ext}\ R = \{(X, Y) \mid Y \neq \emptyset \wedge (\forall x{\in}X.\ \exists y{\in}Y.\ (x, y) \in R)\}$
$\mathit{min\text{-}ext}\ R = \{(X, Y) \mid X \neq \emptyset \wedge (\forall y{\in}Y.\ \exists x{\in}X.\ (x, y) \in R)\}$
$\mathit{ms\text{-}ext}\ R = \{(X, Y) \mid \exists a\, Z\, K.\ Y = Z \cup \{a\} \wedge X = Z \cup K \wedge (\forall x{\in}K.\ (x, a) \in R)\}^+$

Intuitively, *max-ext* and *min-ext* compare sets by their maximum or minimum, and *ms-ext* is the standard multiset order [1]. Ben-Amram and Codish also use a fourth extension called *dual multiset order*, but experiments show that it contributes very little to the power of the approach, and we decided not to include it. The extensions above are applied to the lexicographic order on pairs of natural numbers. Formally:

$$\begin{array}{lll} \mathit{lex}_< \ = \ \{((a, b), (c, d)) \mid a < c \vee a \le c \wedge b < d\}, & & \mathit{lex}_\le = \mathit{lex}_< \cup \mathit{Id}, \\ \mathit{max}_< = \mathit{max\text{-}ext}\ \mathit{lex}_<, & \mathit{min}_< = \mathit{min\text{-}ext}\ \mathit{lex}_<, & \mathit{ms}_< = \mathit{ms\text{-}ext}\ \mathit{lex}_<, \\ \mathit{max}_\le = \mathit{max\text{-}ext}\ \mathit{lex}_\le \cup \{(\emptyset, \emptyset)\}, & \mathit{min}_\le = \mathit{min\text{-}ext}\ \mathit{lex}_\le \cup \{(\emptyset, \emptyset)\}, & \mathit{ms}_\le = \mathit{ms}_< \cup \mathit{Id}. \end{array}$$

Using our comprehension notation, a call $\{(r_i, l_i) \mid \Gamma_i\}$ is said to be *R-decreasing under a map f* if $\{(f\, r_i, f\, l_i) \mid \Gamma_i\} \subseteq R$ or, equivalently, $\Gamma_i \Longrightarrow (f\, r_i, f\, l_i) \in R$. We are looking for a map under which some calls are $\mu_<$-decreasing and the remaining calls are $\mu_\le$-decreasing for some $\mu \in \{\mathit{max}, \mathit{min}, \mathit{ms}\}$. Then the $\mu_<$-decreasing calls can be removed from the termination problem and the process is repeated (possibly with a different ordering) until no calls are left.

The SCNP approach uses maps which assign a multiset of pairs to each argument vector. The first component of the pairs is the size of some argument[2], and the second is a fixed natural number called a *tag*. A function of this kind is called a *level mapping*. There is no intuitive motivation for the tags, but they add an extra degree of freedom to the analysis, thus making it slightly more powerful in some cases.

Example (cont'd). The termination proof for the function *perm* above uses the level mapping $\lambda(m, n, r).\ \{(m, 0), (n, 0), (r, 0)\}$. Both calls are $\mathit{ms}_<$-decreasing under that level mapping.

[2]Isabelle generates size measures heuristically based on the types. The size of a natural number is the number itself.

$max_<$:	(1)	$Y \neq \emptyset \Longrightarrow (\emptyset, Y) \in max_<$
	(2)	$y \in Y \Longrightarrow (x, y) \in lex_< \Longrightarrow (X, Y) \in max_< \Longrightarrow (\{x\} \cup X, Y) \in max_<$
$ms_<$:	(3)	$(Z, Z') \in pairwise_\leq \Longrightarrow (A, B) \in max_< \Longrightarrow (Z \cup A, Z' \cup B) \in ms_<$
$pairwise_\leq$:	(4)	$(\emptyset, \emptyset) \in pairwise_\leq$
	(5)	$(x, y) \in lex_\leq \Longrightarrow (X, Y) \in pairwise_\leq \Longrightarrow (\{x\} \cup X, \{y\} \cup Y) \in pairwise_\leq$
$lex_{</\leq}$:	(6)	$a < b \Longrightarrow ((a, s), (b, t)) \in lex_<$
	(7)	$a \leq b \Longrightarrow s < t \Longrightarrow ((a, s), (b, t)) \in lex_<$
	(8)	$a < b \Longrightarrow ((a, s), (b, t)) \in lex_\leq$
	(9)	$a \leq b \Longrightarrow s \leq t \Longrightarrow ((a, s), (b, t)) \in lex_\leq$

Figure 2: Introduction rules for proof reconstruction

Certificates A certificate is the justification for removing some calls from a termination problem. It contains the following information:

1. The ordering $\mu \in \{max, min, ms\}$,
2. the level mapping f, which maps the argument vectors to multisets,
3. the calls that are $\mu_<$-decreasing under f, and
4. a *covering function* for each call, which contains information required in the proof that the call is decreasing. If $\mu \in \{max, min\}$, the function provides the instantiations of the existential quantifiers in Def. 1. If $\mu = ms$, it provides the decomposition of the multisets used in rule (3) below.

To construct certificates, we use the SAT-encoding given by Ben-Amram and Codish [2]. By calling an external SAT solver, we obtain a satisfying assignment from which we obtain the information above.

4 Proof Reconstruction

The main task in the proof reconstruction is to formally derive the inclusions $\{(f\,r_i, f\,l_i) \mid \Gamma_i\} \subseteq \mu_{</\leq}$ that are postulated by the certificate. For this purpose we derive a set of simple facts about $max_{</\leq}$, $min_{</\leq}$, $ms_{</\leq}$, and $lex_{</\leq}$, which serve as introduction rules for the respective relations. Figure 2 shows some of the rules; the complete list can be found in [7]. The actual reconstruction consists of applying these rules in a controlled manner. To express intermediate subgoals, we must introduce the auxiliary relation $(X, Y) \in pairwise_\leq$ which expresses that there is a one-to-one correspondence between the respective elements of X and Y, such that corresponding elements $(x, y) \in lex_\leq$.

Proving that two given sets are $max_<$-decreasing amounts to showing that for each $x \in X$ there is a $y \in Y$ such that $x < y$ using rules (1) and (2). At each application of rule (2), the variable y must be instantiated with an element of Y, which is given by the covering function in the certificate. Multiset descent can be reduced to max-descent using rule (3), which decomposes the multisets into two parts. The decomposition is part of the certificate.

Example (cont'd). To illustrate the automated proof reconstruction, we show a detailed proof that the first call of *perm* is $ms_<$-decreasing. Thus we start with the following goal, which we will refine gradually:

$$0 < r \Longrightarrow (\{(m, 0), (r-1, 0), (n, 0)\}, \{(m, 0), (r, 0), (n, 0)\}) \in ms_<$$

We rewrite the multisets using associativity and commutativity to split them into two parts, as in rule (3):

$$0 < r \Longrightarrow (\{(m, 0), (n, 0)\} \cup \{(r-1, 0)\}, \{(m, 0), (n, 0)\} \cup \{(r, 0)\}) \in ms_<$$

After applying rule (3), we get two subgoals:

$$0 < r \Longrightarrow (\{(m, 0), (n, 0)\}, \{(m, 0), (n, 0)\}) \in pairwise_\leq$$

$$0 < r \Longrightarrow (\{(r-1, 0)\}, \{(r, 0)\}) \in max_<$$

We apply the introduction rules for $\mathit{pairwise}_{\leq}$ to the first subgoal, obtaining the following goal state:

$$0 < r \Longrightarrow ((m, 0), (m, 0)) \in \mathit{lex}_{\leq}$$
$$0 < r \Longrightarrow ((n, 0), (n, 0)) \in \mathit{lex}_{\leq}$$
$$0 < r \Longrightarrow (\{(r - 1, 0)\}, \{(r, 0)\}) \in \mathit{max}_{<}$$

Then we apply the introduction rules for $\mathit{lex}_{\leq}$:

$$0 < r \Longrightarrow m \leq m$$
$$0 < r \Longrightarrow 0 \leq 0$$
$$0 < r \Longrightarrow n \leq n$$
$$0 < r \Longrightarrow 0 \leq 0$$
$$0 < r \Longrightarrow (\{(r - 1, 0)\}, \{(r, 0)\}) \in \mathit{max}_{<}$$

Now, subgoals 1 and 3 are precisely the local descent properties (L1) and (L2) that we have already proved (cf. Fig. 1). Subgoals 2 and 4 are trivial inequalities between number literals (the tags) and are easily discharged. For the last subgoal, we apply rule (2), instantiating y with $(r, 0)$. This yields

$$0 < r \Longrightarrow ((r - 1, 0), (r, 0)) \in \mathit{lex}_{<}$$
$$0 < r \Longrightarrow (\emptyset, \{(r, 0)\}) \in \mathit{max}_{<}$$

Then, using rule (6), we can apply the local descent property (L3). Rule (1) solves the last subgoal.

Fortunately, these tedious proofs are fully automated and invisible to the user.

Acknowledgements We thank Amir Ben-Amram and Michael Codish for answering all our questions on SCNP details.

References

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

[2] A. Ben-Amram and M. Codish. A SAT-based approach to size change termination with global ranking functions. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, pages 218–232. Springer, March 2008.

[3] Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Texts in theoretical computer science. Springer, 2004.

[4] F. Blanqui, S. Coupet-Grimal, W. Delobel, S. Hinderer, and A. Koprowski. CoLoR, a Coq library on rewriting and termination. In A. Geser and H. Söndergaard, editors, *International Workshop on Termination (WST'06)*, 2006.

[5] E. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Certification of automated termination proofs. In B. Konev and F. Wolter, editors, *Frontiers of Combining Systems (FroCoS'07)*, volume 4720 of *LNAI*, pages 148–162. Springer, 2007.

[6] A. Krauss. Certified size-change termination. In F. Pfenning, editor, *Automated Deduction (CADE-21)*, volume 4603 of *LNCS*, pages 460–476. Springer, 2007.

[7] A. Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic.* PhD thesis, Institut für Informatik, Technische Universität München, Germany, January 2009. Submitted.

[8] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Principles of Programming Languages (PoPL 2001)*, pages 81–92, 2001.

[9] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[10] R. Thiemann and C. Sternagel. Certification of termination proofs with CeTA. In T. Nipkow and C. Urban, editors, *Theorem Proving in Higher Order Logics (TPHOLs '09)*, LNCS. Springer, 2009. To appear.

Monotonicity of Parametric Polynomials

Friedrich Neurauter
Institute of Computer Science
University of Innsbruck
6020 Innsbruck, Austria
`friedrich.neurauter@uibk.ac.at`

Abstract

In this note we investigate on monotonicity and well-definedness of parametric polynomials used in automated polynomial termination proofs, providing exact constraints on the abstract coefficients such that the above properties are guaranteed.

1 Introduction

This note is concerned with automatically proving termination of term rewrite systems by means of polynomial interpretations over the natural numbers. In the classical approach due to Lankford [5], we associate with every n-ary function symbol f a polynomial P_f in n indeterminates over the natural numbers, which induces a mapping or *interpretation* from terms to natural numbers in the obvious way. In order to conclude termination of a given TRS, three conditions have to be satisfied. First, every polynomial must be *well-defined*, i.e., it must induce a well-defined polynomial function $f_{\mathbb{N}} : \mathbb{N}^n \to \mathbb{N}$. In addition, the interpretation functions $f_{\mathbb{N}}$ are required to be *strictly monotone* in all arguments. Finally, one has to show *compatibility* of the interpretation with the given TRS. These three requirements essentially carry over to the case of using polynomial interpretations as reduction pairs in the dependency pair framework [1, 2, 3], but in a weakened form. Most notably, the interpretation functions are merely required to be weakly monotone.

In an automated setting, termination tools are concerned with parametric polynomials whose coefficients (i.e., the parameters) are initially unknown and have to be instantiated suitably such that the resulting concrete polynomials satisfy the above conditions. In this note we focus on monotonicity and well-definedness of certain kinds of parametric polynomials (linear, quadratic, etc.), two conditions that are independent of the respective term rewrite system considered. The aim is to provide exact constraints in terms of the abstract coefficients of a parametric polynomial such that monotonicity and well-definedness of the resulting concrete polynomial are guaranteed for every instantiation of the coefficients that satisfies the constraints.

Our approach subsumes the absolute positiveness approach [4], which is currently used in many termination tools. While the latter approach essentially restricts all coefficients to be non-negative, our approach allows for negative numbers in certain coefficients.

2 Parametric Polynomials

Polynomial interpretations over the natural numbers are based on the well-founded algebra $(\mathcal{N}, >)$, where $>$ is the standard order on $\mathbb{N}$ and $\mathcal{N} = (\mathbb{N}, \{f_{\mathbb{N}}\}_{f \in \mathcal{F}})$ such that every algebra operation $f_{\mathbb{N}} : \mathbb{N}^n \to \mathbb{N}$ is a polynomial. Depending on whether the algebra operations are strictly or weakly monotone, $(\mathcal{N}, >)$ is either a well-founded monotone algebra or a well-founded weakly monotone algebra. Note, however, that this does not imply that all coefficients of the polynomials must be natural numbers; for example, the polynomial $2x^2 - x + 1 \in \mathbb{Z}[x]$ gives rise to the polynomial function $f_{\mathbb{N}}(x) = 2x^2 - x + 1$, which is both strictly monotone and well-defined, i.e., for every natural number x the image $f_{\mathbb{N}}(x)$ is again a natural number.

Summing up, an n-ary polynomial function $f_{\mathbb{N}}$ used in a polynomial interpretation is an element of the polynomial ring $\mathbb{Z}[x_1, \ldots, x_n]$, which has to satisfy the following two properties:

1. Well-definedness $f_{\mathbb{N}}(x_1,\ldots,x_n) \geq 0$ for all $x_1,\ldots,x_n \in \mathbb{N}$.

2. Strict (weak) monotonicity $x_i > y \implies f_{\mathbb{N}}(x_1,\ldots,x_i,\ldots,x_n) \underset{(\geq)}{>} f_{\mathbb{N}}(x_1,\ldots,y,\ldots,x_n)$ for all $x_1,\ldots,x_n,y \in \mathbb{N}$ and $i \in \{1,\ldots,n\}$.

Both of these properties are instances of the undecidable problem of checking *positiveness of polynomials*[1] in the polynomial ring $\mathbb{Z}[x_1,\ldots,x_n]$ (undecidable by reduction from Hilbert's 10^{th} problem). Termination tools face an even tougher problem. That is to say, they have to deal with parametric polynomials, i.e., polynomials whose coefficients are unknowns (e.g., ax^2+bx+c), and the task is to find suitable (integer) numbers for the unknown coefficients such that the resulting polynomials satisfy both of the above properties. The solution that is used in practice is to restrict the search space for the unknown coefficients to the non-negative integers (absolute positiveness approach [4]) because then well-definedness and weak monotonicity are obtained for free. To obtain strict monotonicity in the i-th argument of a polynomial function, the coefficient associated with the monomial x_i must be at least 1; e.g., the polynomial ax^2+bx+c is both well-defined and strictly monotone if $a \geq 0$, $b > 0$ and $c \geq 0$.

Obviously, this approach is easy to implement and works quite well in practice, but it is not satisfactory because it is not optimal in the sense that it excludes certain polynomials, like $2x^2-x+1$, which might be useful to prove termination of certain term rewrite systems. So how can we do better? Therefore let us observe that in general termination tools only use restricted forms of polynomials to interpret function symbols, for which monotonicity and well-definedness might be decidable.

In the remainder of this note we analyze parametric polynomials whose only restriction is a bound on the degree. We will first treat linear parametric polynomials. While this does not yield new results or insights, it is instructive to demonstrate our approach in a simple setting. This is followed by an analysis of quadratic and finally also cubic parametric polynomials, both of which yield new results. The following lemmas will be helpful in this analysis. The first one gives a more succinct characterization of monotonicity, whereas the second one relates monotonicity and well-definedness.

Lemma 2.1. *The function $f_{\mathbb{N}} : \mathbb{N}^n \to \mathbb{N}$ (not necessarily polynomial!) is strictly (weakly) monotone in all arguments iff $f_{\mathbb{N}}(x_1,\ldots,x_i+1,\ldots,x_n) \underset{(\geq)}{>} f_{\mathbb{N}}(x_1,\ldots,x_i,\ldots,x_n)$ for all $x_1,\ldots,x_n \in \mathbb{N}$ and $i \in \{1,\ldots,n\}$.*

Lemma 2.2. *Let $f : \mathbb{Z}^n \to \mathbb{Z}$ be the polynomial function associated with a polynomial in $\mathbb{Z}[x_1,\ldots,x_n]$, and let $f_{\mathbb{N}} : \mathbb{N}^n \to \mathbb{Z}$ denote its restriction to $\mathbb{N}$. Then $f_{\mathbb{N}}$ is strictly (weakly) monotone (in all arguments) and well-defined if and only if it is strictly (weakly) monotone and $f_{\mathbb{N}}(0,\ldots,0) \geq 0$.*

2.1 Linear and Quadratic Parametric Polynomials

In this section we consider the generic linear parametric polynomial function $f_{\mathbb{N}}(x_1,\ldots,x_n) = a_nx_n+\ldots+a_1x_1+a_0$, deriving constraints on the coefficients a_i that guarantee monotonicity and well-definedness.

Theorem 2.3. *The function $f_{\mathbb{N}}(x_1,\ldots,x_n) = a_nx_n+\ldots+a_1x_1+a_0$ $(a_i \in \mathbb{Z}, 0 \leq i \leq n)$ is strictly (weakly) monotone and well-defined if and only if $a_0 \geq 0$ and $a_i > 0$ $(a_i \geq 0)$ for all $i \in \{1,\ldots,n\}$.*

Proof. By Lemmata 2.1 and 2.2, this theorem holds if and only if $f_{\mathbb{N}}(0,\ldots,0) \geq 0$, and for all $x_1,\ldots,x_n \in \mathbb{N}$ and all $i \in \{1,\ldots,n\}$, $f_{\mathbb{N}}(x_1,\ldots,x_i+1,\ldots,x_n) \underset{(\geq)}{>} f_{\mathbb{N}}(x_1,\ldots,x_i,\ldots,x_n)$. Obviously, $f_{\mathbb{N}}(0,\ldots,0) \geq 0$ if and only if $a_0 \geq 0$. Moreover, by definition of $f_{\mathbb{N}}$, $f_{\mathbb{N}}(x_1,\ldots,x_i+1,\ldots,x_n) > f_{\mathbb{N}}(x_1,\ldots,x_i,\ldots,x_n)$ is equivalent to $a_nx_n+\ldots+a_i(x_i+1)+\ldots+a_1x_1+a_0 > a_nx_n+\ldots+a_ix_i+\ldots+a_1x_1+a_0$, which holds if and only if $a_i > 0$. This proves the claim for strict monotonicity; for weak monotonicity we just have to replace $>$ by $\geq$ in the above calculation. □

[1]Given $P \in \mathbb{Z}[x_1,\ldots,x_n]$, decide $\forall x_1,\ldots,x_n \in \mathbb{N}$ $P(x_1,\ldots,x_n) > 0$

Note that all coefficients are non-negative, and the final result corresponds to the absolute positiveness approach. Next we apply the approach illustrated in the proof of Theorem 2.3 to the generic quadratic parametric polynomial function $f_{\mathbb{N}}(x_1,\dots,x_n) = a_0 + \sum_{i=1}^{n} a_i x_i + \sum_{i=1}^{n}\sum_{j=i}^{n} a_{ij}x_i x_j \in \mathbb{Z}[x_1,\dots,x_n]$.

Theorem 2.4. *The function $f_{\mathbb{N}}$ is strictly (weakly) monotone and well-defined if and only if $a_0 \geq 0$, $a_{ij} \geq 0$ and $a_i > -a_{ii}$ $(a_i \geq -a_{ii})$ for all $1 \leq i \leq j \leq n$.*

Proof. By Lemmata 2.1 and 2.2, this theorem holds if and only if $f_{\mathbb{N}}(0,\dots,0) \geq 0$, and for all $x_1,\dots,x_n \in \mathbb{N}$ and all $i \in \{1,\dots,n\}$, $f_{\mathbb{N}}(x_1,\dots,x_i+1,\dots,x_n) \underset{(\geq)}{>} f_{\mathbb{N}}(x_1,\dots,x_i,\dots,x_n)$. Clearly, $f_{\mathbb{N}}(0,\dots,0) \geq 0$ if and only if $a_0 \geq 0$. Identifying a_{jk} and a_{kj}, $f_{\mathbb{N}}(x_1,\dots,x_i+1,\dots,x_n) > f_{\mathbb{N}}(x_1,\dots,x_i,\dots,x_n)$ becomes $a_i(x_i+1) + a_{ii}(x_i+1)^2 + \sum_{1\leq j\leq n, j\neq i} a_{ij}(x_i+1)x_j > a_i x_i + a_{ii}x_i^2 + \sum_{1\leq j\leq n, j\neq i} a_{ij}x_i x_j$, which simplifies to $a_i + 2a_{ii}x_i + a_{ii} + \sum_{1\leq j\leq n, j\neq i} a_{ij}x_j > 0$. This holds for all $x_1,\dots,x_n \in \mathbb{N}$ if and only if $a_{ij} \geq 0$ $(j \neq i)$ and $2a_{ii}x_i + a_{ii} + a_i > 0$ for all $x_i \in \mathbb{N}$. Now this last inequality holds if and only if $a_{ii} \geq 0$ and $a_i > -a_{ii}$. Altogether, this proves the claim for strict monotonicity; for weak monotonicity we just have to replace $>$ by $\geq$ in the above calculation. □

Corollary 2.5. *The function $f_{\mathbb{N}}(x) = ax^2 + bx + c$ is strictly (weakly) monotone and well-defined if and only if $a \geq 0$, $c \geq 0$ and $b > -a$ $(b \geq -a)$.*

Hence, in a quadratic parametric polynomial all coefficients must be non-negative except the coefficients of the linear monomials. Finally, note the following. Not only does our approach subsume the absolute positiveness approach, but the results derived from it are even optimal, i.e., necessary and sufficient for monotonicity and well-definedness.

2.2 Cubic Parametric Polynomials

Next we apply our approach to the cubic parametric polynomial function $f_{\mathbb{N}}(x) = ax^3 + bx^2 + cx + d \in \mathbb{Z}[x_1,\dots,x_n]$, and we run into a problem. That is to say, the monotonicity constraint $\forall x \in \mathbb{N}\ f_{\mathbb{N}}(x+1) > f_{\mathbb{N}}(x)$ turns into

$$\forall x \in \mathbb{N}\ 3ax^2 + (3a+2b)x + (a+b+c) > 0, \tag{1}$$

whose exact solution poses a problem if $a \neq 0$, which is of course the interesting case. Since a is involved in the lead coefficient of $P := 3ax^2 + (3a+2b)x + (a+b+c)$, it must necessarily be positive in order for (1) to hold. Next we distinguish two possible cases. In the first case, when P has no roots in $\mathbb{R}$, (1) trivially holds. Moreover, this case is completely characterized algebraically by the discriminant of P being negative, i.e., $4b^2 - 3a^2 - 12ac < 0$. In the other case, when both roots r_1 and r_2 are real numbers, (1) holds if and only if the closed interval $[r_1, r_2]$ does not contain a natural number, i.e., $[r_1, r_2] \cap \mathbb{N} = \emptyset$. While we can easily characterize r_1 and r_2 algebraically, it is not clear how to characterize the condition $[r_1, r_2] \cap \mathbb{N} = \emptyset$ succinctly. However, if we require the larger of the two roots, that is, r_2, to be negative, then (1) is guaranteed to hold. This observation leads to the constraints $4b^2 - 3a^2 - 12ac \geq 0$ and $r_2 = \frac{-(3a+2b)+\sqrt{4b^2-3a^2-12ac}}{6a} < 0$. Putting everything together and simplifying the constraints as much as possible, we obtain the following theorem.

Theorem 2.6. *The function $f_{\mathbb{N}}(x) = ax^3 + bx^2 + cx + d$ is strictly monotone and well-defined if $a \geq 0$, $d \geq 0$ and either $4b^2 - 3a^2 - 12ac < 0$ or $4b^2 - 3a^2 - 12ac \geq 0$, $a+b+c > 0$ and $3a+2b \geq 0$.*

Note that these conditions are only sufficient for monotonicity and well-definedness, they are not necessary conditions. Furthermore, this result is not yet nice if one has automation in mind. This is due to the case distinction, which gives rise to exponential behaviour. However, we can get rid of it by applying some basic logic:

Corollary 2.7. *The function $f_{\mathbb{N}}(x) = ax^3 + bx^2 + cx + d$ is strictly monotone and well-defined if $a \geq 0$, $d \geq 0$, $a+b+c > 0$ and $b \geq -3a/2$.*

The result of Corollary 2.7 is weaker than the result of Theorem 2.6, but still better than the absolute positiveness approach because b and c are allowed to be negative.

2.3 Negative Coefficients in Polynomial Interpretations

In the previous subsections we have seen that in principle we may use polynomial interpretations with (some) negative coefficients for proving termination of rewrite systems. Now the obvious question is the following: Does there exist a TRS that can be proved terminating by a polynomial interpretation with negative coefficients according to Theorems 2.4 and 2.6, but cannot be proved terminating by a polynomial interpretation where the coefficients of all polynomials are non-negative?

To elaborate on this question, let us consider the following scenario. Assume we have a TRS whose signature contains (amongst others) the *successor* symbol s, the constant 0 and another unary symbol f, and assume that the interpretations associated with the former two are the natural interpretations $\mathsf{s}_{\mathbb{N}}(x) = x+1$ and $0_{\mathbb{N}} = 0$, whereas f is supposed to be interpreted by $\mathsf{f}_{\mathbb{N}}(x) = ax^2 + bx + c$. Now the idea is to add rules to the TRS which force $\mathsf{f}_{\mathbb{N}}(x) = 2x^2 - x + 1$. This can be achieved as follows.

First, note that by polynomial interpolation the coefficients a, b, c of the polynomial function $\mathsf{f}_{\mathbb{N}}(x) = ax^2 + bx + c$ are uniquely determined by the image of $\mathsf{f}_{\mathbb{N}}$ at three pairwise different locations; for example, the constraints $\mathsf{f}_{\mathbb{N}}(0) = 1, \mathsf{f}_{\mathbb{N}}(1) = 2$ and $\mathsf{f}_{\mathbb{N}}(2) = 7$ enforce $\mathsf{f}_{\mathbb{N}}(x) = 2x^2 - x + 1$, as desired. Now the only thing that remains to be done is to encode these three constraints in terms of some rewrite rules:

$$\begin{array}{rclrcl}
\mathsf{f}(0) & \to & 0 & \mathsf{s}^2(0) & \to & \mathsf{f}(0) \\
\mathsf{f}(\mathsf{s}(0)) & \to & \mathsf{s}(0) & \mathsf{s}^3(0) & \to & \mathsf{f}(\mathsf{s}(0)) \\
\mathsf{f}(\mathsf{s}^2(0)) & \to & \mathsf{s}^6(0) & \mathsf{s}^8(0) & \to & \mathsf{f}(\mathsf{s}^2(0))
\end{array}$$

Every constraint gives rise to two rewrite rules; for example, the constraint $\mathsf{f}_{\mathbb{N}}(0) = 1$ is expressed by the two rewrite rules $\mathsf{f}(0) \to 0$ and $\mathsf{s}^2(0) \to \mathsf{f}(0)$ in the first line. The former encodes $\mathsf{f}_{\mathbb{N}}(0) > 0$, whereas the latter encodes $\mathsf{f}_{\mathbb{N}}(0) < 2$. So these rewrite rules are polynomially terminating by construction, with $\mathsf{f}_{\mathbb{N}}(x) = 2x^2 - x + 1$. Moreover, the assumption to interpret f by a quadratic polynomial function is not necessary because any feasible interpretation $\mathsf{f}_{\mathbb{N}}$ must necessarily contain at least one monomial with a negative coefficient. Therefore let us observe that no linear interpretation for f is feasible because the set of points $\{(i, \mathsf{f}_{\mathbb{N}}(i))\}_{i \in \{0,1,2\}}$ is not collinear. The case when $\mathsf{f}_{\mathbb{N}}$ is quadratic was dealt with above. So let us consider interpretations of degree at least three. Then the lead monomial of $\mathsf{f}_{\mathbb{N}}$ has the shape ax^k, where $a \geq 1$ and $k \geq 3$. Using the fact that the constraint $\mathsf{f}_{\mathbb{N}}(2) = 7$ must be satisfied, the claim follows immediately because for $x = 2$ the lead monomial alone contributes a value of at least 8.

References

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

[2] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In Franz Baader and Andrei Voronkov, editors, *LPAR*, volume 3452 of *Lecture Notes in Computer Science*, pages 301–331. Springer, 2004.

[3] Nao Hirokawa and Aart Middeldorp. Automating the dependency pair method. *Inf. Comput.*, 199(1-2):172–199, 2005.

[4] Hoon Hong and Dalibor Jakus. Testing positiveness of polynomials. *J. Autom. Reasoning*, 21(1):23–38, 1998.

[5] Dallas S. Lankford. On proving term rewriting systems are noetherian. Technical Report MTP–3, Louisiana Technical University, 1979.

Termination Analysis of Java Bytecode by Term Rewriting[*]

C. Otto, M. Brockschmidt, C. von Essen, J. Giesl
LuFG Informatik 2
RWTH Aachen University
Aachen, Germany

1 Introduction

We present an automated approach to prove termination of Java Bytecode [7] programs by automatically transforming them to term rewrite systems (TRSs). In this way, the numerous techniques and tools developed for TRS termination can also be used for imperative object-oriented programming languages that can be compiled into Java Bytecode. Compared to direct termination analysis of imperative programs (e.g., [2, 3]), rewrite techniques have the advantage that they are very powerful for algorithms on user-defined data structures, since they can automatically generate suitable well-founded orders comparing arbitrary forms of terms. Moreover, by using term rewriting with built-in integers [4], rewrite techniques are also powerful for algorithms on pre-defined data structures like integers. Our work is related to [1, 8] where Java Bytecode is transformed into constraint logic programming and techniques from logic programming are used to prove termination of the Bytecode program.

To translate a Java Bytecode program to a TRS, we adapt an approach which was already successfully used to prove termination of Haskell programs by term rewriting [5]. More precisely, we construct a *termination graph* representing all execution paths of the program. Sect. 2 illustrates this for programs on primitive data types and Sect. 3 extends the termination graph construction to programs on user-defined types (operating on the heap). Afterwards, a set of TRSs is generated from the termination graph such that termination of the TRSs implies termination of the original Bytecode program (Sect. 4).

2 Termination Graphs for Programs with Primitive Data Types

We first construct a termination graph by interpreting the code of the given Java Bytecode program. As long as the current memory state is completely known, this interpretation is well defined by the specification of the Java Virtual Machine (JVM) [7]. Every state (defined as the values of program counter, stack, and heap), provides all information needed to define all successor states along the execution path.

However, we do not only want to prove termination of methods for specific given input values (e.g., termination of `f(4,3)` for some method `f`), but we want to prove termination of a method for *all* possible input values (e.g., of `f(x,y)` for all $\mathtt{x},\mathtt{y} \in \mathbb{Z}$). Thus, we obtain *abstract* states with abstract information like "$\mathtt{x} : \mathbb{Z}$" instead of concrete information like "`x == 4`". If the current state does not provide enough information to deterministically interpret the next instruction, we refine the state information such that in the next step, evaluation can continue as defined in the specification of the JVM. These refinement steps usually introduce branches in the sequence of evaluation steps, as can be seen in the left part of Fig. 1. It shows the termination graph for the (abstracted) code snippet[1] "`boolean x = ?; if (x) { f(); } else { g(); }`". Here, the value of `x` is unknown, i.e., we only know $\mathtt{x} \in \mathbb{B} = \{\mathtt{true}, \mathtt{false}\}$.

In general, refinement and evaluation steps might be repeated infinitely often. To avoid infinite (or too long) paths in the termination graph, we perform abstraction steps where we deliberately lose information. This can be seen in the termination graph for the code snippet "`int x = -10; while(x < 1000) { x++; }; h();`", cf. the right part of Fig. 1. Here, when just interpreting the code, the loop would cause a huge number of states, where the value of `x` is incremented in each iteration.

Instead, if we abstract away the initial knowledge "`x == -10`" (Node 6) to "$\mathtt{x} : \mathbb{Z}$" (Node 7), then the state after a single loop iteration (Node 12) is an *instance* of the state before (Node 7). Hence, instead of

[*]Supported by the DFG grant GI 274/5-2 and by the G.I.F. grant 966-116.6.
[1]For simplicity, we give the (pseudo) code in Java [6] and not in Java Bytecode.

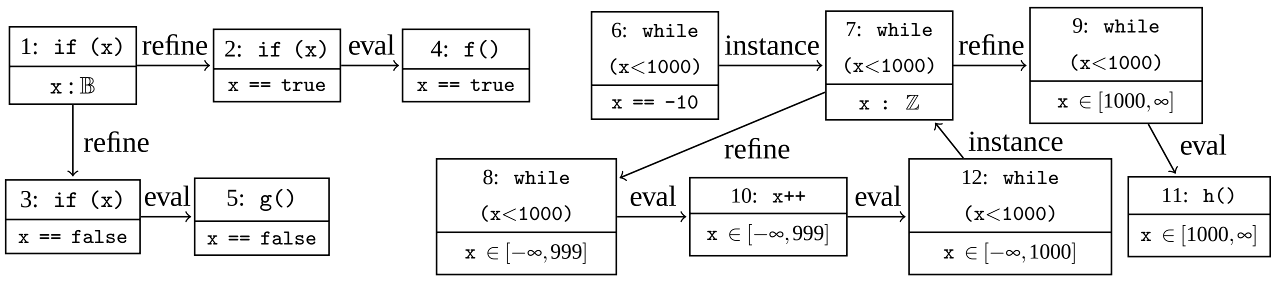


Figure 1: Termination Graphs

evaluating Node 12 further, we connect it back to Node 7. Now the original loop of the program results in a loop in the constructed graph. By losing information, we may of course have to handle execution paths that were not possible in the original program. Therefore, our approach cannot prove non-termination.

Of course, we need heuristics to decide when to abstract away information (as in the step from Node 6 to Node 7). To this end, we count how often an instruction is evaluated when building up the termination graph. If an instruction is evaluated more often than we allow it to, then we try to merge the state that we are about to construct with some state that was already present in the graph. In our example, we would start with Node 6. Further evaluation would lead to a new node where one again reaches the instruction "`while (x < 1000)`", but this time with the value `x == -9`. If we only allow instructions to be evaluated at most once, then this new node would not be permitted, but instead one would try to merge Node 6 with this new node. Merging the information "`x == -10`" of Node 6 with the information "`x == -9`" of the new node gives the more general information "`x` : $\mathbb{Z}$". In this way, we obtain Node 7.

By evaluation, refinement, and merging we always construct a finite graph representing every possible computation originating from the start state. Because of side effects on the heap, we inline the code of called methods.[2] This means that in the resulting graph there is no difference between a method invocation and any other instruction. Thus, we integrate all possible side effects from any called method.

3 Termination Graphs for Programs Operating on the Heap

As shown in Sect. 2, for primitive data types, states contain information about the ranges of values (e.g., "`x == -10`", "`x` $\in [-\infty, 999]$", or "`x` : $\mathbb{Z}$"). For object instances, states contain information about the existence (Sect. 3.1), type (Sect. 3.2), equality (Sect. 3.3), and the structure of instances (Sect. 3.4).

3.1 Existence

Sometimes, it is only important to know whether a referenced instance exists. Consider the algorithm "`while (x.next != null) {x = x.next;}`", where `x` is a variable of a user-defined class type `List`. In the initial state, *we do not know if the referenced instance* `x.next` *exists*. To express this, we use the *annotation* "`x.next` : ∃?". To evaluate the condition "`x.next != null`", we use a refinement step to generate two states (one where "`x.next == null`" and one where we use the annotation "`x.next` : `List`" to express that the instance `x.next` indeed exists and is an instance of the class `List`).

3.2 Type

Due to class extensions, we may encounter situations where the same referenced instance exists in two

[2]To handle recursion, we need a special treatment based on an abstraction of the call stack.

states, but the type of the instance differs in the two states. When merging these two states, we create a state where only information up to the common superclass is kept (in the worst case this is the class `Object` which is the superclass of all classes) and information about the subclasses is abstracted away. If we now need to know whether the instance implements a subtype of the common superclass currently represented, we branch according to the type hierarchy until the information needed for evaluation is represented explicitly. Here it is important to also regard the case where the instance is no instance of any subclass, e.g., just an object created by "`new Object()`".

3.3 Equality

Because of aliasing, it is important to know which variables reference the same instance. A state with "`x == y`" may be merged with "`x != y`". Then the merged state must represent both possibilities. We therefore use an annotation "*may be equal*" to denote these two possibilities.

3.4 Structural Information

We also use annotations to describe information about the structure of an instance. When abstracting away concrete instances, it is often beneficial to keep information about their structure, e.g., to keep the information that the structure was acyclic. This has advantages when performing refinement steps (since one does not have to perform refinement steps that would violate acyclicity) and it has advantages when generating TRSs out of termination graphs afterwards. The reason is that variables in a TRS can only represent instances that are guaranteed to be acyclic. More precisely, we use two annotations denoting that an instance is "*sharing*" (e.g., a DAG) or "*cyclic*". If none of these two annotations is present, then the instance has the form of a tree.

Furthermore, we use an annotation to describe that two instances are not equal, but one is a sub-structure of the other. Consider an instance `x` referencing a tree, where one element of that tree is identical to some other instance `y` on the heap, depicted in Fig. 2. In this situation we add an annotation "`x` *may join instance* `y`" which is considered when performing refinement steps. This joining information is a binary relation between two abstract instances, whereas the "*sharing*" information above is a unary relation that describes the structure defined by a single instance (it joins itself).

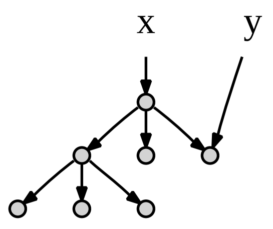


Figure 2: `x` joining `y`

4 From Termination Graphs to TRSs

We encode the strongly connected components of the termination graph into separate TRSs[3] that have to be proved terminating. For every path between two "instance" edges we introduce a corresponding rewrite rule. The information gained by "refine" edges is propagated to the term in the left-hand side of the rewrite rule. The rules are built using a unique function symbol for each node at the end of an "instance" edge. The arguments of this function symbol are the terms representing all values of the variables in the corresponding state. Every condition encountered on a path (resulting from `if`- or `while`-statements) is translated to a condition for the rule corresponding to the path. For example, the following ITRS results from the right termination graph in Fig. 1. Its termination is easy to show [4].

$$\mathsf{f}(x) \to \mathsf{f}(x+1) \quad | \quad x < 1000$$

To represent instances, we use a distinct constructor symbol C for every class `C`. If the class `C` has n

[3] In order to handle pre-defined data structures like integers, we use conditional *integer TRSs* (ITRSs) where operations like $<$, $+$, etc. are built in, cf. [4].

non-static fields, then the function symbol C has n arguments for the values of these fields. Unknown parts of the instance are represented by variables, provided that the structure of the instance is acyclic (and therefore finite). All problems related to aliasing and sharing are already handled during the construction of the termination graph. Thus, they do not need special considerations in the TRS anymore. If `C` is not a `final` class (i.e., it may have subclasses), then the function symbol C gets an additional last argument which is used to represent instances of subclasses of `C`. So if `D` is a `final` subclass of `C` and `C` has a non-static field "`int x`" and `D` has a non-static field "`int y`", then the instance `d` of type `D` with `d.x == 4` and `d.y == 3` would be represented by the term $\mathsf{C}(4,\mathsf{D}(3))$.

As another example, let the `final` class `Tree` have 3 non-static fields: `Tree left; Tree right; int value`. The instance of class `Tree` with no left or right subtree and `value 5` is represented by the term $\mathsf{O}(\mathsf{T}(\mathsf{null},\mathsf{null},5))$, where O and T are the function symbols for the classes `Object` and `Tree`.

The following ITRS is the result of transforming a program that flattens a list of trees (first argument of `f`) into the list of the values contained in the tree nodes (second argument of `f`). Here, the initial list is assumed to be acyclic and non-sharing (i.e., the list of trees can be represented as a finite tree structure). In addition to O and T, the symbols TL and L represent the `final` classes `TreeList` and `List`. The first non-static field in the class `TreeList` is a reference to the next element of the list (of type `TreeList`) and the second field in `TreeList` is the value of the first element in the list (of type `Tree`). Similarly, the class `List` implements lists with `int` values.

$$\begin{aligned}
\mathsf{f}(\mathsf{O}(\mathsf{TL}(x_1,\mathsf{null})),x_2) &\to \mathsf{f}(x_1,x_2)\\
\mathsf{f}(\mathsf{O}(\mathsf{TL}(x_1,\mathsf{O}(\mathsf{T}(\mathsf{null},\mathsf{null},x_6)))),x_2) &\to \mathsf{f}(x_1,\mathsf{O}(\mathsf{L}(x_2,x_6)))\\
\mathsf{f}(\mathsf{O}(\mathsf{TL}(x_1,\mathsf{O}(\mathsf{T}(\mathsf{null},\mathsf{O}(x_5),x_6)))),x_2) &\to \mathsf{f}(\mathsf{O}(\mathsf{TL}(x_1,\mathsf{O}(x_5))),\mathsf{O}(\mathsf{L}(x_2,x_6)))\\
\mathsf{f}(\mathsf{O}(\mathsf{TL}(x_1,\mathsf{O}(\mathsf{T}(\mathsf{O}(x_4),\mathsf{null},x_6)))),x_2) &\to \mathsf{f}(\mathsf{O}(\mathsf{TL}(x_1,\mathsf{O}(x_4))),\mathsf{O}(\mathsf{L}(x_2,x_6)))\\
\mathsf{f}(\mathsf{O}(\mathsf{TL}(x_1,\mathsf{O}(\mathsf{T}(\mathsf{O}(x_4),\mathsf{O}(x_3),x_6)))),x_2) &\to \mathsf{f}(\mathsf{O}(\mathsf{TL}(\mathsf{O}(\mathsf{TL}(x_1,\mathsf{O}(x_4))),\mathsf{O}(x_3))),\mathsf{O}(\mathsf{L}(x_2,x_6)))
\end{aligned}$$

This example shows the advantages of rewrite techniques for algorithms on user-defined data structures. As object instances are simply represented as terms, TRS techniques (that are specialized on comparing terms) can easily prove termination of this TRS. We are currently implementing our approach in the termination tool AProVE. Afterwards, we will perform extensive experiments to evaluate our technique and to compare it with related work, e.g., [1, 2, 3, 8].

References

[1] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination analysis of Java Bytecode. In *Proc. FMOODS'08*, LNCS 5051, pp. 2-18, 2008.

[2] M. Colón and H. B. Sipma. Practical methods for proving program termination. In *Proc. CAV'02*, LNCS 2034, pp. 442-454, 2002.

[3] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proc. PLDI'06*, ACM Press, pp. 415-426, 2006.

[4] C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving termination of integer term rewriting. In *Proc. RTA'09*, LNCS, 2009. To appear.

[5] J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated termination analysis for Haskell: From term rewriting to programming languages. In *Proc. RTA'06*, LNCS 4098, pp. 297-312, 2006.

[6] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*, Prentice Hall, 2005.

[7] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Prentice Hall, 1999.

[8] F. Spoto, F. Mesnard, and E. Payet. A termination analyser for Java Bytecode based on path-length. `http://profs.sci.univr.it/~spoto/termination09.pdf`, 2009.

Proving termination by invariance relations

Paolo Pilozzi[*]
Dept. of Computer Science
K.U.Leuven, Belgium
paolo.pilozzi@cs.kuleuven.be

Danny De Schreye
Dept. of Computer Science
K.U.Leuven, Belgium
danny.deschreye@cs.kuleuven.be

Abstract

We propose a new constraint-based approach to termination analysis, applicable to Logic Programming (LP) and Constraint Handling Rules (CHR). Our approach further extends the existing constraint-based approaches for LP based on polynomial interpretations and introduces a whole new level of expressivity. We can handle problems such as bounded increase and integer arithmetic, where arguments are not necessarily bounded by below, elegantly. Furthermore, we are able to prove termination of programs that only terminate for subsets of the considered queries. Examples are algorithms that manipulate graphs and that only terminate if the graph in the input is cycle-free. This information cannot be represented using the existing techniques in termination analysis. Therefore, we introduce invariance relations, representing relations among terms that hold on atoms during calls to the program. These relations can also be derived in a constraint-based manner and they can be used as a basis for a more expressive interpretation of the atoms of the program. We discuss our technique in the context of CHR, solving an important class of open problems containing transitivity rules. We also demonstrate the technique in an LP context and show that it is more powerful than existing constraint-based approaches.

The following CHR program [3,7], computes the transitive closure of a graph.

$$transitivity \ @ \ arc(X,Y), arc(Y,Z) \Rightarrow arc(X,Z).$$

This program is representative for a large class of practical programs in CHR that cannot be handled using any existing automated technique. For it to be proven terminating, the query graph may not contain cycles. Consider for example a query of two constraints, $arc(a,b)$ and $arc(b,c)$, representing a cycle-free graph, consisting of three nodes: a, b and c. The transitivity rule is applicable to these constraints, adding the constraint $arc(a,c)$. Since a fire-once policy prevents multiple applications of a propagation rule on the same combination of constraints, the program terminates. If on the other hand, the query graph contains a cycle, then arcs are added progressively along that cycle, ad infinitum.

To tackle the problem of programs that only terminate for some subset of the considered queries, we present in the next paragraphs a new approach to constraint-based termination analysis. The technique is based on polynomial interpretations [5], however, we allow more expressive polynomials than the ones used in [5]. Our approach is applicable to both LP and CHR programs. First, we discuss our approach for CHR programs containing a transitivity rule. Then, we show the technique's applicability to LP and argue that it is more powerful than existing approaches.

To prove termination automatically, we develop verifiable sufficient conditions [2,6]. To express such conditions, we require information on possible calls to a program. Such information comes in the form of a *call set*. In LP, this corresponds to the set of atoms selected in some derivation of the program, for some query. In CHR, it corresponds to the atoms used to fire a rule in some computation of the program for some query. To specify the intended use of a program, we use a *query set*, finitely represented using query patterns. From it, we select atoms to compose queries. Then, by application of abstract interpretation, we derive from query patterns, call patterns [4,7] to represent the call set.

[*]Supported by I.W.T. Flanders - Belgium

In [5], termination is proven using a polynomial interpretation. There, the interpretation maps atoms and terms to $\mathbb{N}$-closed polynomials, using a *polynomial level mapping*, $|.|$, for atoms and a *polynomial norm*, $\|.\|$, for terms. The call set has to be rigid w.r.t. this interpretation, meaning that only polynomials can be constructed using the instantiated parts of atoms and terms. To verify this, syntactic conditions are constructed, based on call patterns [1].

In [5], the $\mathbb{N}$-closed polynomials that are used, are constructed from positive monomials. For example, a polynomial $a_0 + a_1.X_1 + \cdots + a_n.X_n$. However, for programs such as our running example, we require negative monomials to prove termination. We allow therefore polynomials that are the difference of two (linear) positive polynomials: $a_0^1 + a_1^1.X_1 + \cdots + a_n^1.X_n - (a_0^2 + a_1^2.X_1 + \cdots + a_n^2.X_n)$. For such polynomials, $\mathbb{N}$-closedness is not trivial: there is obviously a positive integer assignment to variables and coefficients, resulting in a negative integer outcome. To guarantee that such polynomials are $\mathbb{N}$-closed, $a_0^1 + a_1^1.X_1 + \cdots + a_n^1.X_n \geq a_0^2 + a_1^2.X_1 + \cdots + a_n^2.X_n$ must hold during the execution of the program. In the next paragraph, we derive this information, using *invariance relations*.

To prove termination of a CHR program containing a transitivity rule, we need to be able to represent more restricted kinds of queries. Call patterns [4] provide information on the instantiated parts of atoms in the call set, however, do not represent relations that hold among these parts. To represent this information, we introduce invariance relations, holding on atoms in the query or the call set.

To prove termination of our example, we must guarantee the use of a cycle-free query graph. In such a graph, one can always find an ordering on nodes, such that every arc points from a strictly smaller sized node to a bigger sized node. Thus, the queries that we can use are in $\{\bigwedge_{i=1,\dots,n} arc(t_i', t_i'') \mid \forall i : t_i'' \succ t_i'\}$.

Using an invariance relation $I_{arc/2}^{query} = \{(t_1,t_2) \mid t_1,t_2 \in Term_P, t_{max} \succ t_2 \succ t_1 \succ t_{min}\}$, we can formulate cycle-freeness on the level of individual query atoms. The constants, t_{max} and t_{min}, fix the domain of terms that can be used as nodes. The ordering on terms in arcs is expressed on individual atoms. In contrast to the invariant formulated on the query, the latter invariance relation therefore represents a relation holding on atoms that can be used in a query, while the former invariant represents actual queries.

Given the invariance relation $I_{arc/2}^{query}$ on query atoms, we aim to derive an invariance relation $I_{arc/2}^{callset}$ on the atoms used in calls. For that purpose, we will now assume that $I_{arc/2}^{query}$ is not just an invariance relation, but actually specifies the query atoms of interest. Since we also assume that the query atoms are a subset of the call atoms, the invariance relation $I_{arc/2}^{callset}$ on the call atoms must hold on the query atoms. Thus, $I_{arc/2}^{query} \subseteq I_{arc/2}^{callset}$. More explicitly, we get the condition $\forall t_1,t_2 \in Term_P : (t_1,t_2) \in I_{arc/2}^{query} \rightarrow (t_1,t_2) \in I_{arc/2}^{callset}$, meaning that the invariance relation on query atoms must be more restrictive than the one on atoms in the call set. On the basis of CHR rules, we derive rule conditions. For the transitivity rule, we get that $\forall t_1,t_2,t_3 \in Term_P : (t_1,t_2) \in I_{arc/2}^{callset} \wedge (t_2,t_3) \in I_{arc/2}^{callset} \rightarrow (t_1,t_3) \in I_{arc/2}^{callset}$, meaning that the invariance relation holding on $arc/2$ atoms, has to remain valid under transitivity.

We derive invariance relations on call atoms in a constraint-based manner, using a polynomial interpretation. That is, we reformulate the conditions by using a symbolic polynomial form for invariance relations in the same way as this is done for interargument relations in [5]. The invariance relation in polynomial form for the query is $I_{arc/2}^{query} = \{(t_1,t_2) \mid \|t_{max}\| > \|t_2\| \wedge \|t_2\| > \|t_1\| \wedge \|t_1\| > \|t_{min}\|\}$.

For the invariance relation on the atoms used in calls, we provide the same level of expressivity as for the query. Thus for the $arc/2$ constraints, we express their invariance relation using three symbolic polynomial inequalities: $I_{arc/2}^{callset} = \{(t_1,t_2) \mid ii_0^1 + ii_1^1.\|t_1\| + ii_2^1.\|t_2\| \geq io_0^1 + io_1^1.\|t_1\| + io_2^1.\|t_2\| \wedge ii_0^2 + ii_1^2.\|t_1\| + ii_2^2.\|t_2\| \geq io_0^2 + io_1^2.\|t_1\| + io_2^2.\|t_2\| \wedge ii_0^3 + ii_1^3.\|t_1\| + ii_2^3.\|t_2\| \geq io_0^3 + io_1^3.\|t_1\| + io_2^3.\|t_2\|\}$. To find values for the symbolic coefficients, we formulate the invariance conditions in terms of these symbolic polynomials. Thus, $\forall t_1,t_2 \in Term_P :$

$$\|t_{max}\| > \|t_2\| \wedge \|t_2\| > \|t_1\| \wedge \|t_1\| > \|t_{min}\| \rightarrow ii_0^1 + ii_1^1.\|t_1\| + ii_2^1.\|t_2\| \geq io_0^1 + io_1^1.\|t_1\| + io_2^1.\|t_2\| \wedge ii_0^2 + ii_1^2.\|t_1\| + ii_2^2.\|t_2\| \geq \\ io_0^2 + io_1^2.\|t_1\| + io_2^2.\|t_2\| \wedge ii_0^3 + ii_1^3.\|t_1\| + ii_2^3.\|t_2\| \geq io_0^3 + io_1^3.\|t_1\| + io_2^3.\|t_2\|$$

For the condition related to the transitivity rule, we get that $\forall t_1, t_2, t_3 \in Term_P$:

$$ii_0^1 + ii_1^1.\|t_1\| + ii_2^1.\|t_2\| \geq io_0^1 + io_1^1.\|t_1\| + io_2^1.\|t_2\| \wedge ii_0^2 + ii_1^2.\|t_1\| + ii_2^2.\|t_2\| \geq io_0^2 + io_1^2.\|t_1\| + io_2^2.\|t_2\| \wedge ii_0^3 + ii_1^3.\|t_1\| + ii_2^3.\|t_2\| \geq \\ io_0^3 + io_1^3.\|t_1\| + io_2^3.\|t_2\| \wedge ii_0^1 + ii_1^1.\|t_2\| + ii_2^1.\|t_3\| \geq io_0^1 + io_1^1.\|t_2\| + io_2^1.\|t_3\| \wedge ii_0^2 + ii_1^2.\|t_2\| + ii_2^2.\|t_3\| \geq \\ io_0^2 + io_1^2.\|t_2\| + io_2^2.\|t_3\| \wedge ii_0^3 + ii_1^3.\|t_2\| + ii_2^3.\|t_3\| \geq io_0^3 + io_1^3.\|t_2\| + io_2^3.\|t_3\| \rightarrow ii_0^1 + ii_1^1.\|t_1\| + ii_2^1.\|t_3\| \geq \\ io_0^1 + io_1^1.\|t_1\| + io_2^1.\|t_3\| \wedge ii_0^2 + ii_1^2.\|t_1\| + ii_2^2.\|t_3\| \geq io_0^2 + io_1^2.\|t_1\| + io_2^2.\|t_3\| \wedge ii_0^3 + ii_1^3.\|t_1\| + ii_2^3.\|t_3\| \geq io_0^3 + io_1^3.\|t_1\| + io_2^3.\|t_3\|$$

These conditions are similar to the ones obtained in [5] and thus can be solved in a similar way. That is, by transforming them to a system of Diophantine constraints on the symbolic coefficients and solving the resulting system by an existing constraint solver. One possible solution results in an invariance relation,

$$I_{arc/2}^{callset} = \{(t_1, t_2) \mid \|t_{max}\| + 0.\|t_1\| + 0.\|t_2\| \geq 0 + 0.\|t_1\| + 1.\|t_2\| \wedge 0 + 0.\|t_1\| + 1.\|t_2\| \geq \\ 0 + 1.\|t_1\| + 0.\|t_2\| \wedge 0 + 1.\|t_1\| + 0.\|t_2\| \geq \|t_{min}\| + 0.\|t_1\| + 0.\|t_2\|\}$$

Thus, the invariance relation on the query is preserved under transitivity. That is, the bounds, t_{max} and t_{min}, remain to be bounds for the added arcs and these arcs keep pointing from strictly smaller sized nodes to bigger sized nodes.

To prove the transitivity program terminating, we obtain the following termination conditions, according to [6]. The scope of the decrease conditions is restricted by invariance relations holding on the atoms used in the head.

$$\forall t_1, t_2, t_3 : (t_1, t_2) \in I_{arc/2}^{callset} \wedge (t_2, t_3) \in I_{arc/2}^{callset} \rightarrow arc(t_1, t_2) \succ arc(t_1, t_3)$$
$$\forall t_1, t_2, t_3 : (t_1, t_2) \in I_{arc/2}^{callset} \wedge (t_2, t_3) \in I_{arc/2}^{callset} \rightarrow arc(t_2, t_3) \succ arc(t_1, t_3)$$

To prove validity of these conditions, we require an interpretation mapping $arc/2$ atoms to a polynomial of the form $|arc(t_1, t_2)| = (\|t_{max}\| - \|t_{min}\|) - (\|t_2\| - \|t_1\|)$. Since, $(\|t_{max}\| - \|t_{min}\|) - (\|t_2\| - \|t_1\|) \geq 0$ is implied by $I_{arc/2}^{callset}$, we map all $arc/2$ atoms in the call set, to $\mathbb{N}$-closed polynomials.

To obtain an integrated approach, incorporating invariance and termination conditions, we introduce a symbolic form for level mappings, parameterizable by invariance relations. For $arc/2$ atoms, we have $|arc(t_1, t_2)| = ii_0 + ii_1.\|t_1\| + ii_2.\|t_2\| - (io_0 + io_1.\|t_1\| + io_2.\|t_2\|)$ as its symbolic form, which depends on the invariance relation holding on $arc/2$ atoms in the call set. Thus, $\forall t_1, t_2 \in Term_P$:

$$ii_0^1 + ii_1^1.\|t_1\| + ii_2^1.\|t_2\| \geq io_0^1 + io_1^1.\|t_1\| + io_2^1.\|t_2\| \wedge ii_0^2 + ii_1^2.\|t_1\| + ii_2^2.\|t_2\| \geq io_0^2 + io_1^2.\|t_1\| + io_2^2.\|t_2\| \wedge ii_0^3 + ii_1^3.\|t_1\| + ii_2^3.\|t_2\| \geq \\ io_0^3 + io_1^3.\|t_1\| + io_2^3.\|t_2\| \rightarrow ii_0 + ii_1.\|t_1\| + ii_2.\|t_3\| \geq io_0 + io_1.\|t_1\| + io_2.\|t_3\|$$

Notice that the symbolic form, used in existing constraint-based approaches [5], is implied by default. After all, $ii_0 + ii_1.\|t_1\| + ii_2.\|t_2\| \geq 0$ is trivially implied, since only polynomials are constructed with positive integer coefficients. Therefore, if termination can be proven by existing systems, we can prove it as well.

As termination conditions, we obtain now:

$$\forall t_1, t_2, t_3 : (t_1, t_2) \in I_{arc/2}^{callset} \wedge (t_2, t_3) \in I_{arc/2}^{callset} \rightarrow ii_0 + ii_1.\|t_1\| + ii_2.\|t_2\| - (io_0 + io_1.\|t_1\| + io_2.\|t_2\|) \geq \\ ii_0 + ii_1.\|t_1\| + ii_2.\|t_3\| - (io_0 + io_1.\|t_1\| + io_2.\|t_3\|)$$
$$\forall t_1, t_2, t_3 : (t_1, t_2) \in I_{arc/2}^{callset} \wedge (t_2, t_3) \in I_{arc/2}^{callset} \rightarrow ii_0 + ii_1.\|t_2\| + ii_2.\|t_3\| - (io_0 + io_1.\|t_2\| + io_2.\|t_3\|) \geq \\ ii_0 + ii_1.\|t_1\| + ii_2.\|t_3\| - (io_0 + io_1.\|t_1\| + io_2.\|t_3\|)$$

Solving these conditions using the techniques in [5], provides us with the aforementioned interpretation for $arc/2$ constraints, proving termination.

Our technique is also applicable to LP. Similarly, we derive invariance relations for call atoms on the basis of invariance relations for query atoms. We demonstrate this on the following LP program.

$$a(Max,Max). \qquad a(N,Max)\ :-\ neq(N,Max),a(s(N),Max).$$

Currently, no technique for termination analysis of LP can handle such programs. Since a call to $neq/2$ also succeeds whenever the first term of an $a/2$ atom is bigger than the second, the program will run forever, as for such queries the first argument has no upper bound. Thus, in order to prove termination, we have to consider that a query consists of $a(t_1,t_2)$ atoms, where $t_1 \prec t_2$, represented using an invariance relation, $I^{query}_{a/2} = \{(t_1,t_2) \mid t_1,t_2 \in Term_P, \|t_2\| > \|t_1\|\}$.

Consequently, to derive $I^{callset}_{a/2}$, we initially set it to a general symbolic form, $\{(t_1,t_2) \mid t_1,t_2 \in Term_P \wedge ii^1_0 + ii^1_1.\|t_1\| + ii^1_2.\|t_2\| \geq io^1_0 + io^1_1.\|t_1\| + io^1_2.\|t_2\|\}$ and then, we formulate invariance conditions. To express $I^{query}_{a/2} \subseteq I^{callset}_{a/2}$, we get that $\forall t_1,t_2 \in Term_P : \|t_2\| > \|t_1\| \to ii^1_0 + ii^1_1.\|t_1\| + ii^1_2.\|t_2\| \geq io^1_0 + io^1_1.\|t_1\| + io^1_2.\|t_2\|$. For the clause, we obtain that $\forall t_1,t_2 \in Term_P : ii^1_0 + ii^1_1.\|t_1\| + ii^1_2.\|t_2\| \geq io^1_0 + io^1_1.\|t_1\| + io^1_2.\|t_2\| \wedge (\|t_1\| > \|t_2\| \vee \|t_2\| > \|t_1\|) \to ii^1_0 + ii^1_1.\|s(t_1)\| + ii^1_2.\|t_2\| \geq io^1_0 + io^1_1.\|s(t_1)\| + io^1_2.\|t_2\|$. Here, $R_{neq/2} = \{(t_1,t_2) \mid \|t_1\| > \|t_2\| \vee \|t_2\| > \|t_1\|\}$ estimates the effect of a call to $neq/2$. Then, a suitable polynomial level mapping for $a/2$ atoms is derived on the basis of $I^{callset}_{a/2}$. That is, $\forall t_1,t_2 \in Term_P :$ $ii^1_0 + ii^1_1.\|t_1\| + ii^1_2.\|t_2\| \geq io^1_0 + io^1_1.\|t_1\| + io^1_2.\|t_2\| \to ii_0 + ii_1.\|t_1\| + ii_2.\|t_2\| \geq io_0 + io_1.\|t_1\| + io_2.\|t_2\|$.

To prove termination, we must show that all recursive body atoms are smaller than the atom used in the head [2]. This results, for the LP program, in the following condition: $\forall t_1,t_2 \in Term_P : ii^1_0 + ii^1_1.\|t_1\| + ii^1_2.\|t_2\| \geq io^1_0 + io^1_1.\|t_1\| + io^1_2.\|t_2\| \wedge (\|t_1\| > \|t_2\| \vee \|t_2\| > \|t_1\|) \to ii_0 + ii_1.\|t_1\| + ii_2.\|t_2\| - (io_0 + io_1.\|t_1\| + io_2.\|t_2\|) \geq ii_0 + ii_1.\|s(t_1)\| + ii_2.\|t_2\| - (io_0 + io_1.\|s(t_1)\| + io_2.\|t_2\|)$. When solving the termination, invariance and interargument conditions, we obtain $|a(t_1,t_2)| = \|t_2\| - \|t_1\|$ as one possible level mapping for $a/2$ atoms, proving termination.

To conclude, we proposed a new constraint-based approach, more powerful than existing approaches in LP. Our approach can handle both LP and CHR programs containing bounded increase, integer arithmetic or programs that only terminate for subsets of considered queries. We overcame these problems by introducing invariance relations, resulting in a useful specification for relations holding among the instantiated parts of call atoms and thus complementing call types. As such, we are able to represent more restricted kinds of queries, related to the use of domains or even global properties of queries, such as cycle-freeness of a query graph. In the continuation of this work, we will formalize this technique and further evaluate it, in the context of CHR and Logic Programs.

References

[1] A. Bossi, N. Cocco, and M. Fabris. Proving termination of logic programs by exploiting term properties. *TAPSOFT91, p.153–180 (1991).*

[2] D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *JLP, vol.19–20, p.199–260 (1994).*

[3] T. Frühwirth. Theory and practice of constraint handling rules. *JLP, vol.37(1–3), p.95–138 (1998).*

[4] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *JLP, vol.13(2–3), p.205–258 (1992).*

[5] M.T. Nguyen and De Schreye. Polynomial interpretations as a basis for termination analysis of logic programs. In *ICLP 2005, p.311–325 (2005).*

[6] P. Pilozzi and D. De Schreye. Termination analysis of chr revisited. In *ICLP08, p.501–515 (2008).*

[7] T. Schrijvers. Analyses, optimizations and extensions of constraint handling rules: Ph.d. summary. In *ICLP05, p.435–436 (2005).*

On Some Implementation Aspects of VMTL

Felix Schernhammer, Bernhard Gramlich
TU Wien, Austria, {`felixs,gramlich`}`@logic.at`

1 Introduction

VMTL (Vienna Modular Termination Laboratory [12]) is a program designed to (semi-) automatically prove termination of various classes of term rewriting systems (TRSs). In particular, conditional and context-sensitive TRSs (CTRSs and CSRSs) are supported. Based on the well-known dependency pair framework of [1, 7, 4], VMTL is extensible by new dependency pair processors and transformations (from (C)TRSs/CSRSs to (C)TRSs/CSRSs). An important design goal of VMTL is to make extension as easy as possible. Moreover, VMTL provides implementations of several standard and new dependency pair processors as well as of transformations (from CTRSs to CSRSs).

In this work we briefly discuss some implementation details of the DP processors of VMTL, focussing in particular on the non-standard ones. The processors currently available in VMTL are a *dependency graph processor*, used for decomposition of dependency pair problems (based on [1, Section 4.1]), a *subterm-criterion* processor, based on [9], a *reduction pair processor using recursive path orderings with status* implemented using an external SAT solver, based on [13], a *reduction pair processor using polynomial orderings* with coefficients from $\mathbb{N}$ and constants from $\mathbb{Z}$, based on [6], *narrowing and instantiation processors* based on [7, 11] and a *size change principle processor*, based on [14].

The inclusion of the (compared to the other ones) more “exotic” size-change principle processor is a tribute to the commitment of VMTL to specialize to a certain extent on the termination analysis of conditional TRSs.

The size-change principle processor is able to trace (the size of) argument terms over several function calls not necessarily comparing arguments at the same positions. This may be advantageous in the setting of TRSs obtained by transforming CTRSs, since there argument terms of auxiliary function symbols change positions frequently.

In the following we discuss some of these processors and their particular implementation in some detail. We assume familiarity with the basic notions and notations of term rewriting, context-sensitive rewriting and the dependency pair framework (cf. e.g. [5, 10, 7]).

2 Dependency Pair Processors of VMTL

2.1 Dependency Graph Processor

For the computation of the estimated dependency graph we basically use the estimation defined in [1, Section 4.1]. This estimation relies on the functions cap^{μ} and ren^{μ} that map terms to terms. $cap^{\mu}(s)$ replaces maximal active subterms of the term s, rooted by defined symbols with fresh variables. $ren^{\mu}(s)$ replaces each active variable of s by a fresh one.

It turned out that for the termination analysis of rewrite systems with a particular class of so-called *forbidden patterns* (cf. [8]) we obtain systems with rules of the form $C[l] \to C[r]$ through transformations. These rules motivate the definition of a modified notion of constructors and defined symbols, respectively.

Definition 1 (Quasi-defined symbols)**.** *Let $\mathscr{R} = (\Sigma, R)$ be TRS. For each rule $l \to r$ (where $l \neq r$) the maximal context C is identified, such that $l = C[s]$ and $r = C[t]$ (note that C might be empty) and $root(s) \in \Sigma$. Then $root(s)$ is a* quasi-defined symbol.

We denote the set of quasi-defined symbols of a TRS (R,Σ) by D^q and the corresponding set of quasi-constructors (i.e., $\Sigma \setminus D^q$) as C^q. Observe that quasi-defined function symbols and quasi-constructors are incomparable with the usual notions of defined functions and constructors.[1] Like ordinary constructors, our modified constructors have the important property that the "top" part of a term - if it is constructed only from such constructors - cannot be modified by reduction, i.e., whenever $s = C[s_1,\dots,s_n]$ and C consists only of functions from C^q and variables, then for all terms t with $s \to^* t$ we have $t = C[t_1,\dots,t_n]$ for some terms $t_1,\dots,t_n$.

Now, we define $\widetilde{cap}^{\mu}$ to replace all maximal active subterms rooted by functions from D^q by fresh variables.

Note that the estimated dependency graph obtained by using $\widetilde{cap}^{\mu}$ is in general not comparable with the one obtained by using cap^{μ} (for the reason mentioned above). However, in practice the intersection of both can be used.

Example 1. *Consider the following (sub-)TRS from [8, Example 4]:*

$$\begin{array}{rr} 2nd(inf(x)) \to 2nd(x : inf(s(x))) & 2nd(x : (y : zs)) \to y \\ s(inf(x)) \to s(x : inf(s(x))) & inf(inf(x)) \to inf(x : inf(s(x))) \\ inf(x) : y \to (x : inf(s(x))) : y & (x' : inf(x)) : y \to (x' : (x : inf(s(x)))) : y \end{array}$$

As this system is context-free, we use a replacement map μ with $\mu(f) = \{1,\dots ar(f)\}$ for all $f \in \Sigma$. Here, $: \in C^q$, while $:$ is not a constructor (in the traditional sense). Thus, consider for instance the rule (which corresponds to a dependency pair) $inf(x) : y \to (x : inf(s(x))) : y$ (i.e., $C[inf(x)] \to C[x : inf(s(x))]$)). Then $\widetilde{cap}^{\mu}(ren^{\mu}((x : inf(s(x))) : y)) = (x : z) : y$ which is not unifiable with $inf(x) : y$. Hence, there is no arc in the estimated dependency graph from this dependency pair to itself. Yet, such an arc would be present if the original cap^{μ} function were used instead of $\widetilde{cap}^{\mu}$.

2.2 Subterm Criterion Processor

The subterm-criterion processor is based on [9] and adapted for context-sensitive dependency pair problems by substituting $\rhd$ by $\rhd_{\mu}$. Then, the correctness of the processor is shown analogously to the proof of Theorem 11 in [9], exploiting that $\rhd_{\mu}$ is closed under substitutions and $\rhd_{\mu} \circ \to_{\mu} \subseteq \to_{\mu} \circ \rhd_{\mu}$.

Note that this generalization of the subterm-criterion processor has already been proposed in [2].

2.3 Narrowing Processor

VMTL's narrowing processors are based on results from [11, Section 6]. There, given a (context-sensitive) dependency pair problem $(P = \{s \to t\} \uplus P', R, \mu)$, and provided that

(1) t does not unify with the left-hand side of any other dependency pair from P,

(2) t is linear, and

(3) there are no migrating variables in $s \to t$, i.e., there are no variables which occur active in t and non-active in s,

the (forward) narrowing processor yields a new DP problem $(\overline{P}, R, \mu)$ where $\overline{P} = (P - \{s \to t\}) \cup \{s_k\theta_k \to t_k\}$ such that t_k are all the context-sensitive one-step narrowings of t_k and θ_k are the corresponding mgu's.

In fact, in some cases the set of the narrowings that need to be included in the resulting DP problem can be restricted further (see [11] for more details). Note that Condition (3) is not mentioned in [3, Theorem 5.3]. However, it is essential as the following example shows.

[1] In $\{g(f(x)) \to g(f(a))\}$, f is quasi-defined, but not defined, and g is defined, but not quasi-defined.

Example 2. *Consider the CSRS*

$$t(f(x)) \to t(h(x)) \qquad t(i(b)) \to t(f(a)) \qquad a \to b \qquad h(x) \to i(x)$$

with $\mu(t) = \mu(h) = \{1\}$ and $\mu(i) = \mu(f) = \emptyset$. The context-sensitive dependency pairs (according to the approach of [1] also used in VMTL) are

$$\begin{array}{lll} T(f(x)) \to T(h(x)) & T(i(b)) \to T(f(a)) & T(f(x)) \to H(x) \\ U(a) \to A & T(f(x)) \to U(x) & \end{array}$$

One might be tempted to narrow $T(h(x))$ to $T(i(x))$ and thus replace the first dependency pair by $T(f(x)) \to T(i(x)))$. However, while the initial DP problem is infinite (and the CSRS indeed μ-non-terminating), the transformed problem is finite, thus the transformation is unsound.

We suggest another slight variation of the narrowing processor defined above. Preconditions (1), (2) and (3) can be replaced by

(1') $ren(t)$ does not unify with the left-hand side of any dependency pair from P, and

(2') for all possible narrowing steps of t, i.e., with $t|_p\theta = l\theta$ for some rule $l \to r$, we have that t_p is linear, l matches t_p and $l \to r$ does not contain migrating variables.

This formulation has the particular advantage that it may also be applicable to non-right-linear dependency pairs.

Example 3. *Consider the example*

$$from(x) \to cons(x, cons(nil, from(s(x)))) \qquad cons(s(x), xs) \to nil$$

taken from the TPDB outermost *section. The corresponding DP problem contains a dependency pair $FROM(x) \to CONS(x, cons(nil, from(s(x))))$. According to our modified definition (one can easily see that the (linearized) right-hand side of this rule does not unify with any left-hand side of any dependency pair) the rule can be narrowed to $FROM(x) \to CONS(x, cons(nil, cons(s(x), cons(nil, from(s(s(x)))))))$. This gain in structure may be useful in a subsequent (outermost) termination analysis.*

2.4 Instantiation Processor

In VMTL we use a modified version of the instantiation processors proposed in [7], because the use of our processors turned out to be fruitful in the termination analysis of CSRSs obtained by transforming CTRSs. In particular, we do not only allow propagation of constructors, but also of defined symbols under certain circumstances (cf. [11, Definitions 12 and 13] for details).

Given a DP problem $(P = \{s \to t\} \uplus P', R, \mu)$ and provided that

- all subterms of s containing variables are rooted by constructor symbols,
- $s \to t$ does not contain migrating variables and
- all $l \to r \in P$ where $cap^{\mu}(ren^{\mu}(r))$ unifies with s (this subset of P is denoted by $Pred_{s\to t}(P)$), satisfy that $s\sigma = r$ for some substitution σ,

the (backward) instantiation processor yields a new dependency pair problem $(P' \cup \{s\sigma \to t\sigma \mid l \to r \in Pred_{s\to t}(P) \wedge r = s\sigma\}, R, \mu)$.

Example 4. *Consider a DP Problem given by*

$$P = \left\{ \begin{array}{lcl} F(x) & \to & G(x) \\ G(b) & \to & F(a) \end{array} \right. \qquad R = \left\{ \begin{array}{lcl} f(x) & \to & g(x) \\ g(b) & \to & f(a) \\ a & \to & b \end{array} \right.$$

and $\mu(f) = \{1,\ldots ar(f)\}$ for all $f \in \Sigma$. $Pred_{F(x)\to G(a)}(P)$ is $\{G(b) \to F(a)\}$ and $F(x)$ matches $F(a)$. Hence, the instantiation processor yields a new DP problem $(\{F(a) \to G(a), G(b) \to F(a)\}, R, \mu)$.

See e.g., [11, Examples 10, 11 and 12] for more complex examples where the application of the instantiation processor is shown.

3 Summary and Conclusion

We have discussed some details and particularities of the dependency pair processors in VMTL. Note that at the time of writing not all proposed features are already present in the publicly available version of VMTL. A more comprehensive presentation of the VMTL system itself can be found in [12]. VMTL is available as stand-alone command-line executable (as a java jar file) and through a webinterface at the VMTL homepage (`http://www.logic.at/vmtl/`). There, also more details regarding extensibility and benchmarks of VMTL on the set of TRSs from the TPDB in various categories can be found.

References

[1] B. Alarcón, F. Emmes, C. Fuhs, J. Giesl, R. Gutiérrez, S. Lucas, P. Schneider-Kamp, and R. Thiemann. Improving context-sensitive dependency pairs. In: I. Cervesato, H. Veith, and A. Voronkov, eds., Proc. LPAR'08, Doha, Qatar, November 22–27, 2008, LNCS 5330, pp. 636–651. Springer, Nov. 2008.

[2] B. Alarcón, R. Gutiérrez, and S. Lucas. Context-sensitive dependency pairs. In: S. Arun-Kumar and N. Garg, eds., Proc. (FST&TCS'06), Kolkata, India, Dec. 13–15, 2006, LNCS 4337, pp. 297–308, Springer, 2006.

[3] B. Alarcón, R. Gutiérrez, and S. Lucas. Improving the Context-Sensitive Dependency Graph. *Electronic Notes in Theoretical Computer Science*, 188:91–103, 2007.

[4] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

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

[6] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In: Proc. SAT'07, LNCS 4501, pp. 340-354. Springer, 2007.

[7] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

[8] B. Gramlich and F. Schernhammer. Extending context-sensitivity in term rewriting. Draft version (submitted for publication), available at `http://www.logic.at/people/schernhammer/`, 2009.

[9] N. Hirokawa and A. Middeldorp. Dependency pairs revisited. In: V. van Oostrom, ed., Proc. RTA'04, Aachen, Germany, June 3 – June 5, 2004, LNCS 3091, pp. 249-268, Springer, June 2004.

[10] S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1), Jan. 1998. The MIT Press.

[11] F. Schernhammer and B. Gramlich. Characterizing and proving operational termination of deterministic conditional term rewriting systems. Technical Report E1852-2009-01, TU Wien, Austria, Mar. 2009.

[12] F. Schernhammer and B. Gramlich. VMTL – A modular termination laboratory. In: R. Treinen, ed., Proc. RTA'09, Brasilia, Brazil, June 29 – July 1, 2009, LNCS, Springer, June 2009. To appear.

[13] P. Schneider-Kamp, R. Thiemann, E. Annov, M. Codish, and J. Giesl. Proving termination using recursive path orders and SAT solving. In: Proc. FroCoS'07, LNCS 4720, pp. 267–282, Springer, 2007.

[14] R. Thiemann and J. Giesl. Size-change termination for term rewriting. In: R. Nieuwenhuis, ed., RTA'03, Valencia, Spain, June 9 – June 11, 2003, LNCS 2706, pp. 264-278, Springer, June 2003.

Proving Termination for Logic Programs with Cut[*]

Peter Schneider-Kamp (petersk@imada.sdu.dk), University of Southern Denmark, Denmark
Jürgen Giesl (giesl@informatik.rwth-aachen.de), RWTH Aachen University, Germany
Alexander Serebrenik (a.serebrenik@tue.nl), TU Eindhoven, The Netherlands
Thomas Ströder (thomas.stroeder@rwth-aachen.de), RWTH Aachen University, Germany
René Thiemann (rene.thiemann@uibk.ac.at), University of Innsbruck, Austria

1 Introduction

Termination is an important and well-studied property for logic programs. Almost all approaches for automated termination analysis focus on *definite* logic programs (see e.g. [1, 5, 6] for some recent work), whereas real-world Prolog programs typically use the **cut** operator (!). Moreover, the cut allows to link two notions of termination found in the literature: existential (finite failure or first answer after a finite number of derivation steps, cf. [4]) and universal (all answers found and no infinite derivation afterwards).

The cut is often used to exploit the order of the clauses. By adding a cut as the first part of the body for a clause c, all following clauses can assume that the head of c does not unify with the selected atom of the query. Consider the following program $\mathscr{P}$, demonstrating this particular kind of use of the cut. But our method is also successful for programs with more subtle uses of the cut, e.g., programs containing clauses like "$\mathsf{not}(X) \leftarrow X, !, \mathsf{fail}$" for *negation as failure* or "$\mathsf{if}(A,B,C) \leftarrow A, !, C$" for *if then else*.

$$\mathsf{div}(X,0,Z) \leftarrow\ !, \mathsf{fail}. \quad (1) \qquad \mathsf{minus}(0,Y,0). \quad (4)$$

$$\mathsf{div}(0,Y,Z) \leftarrow\ !, =(Z,0). \quad (2) \qquad \mathsf{minus}(X,0,X). \quad (5)$$

$$\mathsf{div}(X,Y,\mathsf{s}(Z)) \leftarrow \mathsf{minus}(X,Y,U), \quad (3) \qquad \mathsf{minus}(\mathsf{s}(X),\mathsf{s}(Y),Z) \leftarrow \mathsf{minus}(X,Y,Z). \quad (6)$$

$$\mathsf{div}(U,Y,Z). \qquad\qquad =(X,X). \quad (7)$$

We regard the queries $\mathscr{Q} = \{\mathsf{div}(t_1,t_2,t_3) \mid t_1,t_2 \text{ are ground}\}$. Any termination analyzer that ignores the cut fails as $\mathsf{div}(0,0,Z)$ would lead to the subtraction of $\mathsf{0}$ using Rule (3) and start an infinite derivation.

To analyze termination of $\mathscr{Q}$ w.r.t. $\mathscr{P}$ we present a new method that automatically transforms $\mathscr{P}$ to a new cut-free logic program $\mathscr{P}'$ such that termination of $\mathscr{Q}$ w.r.t. $\mathscr{P}'$ implies termination of $\mathscr{Q}$ w.r.t. $\mathscr{P}$. This approach was inspired by a related method for termination analysis of Haskell programs [3]. In Section 2 we develop a set of inference rules that characterize the behavior of logic programs with cut for *concrete* queries. In Section 3, we extend these rules to handle (potentially infinite) classes of queries represented by abstract queries. With these rules we can automatically build so-called *termination graphs*.[1] In Section 4 we show how to extract cut-free logic programs from these termination graphs.

2 Concrete Derivations

To analyze termination of logic programs with cut, we denote the current state of the computation by a stack of goals (to represent backtrack information). Moreover, a goal can be labeled by the clause that has to be applied next. In addition, our stack of goals also contains explicit marks for the scope of a cut. This allows us to express the non-local effect of the cut by a local rule.

More precisely, a *concrete state* is a finite list of three different types of elements (where list elements are separated by "|"). An element can be a goal q, i.e., a finite sequence of terms. An element can also be a labeled goal q^i_m with $i,m \in \mathbb{N}$. This means that we must apply the i-th program clause to q next. The number m is used to label the cuts that will be introduced when applying the i-th program clause to q. Finally, an element of a state can also be $?_m$ for $m \in \mathbb{N}$. This denotes the end of the scope of a cut labeled with m. So when reaching a labeled cut $!_m$, all elements in the state that precede $?_m$ will be discarded.

For example, a state consisting of just a single goal is $\mathsf{div}(0,0,Z)$. In order to make the backtracking

[*]Supported by the DFG (Deutsche Forschungsgemeinschaft) under grant GI 274/5-2.

[1]Similar graphs have also been studied in program optimization [8].

possibilities explicit, we model the resolution step with the two inference rules CASE and EVAL. CASE determines which clauses can be applied, labels the current goal by the index of the first such clause and adds appropriately labeled copies of the current goal as backtracking possibilities. Moreover, $?_m$ is added to the end of the list of new backtracking goals in order to denote the scope of cuts introduced. So our state would be transformed to the state $\mathsf{div}(0,0,Z)^1_1 \mid \mathsf{div}(0,0,Z)^2_1 \mid \mathsf{div}(0,0,Z)^3_1 \mid ?_1$.

If the first atom of a labeled goal unifies with the head of the corresponding clause, we apply the EVAL-rule. Otherwise, we apply BACKTRACK. So in our example, the EVAL-rule uses program clause (1) for the first goal $\mathsf{div}(0,0,Z)^1_1$ which yields $!_1, \mathsf{fail} \mid \mathsf{div}(0,0,Z)^2_1 \mid \mathsf{div}(0,0,Z)^3_1 \mid ?_1$. Due to the cut $!_1$, all alternatives up to $?_1$ are discarded. So the CUT-rule yields the state with the only goal fail.

To summarize, we express program derivations by the following inference rules.[2] Here, the unlabeled term t must neither be a ! nor a variable, the list of terms q may be $\square$ (then "$t, \square$" is identified with t), and S and S' are concrete states. One can show that for any (infinite) derivation starting in a cut-free goal q, there is a corresponding (infinite) derivation from the initial state q using these inference rules.

$$\frac{\square \mid S}{S}\ (\textsc{Success}) \qquad \frac{?_m \mid S}{S}\ (\textsc{Failure}) \qquad \frac{!_m, q \mid S \mid ?_m \mid S'}{q \mid ?_m \mid S'}\ (\textsc{Cut}) \quad \text{where } S \text{ contains no } ?_m$$

$$\frac{t, q \mid S}{(t,q)^{i_1}_m \mid \ldots \mid (t,q)^{i_k}_m \mid ?_m \mid S}\ (\textsc{Case}) \quad \text{where } m \text{ is fresh, } i_1 < \ldots < i_k, \text{ and } \{c_{i_1}, \ldots, c_{i_k}\} \text{ are all clauses for the predicate of } t$$

$$\frac{(t,q)^i_m \mid S}{B'\sigma, q\sigma \mid S}\ (\textsc{Eval}) \quad \text{where } c_i = H \leftarrow B,\ mgu(t,H) = \sigma, \text{ and } B' = B[!\ /\ !_m]. \qquad \frac{(t,q)^i_m \mid S}{S}\ (\textsc{Backtrack}) \quad \text{where } c_i = H \leftarrow B \text{ and } t \text{ does not unify with } H.$$

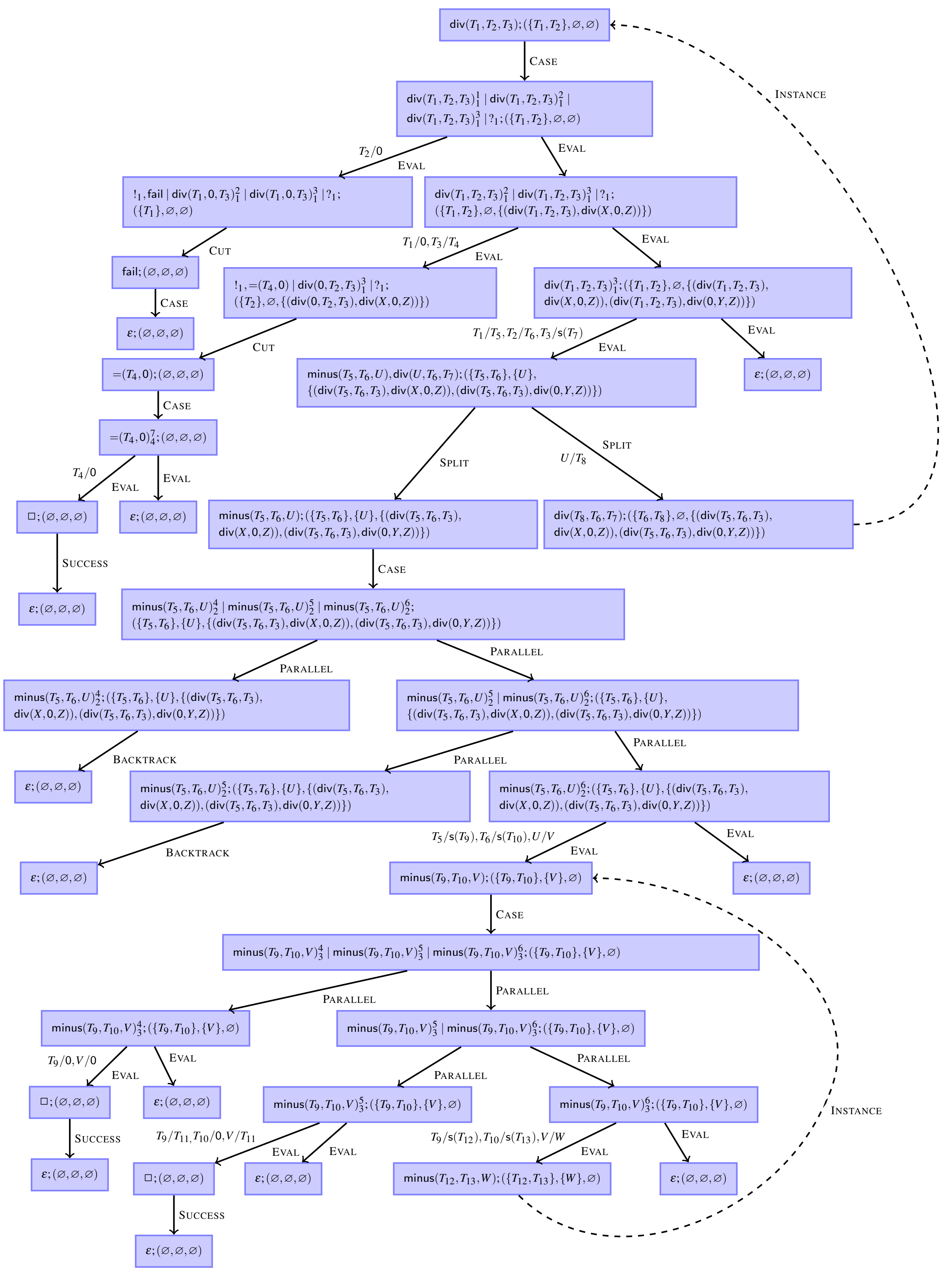
3 Abstract Derivations

To represent *sets* of queries we use *abstract terms*, i.e., terms containing both abstract variables and object variables that correspond to variables in logic programming. We also modify the inference rules introduced above in order to be applicable to abstract terms as well. However, the application of these inference rules to abstract terms would typically result in an infinite derivation (since the abstract variables could be instantiated over and over again, leading to an infinite sequence of CASE- and EVAL-rules). Hence, our goal is to construct a finite representation of potentially infinite derivations as a *termination graph*. To this end, we use three additional inference rules.

The INSTANCE-rule is applicable whenever the current state is an instance of a previous state. Still, INSTANCE is often not enough to obtain a finite graph since due to the backtracking lists, there are many states that are not an instance of any previous state. Therefore, we further introduce the PARALLEL-rule that allows us to split a backtracking list into separate problems. While this rule may lose precision, it is virtually always needed for obtaining a finite graph. In addition, we also need a SPLIT-rule which splits a goal of more than one term into a single-term goal (left successor) and the remainder (right successor). For example, this rule is needed for programs like "$\mathsf{p}(\mathsf{s}(X)) \leftarrow \mathsf{p}(X), \mathsf{q}$" to ensure that we eventually reach a state that is an instance of a previous state.

With these rules, we obtain the following termination graph for the program $\mathscr{P}$ above and the set of queries $\mathscr{Q} = \{\mathsf{div}(t_1,t_2,t_3) \mid t_1, t_2 \text{ are ground}\}$. Its nodes contain both the current state and (after a semicolon) a triple with information on the terms that the abstract variables $T_1, T_2, \ldots$ can represent. (The first component of this triple are those abstract variables that can only be instantiated by ground terms, the second component contains those object variables that may not occur in the terms the abstract variables are instantiated with, and the third component contains pairs of terms that may not unify.)

[2]To handle all possible cases, one needs additional CUT-rules, which we omitted here for the sake of readability.



4 Termination Graph to Logic Program

In order to obtain a cut-free logic program we construct one clause for each "interesting" path in the termination graph. We say that a path is "interesting" if it starts at the topmost node, at the successor of an INSTANCE-node[3] or at a left successor of a SPLIT-node; ends at a SUCCESS-node, at an INSTANCE-node, or at a left successor of a SPLIT-node; and does not traverse an INSTANCE-edge or left successors of SPLIT-nodes (the latter possibility is only allowed if this left successor is the end node).

By constructing clauses from the "interesting" paths of the termination graph, we obtain the following program where we have to show termination w.r.t. the queries $\mathscr{Q} = \{\mathsf{div}_1(t_1,t_2,t_3) \mid t_1,t_2 \text{ are ground}\}$.

$$\begin{array}{rcl}
\mathsf{div}_1(0,T_2,0). & & \\
\mathsf{div}_1(T_5,T_6,\mathsf{s}(T_7)) & \leftarrow & \mathsf{minus}_1(T_5,T_6,T_8), \\
 & & \mathsf{div}_1(T_8,T_6,T_7). \\
\mathsf{div}_1(T_5,T_6,\mathsf{s}(T_7)) & \leftarrow & \mathsf{minus}_1(T_5,T_6,U).
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{minus}_1(\mathsf{s}(0),\mathsf{s}(T_{10}),0). & & \\
\mathsf{minus}_1(\mathsf{s}(T_{11}),\mathsf{s}(0),T_{11}). & & \\
\mathsf{minus}_1(\mathsf{s}(\mathsf{s}(T_{12})),\mathsf{s}(\mathsf{s}(T_{13})),W) & \leftarrow & \mathsf{minus}_2(T_{12},T_{13},W). \\
\mathsf{minus}_2(0,T_{10},0). & & \\
\mathsf{minus}_2(T_{11},0,T_{11}). & & \\
\mathsf{minus}_2(\mathsf{s}(T_{12}),\mathsf{s}(T_{13}),W) & \leftarrow & \mathsf{minus}_2(T_{12},T_{13},W).
\end{array}$$

The above logic program can easily be proved terminating automatically, e.g., by the techniques of [6, 7]. In fact, virtually all methods for proving termination of logic programs will succeed to prove termination of this definite logic program.

5 Conclusions

We have introduced a novel non-termination preserving pre-processing method to eliminate the effect of cuts in many practically relevant cases. After this pre-processing, any technique for proving universal termination of logic programming can be applied.

This pre-processing has been implemented in our tool AProVE [2] and the implementation has been tested successfully on examples where the cut is needed to ensure termination.

References

[1] M. Codish, V. Lagoon, P. Schachte, and P. J. Stuckey. Size-Change Termination Analysis in *k*-Bits. In *ESOP '06*, volume 3924 of *LNCS*, pages 230–245, 2006.

[2] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the DP Framework. In *IJCAR '06*, volume 4130 of *LNAI*, pages 281–286, 2006.

[3] J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages. In *RTA '06*, volume 4098 of *LNCS*, pages 297–312, 2006.

[4] M. Marchiori. Proving Existential Termination of Normal Logic Programs. In *AMAST '96*, volume 1101 of *LNCS*, pages 375–390, 1996.

[5] M. T. Nguyen and D. De Schreye. Polytool: Proving Termination Automatically Based on Polynomial Interpretations. In *LOPSTR '06*, volume 4407 of *LNCS*, pages 210–218, 2007.

[6] M. T. Nguyen, J. Giesl, P. Schneider-Kamp, and D. De Schreye. Termination Analysis of Logic Programs based on Dependency Graphs. In *LOPSTR '07*, volume 4915 of *LNCS*, pages 8–22, 2008.

[7] P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated Termination Proofs for Logic Programs by Term Rewriting. *ACM Transactions on Computational Logic*. To appear.

[8] M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In *ILPS '95*, pages 465–479. MIT Press, 1995.

[3]An INSTANCE-node is a node to which the INSTANCE-rule is applied. SUCCESS-nodes and SPLIT-nodes are defined analogously. An INSTANCE-edge is an edge from an INSTANCE-node to its successor.

Checking the Influence of Non-Termination on Free Theorems[*]

Daniel Seidel[†] and Janis Voigtländer
Technische Universität Dresden, 01062 Dresden, Germany
{seideld,voigt}@tcs.inf.tu-dresden.de

Abstract

Free theorems are a helpful tool for validating program transformations, based only on types. Also in other areas they are useful proof utilities. General recursion and hence the possibility of endless looping reduces the strength of free theorems by forcing additional restrictions. These are, dependent on the type, sometimes dispensable. We present two algorithms, one claimed to disprove the need of the restrictions and one verifying their need, dependent on a given input type.

1 Introduction

For a function $f :: [a] \to [a]$, given in Haskell syntax, a "naive" free theorem [5] tells us that $map\ g\ (f\ xs) = f\ (map\ g\ xs)$ holds for all functions $g :: \tau_1 \to \tau_2$, lists $xs :: [\tau_1]$, and arbitrary types τ_1, τ_2, where *map* is the well-known function applying its first argument to each element of a list. Intuitively we can argue that f cannot distinguish concrete elements in the list and thus works only on the list's structure. On the other hand, g is mapped into the list and thus can only act on the elements. Especially for functions like *reverse* this seems to be obvious.

As long as we stay in the pure polymorphic lambda calculus with data types, the theorem will hold. But if we allow general recursion, *fix id* (short $\perp$) with $id = \lambda x \to x$ is a non-terminating program with undefined value. It exists for any type τ and destroys our ideal world. Consider the above free theorem with $f = \lambda x \to [\perp]$, $\tau_1 = \tau_2 = \mathsf{Int}$, and $g = \lambda x \to 14$. Then for $xs = []$ it states $[14] = [\perp]$. Hence, we have to rule out this and similar examples. The necessary requirement is to force g being *strict*, meaning $g\ \perp = \perp$.

Regarding the equation $f\ xs = f\ (map\ g\ xs)$ obtained from the free theorem for $f :: [a] \to \mathsf{Int}$, we can freely choose g without any restriction and will not find a way to violate the equation. The reason is that g cannot introduce a "harmful" $\perp$ somewhere, for f does not care about the elements of the list. But how to determine in general if some non-strict function causes any harm?

Based on a refined type system tracking the use of *fix* we have developed an algorithm searching for terms that break the "naive" free theorems and extended a slightly altered version to produce whole counterexamples. The second version is implemented and available online[1].

2 A Calculus for Tracking the Use of *fix*

As setting for the investigations we take a polymorphic lambda calculus that provides the types $\tau ::= \alpha \mid \tau \to \tau \mid \mathsf{Int} \mid \mathsf{Bool} \mid [\tau] \mid (\tau,\tau) \mid \mathsf{Either}\ \tau\ \tau \mid ()$ with α ranging over type variables, and terms $t ::= x \mid \lambda x :: \tau.t \mid t\ t \mid n \mid \mathbf{True} \mid \mathbf{False} \mid []_\tau \mid t : t \mid (t,t) \mid \mathsf{Left}_\tau\ t \mid \mathsf{Right}_\tau\ t \mid () \mid \mathbf{fix}\ t \mid \mathbf{case}\ t\ \mathbf{of}\ \{\cdots\}$ with x ranging over term variables and n over the integers. Case-statements are present for integers, Booleans, lists, pairs, disjoint sum types, and the unit type. There is no type abstraction since we only consider rank-1 polymorphism, like in Haskell 98.

The denotational type and term semantics are entirely standard with types interpreted as complete partial orders with $\perp$ as least element in each type. They are denoted by $[\![\cdot]\!]$. Note that for functions we identify the constant function to $\perp$ and $\perp$ itself, while for all other compound types we introduce a fresh $\perp$ as least element. Term semantics corresponds to lazy evaluation.

[*]A fuller exposition of the work reported on in this extended abstract has been submitted to CSL'09.

[†]This author was supported by the DFG under grant VO 1512/1-1.

[1]http://linux.tcs.inf.tu-dresden.de/~seideld/cgi-bin/exfind.cgi

Before we consider the typing rules, a closer look at the nature of free theorems is necessary. The key is the parametricity theorem stated by Reynolds [4]. It roughly says that parametrically polymorphic functions act independently of their instantiations, and therefore independent of concrete values only present for *some* types. This is formalized by a certain relation, called *logical relation*, built up inductively for each type from relations assigned to the type variables. It has to satisfy a *fundamental property*, stating that each closed term is related to itself by the logical relation of its type. As an application of the parametricity theorem, free theorems arise [5].

By the introduction of general recursion, using the fixpoint primitive **fix** t, with the term semantics $[\![\textbf{fix}\,t]\!] = \bigsqcup_{n\geq 0} ([\![t]\!])^n \perp$ we also introduce **fix** id_α (short $\perp_\alpha$) with value $\perp$ for *any* instantiation of the type variable α, i.e. for any environment. Hence, by the fundamental property, the logical relation of α has to relate $\perp$ to itself for *any* type, i.e. it has to be *strict*. Recalling the introductory examples, that strictness condition is exactly the strictness restriction on g, for g represents the logical relation at α specialized to a function.

But, as $\perp$ can only be introduced by **fix**, the strictness condition is not necessary if **fix** is not used. Hence, we can often relax restrictions. This is done by a refined typing system [3] that distinguishes between type variables with a not necessarily strict logical relation and the ones inducing a strictness condition. Further we use a type class Pointed, stating how strictness propagates through the type structure. The two main rules are

$$\frac{\alpha^* \in \Gamma}{\Gamma \vdash \alpha \in \mathsf{Pointed}} \qquad \text{and} \qquad \frac{\Gamma \vdash \tau_2 \in \mathsf{Pointed}}{\Gamma \vdash \tau_1 \to \tau_2 \in \mathsf{Pointed}}$$

where the context Γ contains type variables, whereof the ones that can only be mapped to a strict logical relation are marked by *. Hence, one rule states that only type variables marked by * in the typing environment force strictness and the second states that strictness of a function type depends only on its result type. All unmentioned rules are just axioms, declaring each other type to be in Pointed.

Now we define the system of typing rules in a straightforward way, only with pointedness constraints added to

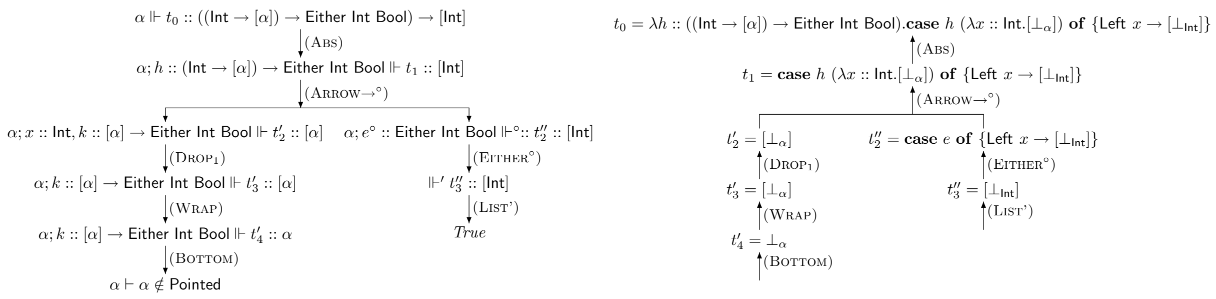
$$\frac{\Gamma \vdash \tau \in \mathsf{Pointed} \qquad \Gamma;\Sigma \vdash t :: \tau \to \tau}{\Gamma;\Sigma \vdash \textbf{fix}\,t :: \tau\,.}\ (\text{FIX'})$$

and the rules introducing case-statements, since **case** $\perp_\tau$ **of** $\{\ldots\}$ evaluates to $\perp$. The context Σ contains all term variables (with type annotations) that occur free in t. The calculus is called **PolyFix***.

3 A Glance at the Term Search Algorithm

The original question, asking if the free theorem for a given type τ requires strictness conditions can be reduced to a question of (differential) type inhabitation in **PolyFix***. Assume we have some type variable α, a type τ, and a type variable context Γ not containing α. We can ask if there exists some term t such that $\alpha^*,\Gamma;\emptyset \vdash t :: \tau$ holds, but $\alpha,\Gamma;\emptyset \vdash t :: \tau$ does not. Unfortunately, it is not quite that easy, as there are many ways to insert a harmless **fix**. For example, take **case** $[\perp_\alpha]$ **of** $\{[] \to 10;\ x : \_ \to 5\}$. Here the $\perp_\alpha$ in the case scrutinee has no influence at all (due to lazy evaluation). Hence, we have to first search for sources of disrelation, like $\perp_\tau$ with τ not in Pointed, and secondly guarantee that the disrelation will be forwarded such that the final term is not in the logical relation of the input type.

With **TermFind** we describe an algorithm doing exactly that. It starts with a type and a type context (with marked/unmarked type variables) and then tries to generate a term using **fix** on an unpointed type and propagating the resulting disrelation. Surely, the algorithm has to correspond to the typing rules of **PolyFix*** and use them backwards to derive a term. But unfortunately, the original rules are not straightforwardly applicable for the algorithm. For example, the rule for term application, (APP), with the premises $\Gamma;\Sigma \vdash t_1 :: \tau_1 \to \tau_2$ and $\Gamma;\Sigma \vdash t_2 :: \tau_1$ and the conclusion $\Gamma;\Sigma \vdash (t_1\ t_2) :: \tau_2$, lacks any kind of subformula property. Here, τ_1 is only present in the premise and backwards application is not possible



$\alpha \Vdash t_0 :: ((\mathsf{Int} \to [\alpha]) \to \mathsf{Either\ Int\ Bool}) \to [\mathsf{Int}]$
(Abs)
$\alpha; h :: (\mathsf{Int} \to [\alpha]) \to \mathsf{Either\ Int\ Bool} \Vdash t_1 :: [\mathsf{Int}]$
(Arrow→°)
$\alpha; x :: \mathsf{Int}, k :: [\alpha] \to \mathsf{Either\ Int\ Bool} \Vdash t'_2 :: [\alpha]$
$\alpha; e^\circ :: \mathsf{Either\ Int\ Bool} \Vdash^\circ :: t''_2 :: [\mathsf{Int}]$
(Drop₁)
(Either°)
$\alpha; k :: [\alpha] \to \mathsf{Either\ Int\ Bool} \Vdash t'_3 :: [\alpha]$
$\Vdash' t''_3 :: [\mathsf{Int}]$
(Wrap)
(List')
$\alpha; k :: [\alpha] \to \mathsf{Either\ Int\ Bool} \Vdash t'_4 :: \alpha$
True
(Bottom)
$\alpha \vdash \alpha \notin \mathsf{Pointed}$
$t_0 = \lambda h :: ((\mathsf{Int} \to [\alpha]) \to \mathsf{Either\ Int\ Bool}).\mathbf{case}\ h\ (\lambda x :: \mathsf{Int}.[\bot_\alpha])\ \mathbf{of}\ \{\mathsf{Left}\ x \to [\bot_{\mathsf{Int}}]\}$
(Abs)
$t_1 = \mathbf{case}\ h\ (\lambda x :: \mathsf{Int}.[\bot_\alpha])\ \mathbf{of}\ \{\mathsf{Left}\ x \to [\bot_{\mathsf{Int}}]\}$
(Arrow→°)
$t'_2 = [\bot_\alpha]$
$t''_2 = \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{Left}\ x \to [\bot_{\mathsf{Int}}]\}$
(Drop₁)
(Either°)
$t'_3 = [\bot_\alpha]$
$t''_3 = [\bot_{\mathsf{Int}}]$
(Wrap)
(List')
$t'_4 = \bot_\alpha$
(Bottom)

Figure 1: **TermFind** run for an example

since τ_1 is unknown. Inspired by a system for proof search in intuitionistic propositional logic [2], extended in [1], we get a suitable transformed rule system, equivalent with respect to type inhabitation to the original set of rules. The transformed system is adapted to our needs by allowing the introduction of **fix**, only on unpointed types. The design principle of **TermFind** is to insert "harmful", meaning "the-free-theorem-affecting", $\bot$ by the use of **fix** on unpointed types thus that the arising disrelation can be propagated. Since there are many rules, we concentrate only on the ones needed to find a term suitable to verify the strictness condition for a somewhat challenging example.

Given a function $f :: ((\mathsf{Int} \to [\alpha]) \to \mathsf{Either\ Int\ Bool}) \to [\mathsf{Int}]$, the corresponding free theorem tells us that for all types τ_1, τ_2, functions $g :: \tau_1 \to \tau_2$ strict and $p :: (\mathsf{Int} \to [\tau_1]) \to \mathsf{Either\ Int\ Bool}$, $q :: (\mathsf{Int} \to [\tau_2]) \to \mathsf{Either\ Int\ Bool}$ such that $p\ r = q\ s$ for all $r :: \mathsf{Int} \to [\tau_1]$, $s :: \mathsf{Int} \to [\tau_2]$ with $(map\ g) \circ r = s$, we have $f\ p = f\ q$. The question is: Does the theorem hold for g non-strict? Let us try to find a counterexample following the ideas of **TermFind**. The derivation is given in Figure 1. In the following explanation we refer to the variable names in the figure. The utilized rules are

$$\frac{\Gamma \vdash \tau \notin \mathsf{Pointed}}{\Gamma;\Sigma \Vdash \bot_\tau :: \tau}\ (\textsc{Bottom}) \qquad \frac{\Gamma;\Sigma, x :: \tau_1 \Vdash t :: \tau_2}{\Gamma;\Sigma \Vdash (\lambda x :: \tau_1.t) :: \tau_1 \to \tau_2}\ (\textsc{Abs}) \qquad \frac{\Gamma;\Sigma \Vdash t :: \tau}{\Gamma;\Sigma, x :: () \Vdash t :: \tau}\ (\textsc{Drop}_1)$$

$$\frac{\Gamma;\Sigma, x :: \tau_1, g :: \tau_2 \to \tau_3 \Vdash t_1 :: \tau_2 \qquad \Gamma;\Sigma, y^\circ :: \tau_3 \Vdash^\circ t_2 :: \tau}{\Gamma;\Sigma, f :: (\tau_1 \to \tau_2) \to \tau_3 \Vdash t_2[(f\ (\lambda x :: \tau_1.t_1[(\lambda z :: \tau_2.f(\lambda u :: \tau_1.z))/g]))/y] :: \tau}\ (\textsc{Arrow}{\to}^\circ)$$

$$\frac{\Gamma;\Sigma \Vdash t :: \tau}{\Gamma;\Sigma \Vdash (t : []_\tau) :: [\tau]}\ (\textsc{Wrap}) \qquad \frac{\Sigma \Vdash' t :: \tau}{\Gamma;\Sigma, e^\circ :: \mathsf{Either}\ \tau_1\ \tau_2 \Vdash^\circ \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{Left}\ x \to t\} :: \tau}\ (\textsc{Either}^\circ)$$

$$\Sigma \Vdash' [\bot_\tau] :: [\tau]\ (\textsc{List'})$$

Surely, the easiest way to provoke a term being typable in **PolyFix*** only if α is *-annotated is to use **fix** on a type that is only in Pointed if α is *-annotated. So (Bottom) is a natural rule in **TermFind**. But since $[\mathsf{Int}]$ and thus $((\mathsf{Int} \to [\alpha]) \to \mathsf{Either\ Int\ Bool}) \to [\mathsf{Int}]$ are in Pointed independently of α, we have to search further and decompose the type via (Abs). Now, the right-hand side on its own seems not promising for a counterexample, but the term variable context can be explored. We could apply h somehow. The rule (Arrow→°)[2], part of the replacement for (App), brings us further. Referring to the variable names in (Arrow→°), the first premise tries to find a replacement for y introducing a "harmful" $\bot$. The other premise ensures only that y is really used in t_2. That is done by a second rule system of **TermFind**, adjusted due to the changed goal. Note that the "must use" for y is signaled by a $^\circ$-mark.

But first we concentrate on the first premise. In the example we have $x :: \mathsf{Int}$ in Σ. That is for sure unnecessary, since when needing an integer we can simply use a constant. Hence we drop x from the context by (Drop$_1$). Now we go on deconstructing the right-hand side type by (Wrap), which allows us to access the element type of lists. Finally, we can use (Bottom) and introduce a "harmful" $\bot$.

The other derivation branch arising from the application of (Arrow→°) heads in the concrete example for a use of e in t''_2. Since there is a case-statement for **Either**, we can try to introduce e that way, using (Either°). The disrelation will arise since $[\![e]\!] = \bot$ in one environment (and hence the whole

[2]The rule corresponds to (L→→) in [1].

expression) and $[\![e]\!] \neq \bot$ in another. The only remaining task is to provide a term for the body of the case-statement. To preserve the disrelation it has to evaluate to something different from $\bot$. Here a third system, $\Vdash'$, comes in. It basically consists of axioms providing terms evaluating to an at least partially defined value. For list types the appropriate axiom is (LIST'). Note that the type variable context is not necessary here.

We produced a complete derivation tree for a term of type $((\mathsf{Int} \to [\alpha]) \to \mathsf{Either\ Int\ Bool}) \to [\mathsf{Int}]$. Walking backwards this derivation tree, composing the final term as shown in the right part of Figure 1, we end up with $f = t_0 = \lambda h :: ((\mathsf{Int} \to [\alpha]) \to \mathsf{Either\ Int\ Bool}).\mathbf{case}\ h\ (\lambda x :: \mathsf{Int}.[\bot_\alpha])\ \mathbf{of}\ \{\mathsf{Left}\ x \to [\bot_{\mathsf{Int}}]\}$, and indeed that term is suitable to generate a counterexample! Take $\tau_1 = \tau_2 = ()$, $g = \lambda x :: ().()$, $p = \lambda r :: (\mathsf{Int} \to [()]).\mathbf{case}\ r\ 0\ \mathbf{of}\ \{[x] \to \mathsf{Left}_{\mathsf{Bool}}\ 0\}$, and $q = \lambda s :: (\mathsf{Int} \to [()]).\mathbf{case}\ s\ 0\ \mathbf{of}\ \{[x] \to \mathbf{case}\ x\ \mathbf{of}\ \{() \to \mathsf{Left}_{\mathsf{Bool}}\ 0\}\}$.

As well as the term, the complete counterexample is not handmade. We have altered **TermFind** to keep track of the origin of the disrelation and additionally to generate all ingredients for the complete counterexample. The altered calculus is called **ExFind** and an implementation is provided via a web interface at `http://linux.tcs.inf.tu-dresden.de/~seideld/cgi-bin/exfind.cgi`.

Unfortunately, there is a small gap between the terms found by **TermFind** and **ExFind** because **ExFind** uses a simplified version of (ARROW$\to^\circ$), omitting g in the second premise and hence the double use of f in the conclusion. Also, pointedness conditions are added and smooth merging of the term variable contexts of the premises is guaranteed by an additional check. As a consequence of sticking to singleton lists, **ExFind** sometimes returns no counterexample even though there is one. In these cases **TermFind** finds a term unsuitable for counterexample generation, but very similar to a suitable one.

4 The Results Summed Up

Inspired by [3] we set up a calculus tracking the use of **fix** by constraints on type variables. With the ideas of [2] and [1] we altered the typing rules to gain a weak subformula property and developed the algorithm **TermFind**. It terminates for every input and is claimed to be complete (because designed by a worst case strategy), meaning strictness conditions in a free theorem are superfluous if **TermFind** terminates without returning a term. **TermFind** is not sound for counterexample generation, due to problems with the (ARROW$\to^\circ$) rule. These problems are solved in a second algorithm, **ExFind**, which is proved to be sound, but incomplete. It also terminates for every input and produces whole counterexamples (if possible). **ExFind** is implemented and available online at `http://linux.tcs.inf.tu-dresden.de/~seideld/cgi-bin/exfind.cgi`.

References

[1] P. Corbineau. First-order reasoning in the calculus of inductive constructions. In *TYPES 2003, Proc.*, volume 3085 of *LNCS*, pages 162–177. Springer-Verlag, 2004.

[2] R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57(3):795–807, 1992.

[3] J. Launchbury and R. Paterson. Parametricity and unboxing with unpointed types. In *ESOP, Proc.*, volume 1058 of *LNCS*, pages 204–218. Springer-Verlag, 1996.

[4] J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proc.*, pages 513–523. Elsevier, 1983.

[5] P. Wadler. Theorems for free! In *FPCA, Proc.*, pages 347–359. ACM Press, 1989.

CeTA – A Tool for Certified Termination Analysis*

Christian Sternagel René Thiemann Sarah Winkler Harald Zankl
University of Innsbruck, Austria

1 Motivation

Since the first termination competition[1] in 2004 it is of great interest, whether a proof—that has been automatically generated by a termination tool—is indeed correct. The increasing number of termination proving techniques as well as the increasing complexity of generated proofs (e.g., combinations of several techniques, exhaustive labelings, tree automata, etc.), make certifying (i.e., checking the correctness of) such proofs more and more tedious for humans. Hence the interest in automated certification of termination proofs. This led to the general approach of using proof assistants (like Coq [2] and Isabelle [12]) for certification. At the time of this writing, we are aware of the two combinations Coccinelle/C*i*ME [4, 5] and CoLoR/Rainbow [3]. Here Coccinelle and CoLoR are Coq libraries, formalizing rewriting theory. Then C*i*ME as well as Rainbow are used to transform XML proof trees into Coq proofs which heavily rely on those libraries. Hence if you want to certify a proof you need a termination tool that produces appropriate XML output, a converter (C*i*ME or Rainbow), a local Coq installation, and the appropriate library (Coccinelle or CoLoR).

In this paper we present the latest developments for the new combination IsaFoR/CeTA [13] (version 1.03). Note that the system design has two major differences in comparison to the two existing ones. Firstly, our library IsaFoR (*Isabelle Formalization of Rewriting*) is written for the theorem prover Isabelle/HOL and not for Coq. Secondly, and more important, instead of generating for each proof tree a new proof, using an auxiliary tool, our library contains several executable check-functions. Here, *executable* means that it is possible to automatically obtain a functional program (e.g., in Haskell), using Isabelle's code generation facilities [8]. For each termination technique that we have implemented in IsaFoR, we have formally proven that whenever such a check is accepted, the termination technique is applied correctly. Hence, we do not need to create an individual Isabelle proof for each proof tree, but just call the check-function for checking the whole tree (which does nothing else but calling the separate checks for each termination technique occurring in the tree). Additionally, our functions deliver error messages that are using notions of term rewriting (in contrast to error messages from a proof assistant that are not easily understandable for the novice). Furthermore, IsaFoR contains a functional parser that accepts XML proof trees. Since even this parser is written in Isabelle, we can freely choose for which programming language we want to generate code (Isabelle currently supports Haskell, OCaml, and SML). At the moment we generate Haskell code, resulting in our certifier CeTA (*Certified Termination Analysis*). However, for a user of CeTA it will make no difference if it was compiled from Haskell sources or OCaml sources.

To certify a proof using CeTA, you just need a CeTA binary plus a termination tool that is able to print the appropriate XML proof tree. Moreover, the runtime of certification is reduced significantly. Whereas it took the other two approaches more than one hour to certify all proofs during the last certified termination competition, CeTA needs about two minutes for all examples, the average time per system being 0.14 seconds. Note that CeTA can also be used for modular certification. Each single application of a termination technique can be certified by just calling the corresponding Haskell function. Another benefit of our system is its robustness. Every proof which uses weaker techniques than those formalized in IsaFoR is accepted. For example, termination provers can use the simple graph estimation of [1], as it is subsumed by our estimation.

IsaFoR, CeTA, and all details about our experiments are available at CeTA's website.[2]

*This project is supported by FWF (Austrian Science Fund) project P18763.

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

[2]`http://cl-informatik.uibk.ac.at/software/ceta`

2 Supported Techniques

Currently, CeTA features certifying proofs for term rewrite systems (TRSs), i.e., the initial problem is always, whether a given TRS $\mathcal{R}$ is terminating or not. Hence on the outermost level of a proof we distinguish between termination and nontermination.

Termination. There are already several techniques for certifying termination proofs. Those techniques can be categorized as follows:

1. A trivial proof (for empty $\mathcal{R}$).
2. Removing some rules $\mathcal{R}'$ from $\mathcal{R}$ such that termination of $\mathcal{R} \setminus \mathcal{R}'$ implies termination of $\mathcal{R}$ [7].
3. Switching to the dependency pair (DP) framework by applying the DP transformation, resulting in the initial DP problem $(\mathsf{DP}(\mathcal{R}), \mathcal{R})$.

In case 2, monotone linear polynomial interpretations over the naturals are supported. For 3, the following processors are available to prove finiteness of a DP problem $(\mathcal{P}, \mathcal{R})$:

Empty $\mathcal{P}$: Emptiness of the $\mathcal{P}$-component of a DP problem implies that the problem is finite.

Dependency Graph: We support a dependency graph estimation that is based on a combination of [6] and [9] (using the function tcap). We call this estimation EDG***. After the estimated graph is computed, $(\mathcal{P}, \mathcal{R})$ is split into the new DP problems $(\mathcal{P}_1, \mathcal{R})$, …, $(\mathcal{P}_n, \mathcal{R})$ (one for each strongly connected component of the estimated graph). Note that our implementation allows a termination tool to use any weaker estimation than EDG***, i.e., any estimation producing a graph which contains at least those edges that are present in EDG***.

Reduction Pair: In the abstract setting of IsaFoR, the notion of *reduction pair* has been formalized. For concrete proofs there are the following instances:

- Weakly monotone linear polynomial interpretations over the natural numbers with negative constants [10]. Those can be used to remove rules from $\mathcal{P}$. Here, only the usable rules [6]—w.r.t. the argument filter that is implicit in the reduction pair—have to be oriented.
- Strictly monotone linear polynomial interpretations over the natural numbers which can be used to remove rules from both $\mathcal{P}$ and $\mathcal{R}$ (where $\mathcal{R}$ can first be reduced to the usable rules).

Nontermination. For the time being, loops are the only certifiable way of proving nontermination. If a TRS is not well-formed (i.e., $l \in \mathcal{V}$ or $\mathcal{V}\mathsf{ar}(r) \not\subseteq \mathcal{V}\mathsf{ar}(l)$ for some rule $l \to r$) the loop is implicit. Otherwise a loop is represented by a context C, a substitution σ, and terms t_1 to t_n such that $t_1 \to \cdots \to t_n \to C[t_1\sigma]$.

3 Use it for Your Termination Prover

To use CeTA for certifying your own proofs, you need a termination tool that generates appropriate XML output plus a CeTA binary (if you want to build CeTA yourself you will still need an Isabelle installation and the IsaFoR library, as well as a Haskell compiler). Hence, the main work will be to modify the termination tool in order to generate XML. In the following we will first give a short overview of the main components that are currently part of our XML format and then show how to call CeTA.

Example 3.1. As an example consider the structure of a termination proof for $\mathcal{R}$, where first a reduction pair has been used to reduce it to the TRS $\mathcal{R}'$. Afterwards the dependency pairs of $\mathcal{R}'$ are computed, resulting in a DP problem. Then the proof proceeds by applying a dependency graph estimation.

```
<proof>
  <ruleRemoval>
    <redPair>...</redPair>
    <trs>ℛ′</trs>
    <dpTrans>
      <dps>DP(ℛ′)</dps>
      <depGraphProc>...</depGraphProc>
    </dpTrans>
  </ruleRemoval>
</proof>
```

The XML format is structured such that on the one hand, new components (like DPs after the dependency pair transformation or the reduction pair processor) are explicitly provided by the user, and on the other hand, the user cannot change components which must not be changed (e.g., the TRS when applying the DP graph processor). The general structure of proofs is as follows:[3]

```
        ⟨proof⟩  ≝  <proof>⟨trsProof⟩</proof> | <proof>⟨trsDisproof⟩</proof>
     ⟨trsProof⟩  ≝  <ruleRemoval>⟨redPair⟩⟨trs⟩⟨trsProof⟩</ruleRemoval>
                 |  <dpTrans>⟨dps⟩⟨dpProof⟩</dpTrans>
                 |  <rIsEmpty/>
      ⟨dpProof⟩  ≝  <depGraphProc>⟨component⟩*</depGraphProc>
                 |  <redPairUrProc>⟨redPair⟩⟨dps⟩⟨usableRules⟩⟨dpProof⟩</redPairUrProc>
                 |  <monoRedPairUrProc>⟨redPair⟩⟨dps⟩⟨trs⟩⟨usableRules⟩⟨dpProof⟩
                        </monoRedPairUrProc>
                 |  <pIsEmpty/>
      ⟨redPair⟩  ≝  <redPair>⟨interpretation⟩</redPair>
⟨interpretation⟩  ≝  <interpretation>⟨type⟩⟨domain⟩⟨interpret⟩*</interpretation>
  ⟨trsDisproof⟩  ≝  <loop>⟨substitution⟩⟨context⟩⟨term⟩*</loop>
                 |  <notWellFormed/>
```

For ⟨*ruleRemoval*⟩, the component ⟨*trs*⟩ holds the rules that could only be weakly oriented by the given ⟨*redPair*⟩. For ⟨*dpTrans*⟩, the dependency pairs are provided by ⟨*dps*⟩. The list of ⟨*component*⟩s within the dependency graph processor denotes all the strongly connected components (including trivial ones consisting of a single node without a self-edge) in topological order. For ⟨*interpretation*⟩s we currently support as ⟨*type*⟩ only linear polynomials and as ⟨*domain*⟩ only the natural numbers. (Detailed descriptions of all the other components can be found on CeTA's website.)

CeTA is called with two arguments: the first is the problem for which a proof should be certified and the second is the corresponding proof. Hence to certify the proof `proof.xml` for the problem `problem.xml`, CeTA is called as follows:

```
$ CeTA problem.xml proof.xml
```

The problem and the proof have to be in XML. For the problem the proposed XTC format—that should soon replace the TPDB format in the termination competition—is used.

Before applying a check-function to a given proof, the internal data structure is converted to XML and compared to the input string. A proof is only accepted if both are equal modulo whitespace. In this way it is ensured that (non)termination of the right TRS is proven.

[3] http://cl-informatik.uibk.ac.at/software/ceta/xml/ceta.xsd[.pdf]

4 Results and Future Work

We ran extensive tests on the 1391 TRSs from version 5.0 of the termination problems data base to evaluate the usefulness of CeTA. As termination tool we used $\mathsf{T_TT_2}$ [11].

All tests have been performed on a server equipped with eight dual-core AMD Opteron® 885 processors running at a CPU rate of 2.6 GHz on 64 GB of system memory and with a time limit of 60 seconds for each TRS. The results can be seen in the following table (where times are given in seconds and (proof-)sizes in kilobytes).

	YES		NO		CeTA		
	#	time (avg.)	#	time (avg.)	#	time (avg.)	size (avg.)
$\mathsf{T_TT_2}$	572	460 (0.80)	214	147 (0.68)	786	114 (0.14)	41,145 (52.35)

The YES columns of the table denote termination proofs, whereas the NO columns denote found loops. That the numbers for YES and NO sum up to 786, shows that CeTA could certify every proof generated by $\mathsf{T_TT_2}$.

For the future we are eager to combine our attempts for a certification XML format with other approaches (like the *termination certificates grammar* of Rainbow) to obtain a standard that can then be used by all termination tools. Further, since we are working with automatically generated code, it would be possible to combine our implementation with extracted code from other formalizations in order to obtain a more powerful certifier.

References

[1] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1–2):133–178, 2000.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science. Springer, 2004.

[3] F. Blanqui, W. Delobel, S. Coupet-Grimal, S. Hinderer, and A. Koprowski. CoLoR, a Coq library on rewriting and termination. In *Proc. WST'06*, pages 69–73, 2006.

[4] É. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Certification of automated termination proofs. In *Proc. FroCoS'07*, volume 4720 of *Lecture Notes in Artificial Intelligence*, pages 148–162, 2007.

[5] P. Courtieu, J. Forest, and X. Urbain. Certifying a termination criterion based on graphs, without graphs. In *TPHOLs'08*, volume 5170 of *Lecture Notes in Computer Science*, pages 183–198, 2008.

[6] J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proc. FroCoS'05*, volume 3717 of *Lecture Notes in Artificial Intelligence*, pages 216–231, 2005.

[7] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE. In *Proc. RTA'04*, volume 3091 of *Lecture Notes in Computer Science*, 2004.

[8] F. Haftmann and T. Nipkow. A code generator framework for Isabelle/HOL. Technical Report 364/07, Department of Computer Science, University of Kaiserslautern, Aug. 2007.

[9] N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1-2):172–199, 2005.

[10] N. Hirokawa and A. Middeldorp. Tyrolean Termination Tool: Techniques and features. *Information and Computation*, 205(4):474–511, 2007.

[11] M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In *Proc. RTA'09*, volume 5595 of *Lecture Notes in Computer Science*, pages 295–304, 2009.

[12] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[13] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proc. TPHOLs'09*, Lecture Notes in Computer Science, 2009. To appear.

Inductive Theorem Proving meets Dependency Pairs[*]

S. Swiderski, M. Parting, J. Giesl, C. Fuhs
LuFG Informatik 2
RWTH Aachen University
Aachen, Germany

P. Schneider-Kamp
Dept. of Mathematics & CS
University of Southern Denmark
Odense, Denmark

Abstract

Current techniques and tools for automated termination analysis of term rewrite systems (TRSs) are already very powerful. However, they fail for algorithms whose termination is essentially due to an *inductive* argument. Therefore, we show how to couple one of the most popular techniques for TRS termination (the *dependency pair* method) with inductive theorem proving. As confirmed by the implementation of our new approach in the tool AProVE, now TRS termination techniques are also successful on this important class of algorithms.

1 Introduction

All current termination tools for TRSs fail on a certain class of natural algorithms like the TRS $\mathcal{R}_{sort}$. It consists of the usual rules for eq and ge on natural numbers represented using 0 and s and of the rules:

$$
\begin{array}{rcll}
\mathsf{max}(\mathsf{nil}) & \to & \mathsf{0} & \\
\mathsf{max}(\mathsf{co}(x,\mathsf{nil})) & \to & x & \\
\mathsf{max}(\mathsf{co}(x,\mathsf{co}(y,xs))) & \to & \mathsf{if}_1(\mathsf{ge}(x,y),x,y,xs) & \\
\mathsf{if}_1(\mathsf{true},x,y,xs) & \to & \mathsf{max}(\mathsf{co}(x,xs)) & \\
\mathsf{if}_1(\mathsf{false},x,y,xs) & \to & \mathsf{max}(\mathsf{co}(y,xs)) & \\
\mathsf{del}(x,\mathsf{nil}) & \to & \mathsf{nil} & \\
\mathsf{del}(x,\mathsf{co}(y,xs)) & \to & \mathsf{if}_2(\mathsf{eq}(x,y),x,y,xs) & \\
\mathsf{if}_2(\mathsf{true},x,y,xs) & \to & xs & (1) \\
\mathsf{if}_2(\mathsf{false},x,y,xs) & \to & \mathsf{co}(y,\mathsf{del}(x,xs)) & \\
\mathsf{sort}(\mathsf{nil}) & \to & \mathsf{nil} & \\
\mathsf{sort}(\mathsf{co}(x,xs)) & \to & \mathsf{co}(\,\mathsf{max}(\mathsf{co}(x,xs)), & \\
 & & \quad \mathsf{sort}(\mathsf{del}(\,\mathsf{max}(\mathsf{co}(x,xs)),\,\mathsf{co}(x,xs)))\,) &
\end{array}
$$

For any list xs, $\mathsf{max}(xs)$ computes its maximum and $\mathsf{del}(n,xs)$ deletes the first occurrence of n from xs. To sort a non-empty list ys, $\mathsf{sort}(ys)$ reduces to "$\mathsf{co}(\mathsf{max}(ys),\mathsf{sort}(\mathsf{del}(\mathsf{max}(ys),ys)))$". So $\mathsf{sort}(ys)$ starts with the maximum of ys and then sort is called recursively on the list that results from ys by deleting the first occurrence of its maximum. Note that

every non-empty list contains its maximum. (2)

Hence, the list $\mathsf{del}(\mathsf{max}(ys),ys)$ is shorter than ys and thus, $\mathcal{R}_{sort}$ is terminating. So (2) is the main argument needed for termination of $\mathcal{R}_{sort}$. For automation, one faces two problems:

(a) One has to detect the main argument needed for termination and one has to find out that the TRS is terminating provided that this argument is valid.

(b) One has to prove that the argument detected in (a) is valid.

Here, (2) requires a non-trivial induction proof that relies on the max- and del-rules. Such proofs cannot be done by TRS termination techniques, but they could be performed by state-of-the-art inductive theorem provers. So we would like to use an inductive theorem prover to solve Problem (b) and combine it with TRS termination provers to solve Problem (a). Thus, one has to extend the TRS termination techniques such that they can automatically synthesize an argument like (2) and find out that this argument suffices to complete the termination proof. Sect. 2 gives the main idea for our improvement. To be powerful in practice, we need the new result that innermost termination of many-sorted term rewriting and of unsorted term rewriting is equivalent. We expect that this observation will be useful also for other applications in term rewriting. In Sect. 3, we couple the DP method with inductive theorem proving to show termination of TRSs like $\mathcal{R}_{sort}$ automatically. We implemented our new technique in the termination prover AProVE [1], which we also extended by a small inductive theorem prover. Note that our results allow to couple *any* termination prover implementing DPs with *any* inductive theorem prover. Thus, by using a more powerful theorem prover, one could further increase the power of our new method. In Sect. 4, we evaluate our contributions.

[*]Supported by the DFG Research Training Group 1298 (*AlgoSyn*), the DFG grant GI 274/5-2, and the G.I.F. grant 966-116.6.

2 Many-Sorted Rewriting for Innermost Termination

Here, we only regard the *innermost* rewrite relation $\xrightarrow{\mathsf{i}}$ and prove innermost termination, For large classes of TRSs (e.g., TRSs resulting from programming languages or non-overlapping TRSs like $\mathcal{R}_{sort}$), innermost termination also implies termination. We use the DP method [2] to prove innermost termination. The set $DP(\mathcal{R}_{sort})$ contains, e.g., the following DP, where SORT is the tuple symbol for sort:

$$\mathsf{SORT}(\mathsf{co}(x,xs)) \quad\to\quad \mathsf{SORT}(\mathsf{del}(\mathsf{max}(\mathsf{co}(x,xs)),\mathsf{co}(x,xs))) \qquad (3)$$

Standard techniques suffice to simplify the initial DP problem $(DP(\mathcal{R}_{sort}),\mathcal{R}_{sort})$ to $(\{(3)\},\mathcal{R}'_{sort})$. Here, $\mathcal{R}'_{sort}$ is $\mathcal{R}_{sort}$ without the two sort-rules. Now, however, standard techniques like the reduction pair processor all fail, since termination of this DP problem essentially relies on the inductive argument (2).

To conclude innermost termination of the original TRS, our goal is to prove the absence of infinite innermost $(\mathcal{P},\mathcal{R})$-chains $s_1\sigma \xrightarrow{\mathsf{i}}_{\mathcal{P}} t_1\sigma \xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}} s_2\sigma \xrightarrow{\mathsf{i}}_{\mathcal{P}} t_2\sigma \xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}} \ldots$ where $s_i \to t_i$ are variable-renamed DPs from $\mathcal{P}$ and "$\xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}}$" denotes zero or more reduction steps to a normal form. The "classical" reduction pair processor ensures $s_1\sigma \underset{(\sim)}{\succsim} t_1\sigma \succsim s_2\sigma \underset{(\sim)}{\succsim} t_2\sigma \succsim \ldots$ and removes DPs with $s_i\sigma \succ t_i\sigma$.

However, instead of requiring a strict decrease when going from the left-hand side $s_i\sigma$ of a DP to the right-hand side $t_i\sigma$, it would also suffice to require a strict decrease when going from the right-hand side $t_i\sigma$ to the *next* left-hand side $s_{i+1}\sigma$. In other words, if *every* reduction of $t_i\sigma$ to normal form makes the term strictly smaller w.r.t. $\succ$, then we also have $t_i\sigma \succ s_{i+1}\sigma$. Hence, then the DP $s_i \to t_i$ cannot occur infinitely often and can be removed from the DP problem. Our goal is to formulate a new processor based on this idea. We can remove a DP $s \to t$ from a DP problem $(\mathcal{P},\mathcal{R})$ where $\mathcal{P}\cup\mathcal{R} \subseteq \succsim$ if

$$\text{for every normal substitution } \sigma,\ t\sigma \xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}} q \text{ implies } t\sigma \succ q. \qquad (4)$$

To remove (3) from $(\{(3)\},\mathcal{R}'_{sort})$ with the criterion above, we must use a reduction pair satisfying (4). Here, t is the right-hand side of (3), i.e., $t = \mathsf{SORT}(\mathsf{del}(\mathsf{max}(\mathsf{co}(x,xs)),\mathsf{co}(x,xs)))$.

Towards automation, we will weaken (4) step by step. We currently have to regard substitutions like $\sigma(x) = \mathsf{true}$ and require that $t\sigma \succ q$ holds, although intuitively, here x stands for a number (and not a Boolean value). However, it suffices to consider "well-typed" terms. So far, we regarded untyped TRSs. We now extend the signature $\mathcal{F}$ by (monomorphic) types. For any TRS $\mathcal{R}$ over a signature $\mathcal{F}$, one can use a standard type inference algorithm to compute a typed variant $\mathcal{F}'$ of the original signature $\mathcal{F}$ such that $\mathcal{R}$ is well typed: All terms in $\mathcal{R}$ are well typed w.r.t. $\mathcal{F}'$ and for each $\ell \to r \in \mathcal{R}$, the terms ℓ and r have the same type. By using the most general typed variant, fewer terms are considered to be well typed and (4) has to be required for fewer substitutions σ. E.g., to make $\{(3)\}\cup\mathcal{R}'_{sort}$ well typed, we use:

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
\mathsf{0} &:& \mathsf{nat} & \mathsf{del},\mathsf{co} &:& \mathsf{nat}\times\mathsf{list}\to\mathsf{list} & \mathsf{ge},\mathsf{eq} &:& \mathsf{nat}\times\mathsf{nat}\to\mathsf{bool}\\
\mathsf{s} &:& \mathsf{nat}\to\mathsf{nat} & \mathsf{true},\mathsf{false} &:& \mathsf{bool} & \mathsf{SORT} &:& \mathsf{list}\to\mathsf{tuple}\\
\mathsf{max} &:& \mathsf{list}\to\mathsf{nat} & \mathsf{nil} &:& \mathsf{list} & \mathsf{if}_1,\mathsf{if}_2 &:& \mathsf{bool}\times\mathsf{nat}\times\mathsf{nat}\times\mathsf{list}\to\mathsf{list}
\end{array}$$

The following theorem shows that innermost termination is a *persistent* property.

Theorem 1. *Let $\mathcal{R}$ be a TRS over $\mathcal{F}$ and $\mathcal{V}$, let $\mathcal{R}$ be well typed w.r.t. the typed variants $\mathcal{F}'$ and $\mathcal{V}'$. $\mathcal{R}$ is innermost terminating f. all well-typed terms w.r.t. $\mathcal{F}'$ and $\mathcal{V}'$ iff $\mathcal{R}$ is innermost terminating f.* all *terms.*

As noted by [4], this property follows from [5]. To our knowledge, it has never been explicitly stated or applied before. We expect several points where Thm. 1 could simplify innermost termination proofs.[1] Here, we use Thm. 1 to weaken the condition (4) to remove a DP from a DP problem $(\mathcal{P},\mathcal{R})$. Now one can use any typed variant where $\mathcal{P}\cup\mathcal{R}$ is well typed. To remove $s \to t$ from $\mathcal{P}$, it suffices if

$$\text{for every normal substitution } \sigma \textit{ where } t\sigma \textit{ is well typed},\ t\sigma \xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}} q \text{ implies } t\sigma \succ q. \qquad (5)$$

[1] For example, by Thm. 1 one could switch to termination methods like [3] exploiting sorts.

3 Coupling DPs and Inductive Theorem Proving

Condition (5) is still too hard. We show (in Thm. 2) that one can often relax (5) to ground substitutions σ. Moreover, we require that for all instantiations $t\sigma$ as above, every reduction of $t\sigma$ to its normal form uses a strictly decreasing rule $\ell \to r$ on a monotonic position π. A position π in a term u is *monotonic* w.r.t. $\succ$ iff $t_1 \succ t_2$ implies $u[t_1]_\pi \succ u[t_2]_\pi$ for all t_1, t_2. To remove $s \to t$ from $\mathscr{P}$, now it suffices if

for every normal σ where $t\sigma$ is well-typed and ground, every reduction "$t\sigma \xrightarrow{\mathrm{i}}{}^{!}_{\mathscr{R}} q$" has the form

$$t\sigma \xrightarrow{\mathrm{i}}{}^{*}_{\mathscr{R}} s[\ell\delta]_\pi \xrightarrow{\mathrm{i}}_{\mathscr{R}} s[r\delta]_\pi \xrightarrow{\mathrm{i}}{}^{!}_{\mathscr{R}} q \tag{6}$$

for a rule $\ell \to r \in \mathscr{R}$ where $\ell \succ r$ and where the position π in s is monotonic w.r.t. $\succ$.

A popular class of reduction pairs $(\succsim, \succ)$ is based on *polynomial interpretations*. E.g., consider the interpretation *Pol* with $\mathsf{s}_{Pol} = 1 + x_1$, $\mathsf{co}_{Pol} = 1 + x_1 + x_2$, $\mathsf{SORT}_{Pol} = \mathsf{max}_{Pol} = x_1$, $\mathsf{if}_{1Pol} = 1 + x_2 + x_3 + x_4$, $\mathsf{del}_{Pol} = x_2$, $\mathsf{if}_{2Pol} = 1 + x_3 + x_4$, and $f_{Pol} = 0$, otherwise. For $(\succsim_{Pol}, \succ_{Pol})$, all rules of $\mathscr{R}'_{sort}$ and (3) are weakly decreasing, and (6) is satisfied for the right-hand side t of (3): In every reduction $t\sigma \xrightarrow{\mathrm{i}}{}^{!}_{\mathscr{R}} q$ where $t\sigma$ is well-typed and ground, eventually one has to apply the strictly decreasing rule (1). The del-algorithm uses (1) to delete an element, i.e., reduce the length of the list. Note that (1) is applied within a context $\mathsf{SORT}(\mathsf{co}(..., \ldots \mathsf{co}(..., \square)))$, so (1) is used on a monotonic position w.r.t. $\succ_{Pol}$.

To check automatically whether every reduction of $t\sigma$ to normal form uses a strictly decreasing rule on a monotonic position, we add new rules and function symbols to $\mathscr{R}$ to get an extended TRS $\mathscr{R}^\succ$, and for every term u we define a corresponding term $u^\succ$. For non-overlapping TRSs $\mathscr{R}$, we then have: If $u^\succ \xrightarrow{\mathrm{i}}{}^{*}_{\mathscr{R}^\succ} \mathsf{tt}$, then for all q, $u \xrightarrow{\mathrm{i}}{}^{!}_{\mathscr{R}} q$ implies $u \succ q$. To get $\mathscr{R}^\succ$, we first introduce a new symbol $f^\succ$ for every defined symbol f in $\mathscr{R}$. Now $f^\succ(u_1, ..., u_n)$ should reduce to tt in $\mathscr{R}^\succ$ whenever the reduction of $f(u_1, ..., u_n)$ in $\mathscr{R}$ uses a strictly decreasing rule on a monotonic position. If $f(\ell_1, ..., \ell_n) \to r \in \mathscr{R}$ was strictly decreasing, then we add $f^\succ(\ell_1, ..., \ell_n) \to \mathsf{tt}$ in $\mathscr{R}^\succ$. Otherwise, a strictly decreasing rule will be used on a monotonic position to reduce an instance of $f(\ell_1, \ldots, \ell_n)$ if this holds for the corresponding instance of the right-hand side r. So then we add $f^\succ(\ell_1, ..., \ell_n) \to r^\succ$ in $\mathscr{R}^\succ$ instead. Next, we define $u^\succ$ for any term u over the signature of $\mathscr{R}$. For $u \in \mathscr{V}$, let $u^\succ = \mathsf{ff}$. If $u = f(u_1, ..., u_n)$, then we regard the subterms on the monotonic positions of u and check whether their reduction uses a strictly decreasing rule. For any n-ary symbol f, let $mon_\succ(f)$ contain those positions from $\{1, \ldots, n\}$ where $f(x_1, ..., x_n)$ is monotonic. If $mon_\succ(f) = \{i_1, ..., i_m\}$, then for $u = f(u_1, ..., u_n)$ we obtain $u^\succ = u^\succ_{i_1} \vee ... \vee u^\succ_{i_m}$, if f is a constructor. If f is defined, then a strictly decreasing rule could also be applied at the root of u. Hence, then we have $u^\succ = u^\succ_{i_1} \vee ... \vee u^\succ_{i_m} \vee f^\succ(u_1, ..., u_n)$. Of course, $\mathscr{R} \subseteq \mathscr{R}^\succ$, and $\mathscr{R}^\succ$ also contains rules for "$\vee$".

The only rules of $\mathscr{R}'_{sort}$ with a strict decrease are the last two max-rules and (1). So $\mathscr{R}'^{\succ_{Pol}}_{sort}$ contains, among others, the rules given on the right.

$$\begin{array}{rcl}
\mathsf{max}^\succ(\mathsf{nil}) & \to & \mathsf{ff} \\
\mathsf{max}^\succ(\mathsf{co}(x, \mathsf{nil})) & \to & \mathsf{tt} \\
\mathsf{max}^\succ(\mathsf{co}(x, \mathsf{co}(y, xs))) & \to & \mathsf{tt} \\
\mathsf{if}^\succ_1(\mathsf{true}, x, y, xs) & \to & \mathsf{max}^\succ(\mathsf{co}(x, xs)) \\
\mathsf{if}^\succ_1(\mathsf{false}, x, y, xs) & \to & \mathsf{max}^\succ(\mathsf{co}(y, xs)) \\
\mathsf{del}^\succ(x, \mathsf{nil}) & \to & \mathsf{ff} \\
\mathsf{del}^\succ(x, \mathsf{co}(y, xs)) & \to & \mathsf{if}^\succ_2(\mathsf{eq}(x, y), x, y, xs) \\
\mathsf{if}^\succ_2(\mathsf{true}, x, y, xs) & \to & \mathsf{tt} \\
\mathsf{if}^\succ_2(\mathsf{false}, x, y, xs) & \to & \mathsf{del}^\succ(x, xs)
\end{array}$$

Now we can again reformulate the condition (6). To remove $s \to t$ from $\mathscr{P}$, now it suffices if

for every normal substitution σ where $t\sigma$ is well typed and ground, we have $t^\succ\sigma \xrightarrow{\mathrm{i}}{}^{*}_{\mathscr{R}^\succ} \mathsf{tt}$. (7)

To remove (3) using $(\succsim_{Pol}, \succ_{Pol})$, we require "$t^{\succ_{Pol}}\sigma \xrightarrow{\mathrm{i}}{}^{*}_{\mathscr{R}'^{\succ_{Pol}}_{sort}} \mathsf{tt}$", where t is the right-hand side of (3). Here, $t^{\succ_{Pol}}$ is $\mathsf{del}^{\succ_{Pol}}(\mathsf{max}(\mathsf{co}(x, xs)), \mathsf{co}(x, xs))$ when simplifying disjunctions with ff. So to remove (3), we require the following for all σ where $t\sigma$ is well typed and ground: $\mathsf{del}^{\succ_{Pol}}(\mathsf{max}(\mathsf{co}(x, xs)), \mathsf{co}(x, xs))\sigma \xrightarrow{\mathrm{i}}{}^{*}_{\mathscr{R}'^{\succ_{Pol}}_{sort}} \mathsf{tt}$. Note that $\mathsf{del}^{\succ_{Pol}}$ computes the *member*-function, i.e., $\mathsf{del}^{\succ_{Pol}}(x, xs)$ holds iff x occurs in the list xs. Thus, the conjecture is equivalent to the main termination argument (2) for $\mathscr{R}_{sort}$, i.e., that every non-empty list contains its maximum. Hence, we can now use termination arguments like (2) with DPs.

Conditions like (7) correspond to the question if a suitable conjecture is *inductively valid*: For a TRS $\mathscr{R}$ and terms t,s over $\mathscr{F}$ and $\mathscr{V}$, $t = s$ is *inductively valid* ("$\mathscr{R} \models_{ind} t = s$") iff there exist typed variants $\mathscr{F}'$ and $\mathscr{V}'$ such that $\mathscr{R}$, t, s are well typed, and $t\sigma \overset{i}{\leftrightarrow}{}^{*}_{\mathscr{R}} s\sigma$ holds for all substitutions σ over $\mathscr{F}'$ where $t\sigma$, $s\sigma$ are well-typed ground terms. While undecidable, $\mathscr{R} \models_{ind} t = s$ can often be proved by inductive theorem provers. From (7), we get that in a DP problem $(\mathscr{P},\mathscr{R})$ with $\mathscr{P} \cup \mathscr{R} \subseteq \succsim$, a pair $s \to t$ can be removed from $\mathscr{P}$ if $\mathscr{R}^{\succ} \models_{ind} t^{\succ} = \mathsf{tt}$. Now we formulate a new DP processor based on this criterion. It transforms $(\mathscr{P},\mathscr{R})$ not only into $(\mathscr{P} \setminus \{s \to t\},\mathscr{R})$, but it also generates the problem $(DP(\mathscr{R}),\mathscr{R})$ to ensure innermost termination of $\mathscr{R}$. Moreover, $(\mathscr{P},\mathscr{R})$ must have the *tuple property*, i.e., for all $s \to t \in \mathscr{P}$, root(s) and root(t) are tuple symbols and tuple symbols occur nowhere else in $\mathscr{P}$ or $\mathscr{R}$.

Theorem 2 (Induction Processor)**.** *Let $(\succsim,\succ)$ be a reduction pair, let $(\mathscr{P},\mathscr{R})$ have the tuple property, let $\mathscr{R}$ be non-overlapping, and let there be no critical pairs between $\mathscr{R}$ and $\mathscr{P}$. Then Proc is sound:*

$$Proc(\,(\mathscr{P},\mathscr{R})\,) = \begin{cases} \{\ (\mathscr{P} \setminus \{s \to t\},\mathscr{R}),\ \ (DP(\mathscr{R}),\mathscr{R})\ \}, & \textit{if } \mathscr{R}^{\succ} \models_{ind} t^{\succ} = \mathsf{tt} \textit{ and } \mathscr{P} \cup \mathscr{R} \subseteq \succsim \\ \{\ (\mathscr{P},\mathscr{R})\ \}, & \textit{otherwise} \end{cases}$$

In our example, we want to remove the DP (3) from the DP problem $(\{(3)\},\mathscr{R}'_{sort})$. We prove $\mathscr{R}'^{\succ_{Pol}}_{sort} \models_{ind} \mathsf{del}^{\succ_{Pol}}(\mathsf{max}(\mathsf{co}(x,xs)),\mathsf{co}(x,xs)) = \mathsf{tt}$ by an inductive theorem prover. The small induction prover in AProVE, e.g., find this proof automatically. Then the induction processor returns the trivial problem $(\varnothing,\mathscr{R}'_{sort})$ and the easily solved problem $(DP(\mathscr{R}'_{sort}),\mathscr{R}'_{sort})$. Thus, termination of $\mathscr{R}_{sort}$ is verified.

4 Experiments and Conclusion

We introduced a new DP processor for TRSs that terminate because of an inductive property. This property is extracted automatically and transformed into a conjecture that can be verified by current inductive theorem provers. To increase power, we showed that it suffices to prove this conjecture only for well-typed terms, even if the original TRS is untyped. We implemented our contributions in our termination tool AProVE [1] and evaluated it on 19 typical TRSs for classical algorithms where the termination proof requires an inductive argument. So far, all tools in the *Termination Competition* failed on these examples, whereas our new version of AProVE automatically proves termination of 16 of them within a timeout of 1 minute per example. Thus, our method substantially advances automated termination proving, since it allows the first combination of powerful TRS termination tools with inductive theorem provers. For details on our experiments and to run our implementation, we refer to `http://aprove.informatik.rwth-aachen.de/eval/Induction/`. A longer version of this paper will appear in [6].

Acknowledgements. We are very grateful to Aart Middeldorp and Hans Zantema for suggesting the proof idea of Thm. 1 and for pointing us to [5].

References

[1] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the DP framework. In *Proc. IJCAR'06*, LNAI 4130, pp. 281-286, 2006.

[2] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155-203, 2006.

[3] S. Lucas and J. Meseguer. Order-sorted dependency pairs. In *Proc. PPDP'08*, pp. 108-119, ACM Press, 2008.

[4] A. Middeldorp and H. Zantema. Personal communication, 2008.

[5] J. van de Pol. Modularity in many-sorted term rewriting. Master's Thesis, Utrecht University, 1992.

[6] S. Swiderski, M. Parting, J. Giesl, C. Fuhs, and P. Schneider-Kamp. Termination analysis by dependency pairs and inductive theorem proving. In *Proc. CADE'09*, LNAI, 2009. To appear.

Deciding Loops under Strategies

René Thiemann (rene.thiemann@uibk.ac.at), University of Innsbruck, Austria
Christian Sternagel[*] (christian.sternagel@uibk.ac.at), University of Innsbruck, Austria

1 Introduction

Termination is an important property of term rewrite systems (TRSs). Therefore, much effort has been spent on developing and automating techniques for showing termination of TRSs. But in order to detect bugs, it is at least as important to prove *non-termination* of TRSs. Note that for rewriting under strategies, one cannot ignore the strategy, since a non-terminating TRS may still be terminating due to the strategy. Thus, it is important to develop automated techniques to disprove termination of TRSs under strategies.

Most of the techniques and tools for showing non-termination detect loops, a derivation of the form $t \to_{\mathcal{R}}^{+} C[t\sigma]$. If one wants to prove non-termination under some strategy $\mathcal{S}$ then one can use a complete transformation $T_{\mathcal{S}}$ like [2, 4, 5] and then try to find a loop of the transformed system. There are two main drawbacks using these transformations. The first problem is a practical one. Often the loops of $\mathcal{R}$ are transformed into much longer loops in $T_{\mathcal{S}}(\mathcal{R})$ and hence, the search space for loops may become critical. And even more severe is the problem, that all complete transformations of [2, 4, 5] translate a loop $t \to_{\mathcal{R}}^{+} C[t\sigma]$ into a non-looping infinite derivation in $T_{\mathcal{S}}(\mathcal{R})$ as soon as $C \neq \Box$.

Example 1. *Consider the TRS $\mathcal{R}_n$ for arithmetic with n-bit numbers.*

$$\begin{aligned} \mathsf{p}(\mathsf{0}) &\to \mathsf{0} & \mathsf{plus}(\mathsf{0},y) &\to y \\ \mathsf{p}(\mathsf{s}(x)) &\to x & \mathsf{plus}(\mathsf{s}(x),y) &\to \mathsf{s}(\mathsf{plus}(x,y)) \\ \mathsf{minus}(x,\mathsf{0}) &\to x & \mathsf{inf} &\to \mathsf{s}(\mathsf{inf}) \\ \mathsf{minus}(x,\mathsf{s}(y)) &\to \mathsf{p}(\mathsf{minus}(x,y)) & \mathsf{s}^{2^n}(x) &\to \mathsf{overflow} \end{aligned}$$

Here, the last rule is used to model that an overflow occurred due to the n-bit restriction by using the outermost strategy. Alternatively, for $n = 0$ one can drop the last rule context-sensitive rewriting where the replacing map μ only restricts rewriting below s*, i.e., $\mu(\mathsf{s}) = \emptyset$. We focus on the loops*

$$t_1 = \mathsf{minus}(x,\mathsf{inf}) \to \mathsf{minus}(x,\mathsf{s}(\mathsf{inf})) \to \mathsf{p}(\mathsf{minus}(x,\mathsf{inf})) = C_1[t_1\sigma_1] \quad \text{and}$$
$$t_2 = \mathsf{plus}(\mathsf{inf},y) \to \mathsf{plus}(\mathsf{s}(\mathsf{inf}),y) \to \mathsf{s}(\mathsf{plus}(\mathsf{inf},y)) = C_2[t_2\sigma_2]$$

where $\sigma_1 = \sigma_2 = \{\}$, $C_1 = \mathsf{p}(\Box)$, and $C_2 = \mathsf{s}(\Box)$. The first loop is an outermost loop. The second loop creates after 2^n rounds a new outermost redex for the overflow rule, so it cannot be outermost. A similar result holds for the context-sensitive variant.

Note that all transformations [2, 4, 5] produce TRSs from Ex. 1 where all infinite reductions are non-looping. In contrast, in this paper we present the three *direct* approaches of [6, 7] in a uniform way. They can *decide* for a given loop whether it is a loop for the context-sensitive, innermost, or outermost strategy and therefore successfully handle Ex. 1. These decision procedures were the key-ingredient to get the highest scores in the termination competition for disproving termination under strategies.

2 Loops under Strategies

We only regard finite signatures and TRSs and refer to [1] for the basics of rewriting. We use ℓ, r, s, t, u for terms, f, g for function symbols, x, y for variables, σ, τ for substitutions, i, j, k, n, m for natural numbers, p, q for positions where ε is the root position, and C, D for contexts. Here, contexts are terms which contain exactly one hole $\Box$. The set of variables is denoted by $\mathcal{V}$. Throughout this paper we assume

[*]This author is supported by FWF (Austrian Science Fund) project P18763.

a fixed TRS $\mathcal{R}$ and we write $t \to_p s$ if one can reduce t to s at position p with $\mathcal{R}$, i.e., $t = C[\ell\tau]$ and $s = C[r\tau]$ for some $\ell \to r \in \mathcal{R}$, substitution τ, and context C with $C|_p = \Box$. The term $\ell\tau$ is called a redex at position p. The reduction is outermost / innermost, written $t \xrightarrow{o}_p$ / $\xrightarrow{i}_p s$, iff t contains no redex at a position q above / below p, written $q < p$ / $p > q$. The reduction is context-sensitive ($t \xrightarrow{\mu}_p s$) iff p is a μ-replacing position [3]. If the position is irrelevant we just write $\to$, $\xrightarrow{o}$, $\xrightarrow{i}$, or $\xrightarrow{\mu}$. The TRS $\mathcal{R}$ is non-terminating iff there is an infinite derivation $t_1 \to t_2 \to \dots$. It is outermost / innermost / context-sensitive non-terminating iff there is such an infinite derivation using $\xrightarrow{o}$ / $\xrightarrow{i}$ / $\xrightarrow{\mu}$ instead of $\to$.

To describe the infinite derivation that is induced by a loop, we use context-substitutions [7].

Definition 2 (Context-substitutions)**.** *A* context-substitution *is a pair (C,σ) consisting of a context C and a substitution σ. The n-fold application of (C,σ) to a term t, written $t(C,\sigma)^n$ is defined as follows.*

$$t(C,\sigma)^0 = t \qquad\qquad t(C,\sigma)^{n+1} = C[t(C,\sigma)^n\sigma]$$

From the definition it is obvious that in $t(C,\sigma)^n$ the context C is added n-times above t and t is instantiated by σ^n. Note that also the added contexts are instantiated by σ. For the term $t(C,\sigma)^3$ this is illustrated in Fig. 1.

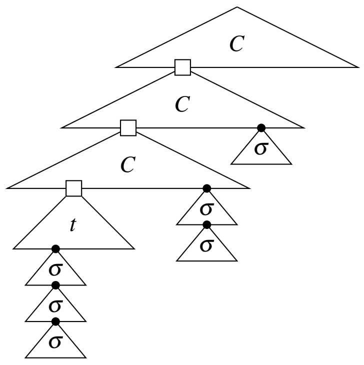


Figure 1: The term $t(C,\sigma)^3$

The following lemma shows that context-substitutions have similar properties to both contexts and substitutions.

Lemma 3 (Properties of context-substitutions)**.**

(i) $t(C,\sigma)^n\sigma = t\sigma(C\sigma,\sigma)^n$.

(ii) $t(C,\sigma)^m(C,\sigma)^n = t(C,\sigma)^{m+n}$.

(iii) *Whenever $t \to_q s$ and $C|_p = \Box$ then $t(C,\sigma)^n \to_{p^n q} s(C,\sigma)^n$.*

Here, property (i) is similar to the fact that $C[t]\sigma = C\sigma[t\sigma]$, and (ii) expresses that context-substitutions can be combined just as substitutions where $\sigma^m\sigma^n = \sigma^{m+n}$. Finally stability and monotonicity of rewriting are used to show in (iii) that rewriting is closed under context-substitutions. Using context-substitutions we can now concisely present the infinite derivation of a loop $t \to^+ C[t\sigma] = t(C,\sigma)$.

$$t(C,\sigma)^0 \to^+ t(C,\sigma)(C,\sigma)^0 = t(C,\sigma)^1 \to^+ \dots \to^+ t(C,\sigma)^n \to^+ \dots \qquad (\star)$$

Hence, for every n the positions of the reductions in the loop are prefixed by an additional p^n where p is the position of the hole in C, cf. Lemma 3 (iii). Now it is natural to define that a derivation $t \to^+ t(C,\sigma)$ is called an $\mathcal{S}$-*loop* iff all steps in ($\star$) respect the strategy $\mathcal{S}$.[1]

Definition 4 ($\mathcal{S}$-loop)**.** *Let $\mathcal{S}$ be a strategy. A loop $t_1 \to_{q_1} t_2 \to_{q_2} \dots t_n \to_{q_n} t_{n+1} = t_1(C,\sigma)$ with $C|_p = \Box$ is an $\mathcal{S}$-*loop *iff all reductions $t_i(C,\sigma)^m \to_{p^m q_i} t_{i+1}(C,\sigma)^m$ respect the strategy $\mathcal{S}$ for all i and m.*

As a direct consequence of Def. 4 one can conclude that every $\mathcal{S}$-loop of a rewrite system $\mathcal{R}$ proves non-termination of $\mathcal{R}$ under strategy $\mathcal{S}$. Note that a loop is not only determined by the derivation $t_1 \to^+ t_{n+1}$, but also by the specific context C that is used. To see this consider the TRS $\mathcal{R} = \{\mathsf{a} \to \mathsf{f}(\mathsf{a},\mathsf{a}), \mathsf{f}(\mathsf{f}(x,y),z) \to \mathsf{b}\}$. Then $\mathsf{a} \to \mathsf{f}(\mathsf{a},\mathsf{a})$ is a looping derivation. For $C = \mathsf{f}(\mathsf{a},\Box)$ this is an outermost loop. However, for $C' = \mathsf{f}(\Box,\mathsf{a})$ we do not obtain an outermost loop since $\mathsf{f}(\mathsf{a},\mathsf{a}) \xrightarrow{o} \mathsf{f}(\mathsf{f}(\mathsf{a},\mathsf{a}),\mathsf{a}) \not\xrightarrow{o} \mathsf{f}(\mathsf{f}(\mathsf{f}(\mathsf{a},\mathsf{a}),\mathsf{a}),\mathsf{a})$. Hence, the choice of C is essential.

To decide whether a loop is an $\mathcal{S}$-loop we first consider the (simple) context-sensitive case: here one can use the essential observation that all positions $p^m q$ are μ-replacing iff p and q are μ-replacing.

Theorem 5 (Deciding context-sensitive-loops)**.** *A loop $t \to^+ C[t\sigma]$ is a context-sensitive loop iff both $t \xrightarrow{\mu}^+ C[t\sigma]$ and the hole in C is at a μ-replacing position.*

[1] Another natural definition of an $\mathcal{S}$-loop would just require that $t(C,\sigma)^n \to^+ t(C,\sigma)^{n+1}$ are $\mathcal{S}$-derivations for all n. That this alternative definition is problematic is discussed in [6, Sect. 2]: the above derivations are innermost iff $t\sigma^n \xrightarrow{i}^+ \circ \trianglerighteq t\sigma^{n+1}$.

3 Deciding Innermost and Outermost Loops

Recall the definition of $\xrightarrow{i}$ and $\xrightarrow{o}$. An innermost / outermost reduction of all terms $t(C,\sigma)^n$ at positions $p^n q$ requires that for no n there is a redex at a position below / above $p^n q$. Hence, all subterms s of all terms $t(C,\sigma)^n$ at positions below / above $p^n q$ must not be *matched* by some left-hand side ℓ of $\mathcal{R}$. Matching problems will allow us to formulate these infinite number of checks in a finite way.

Definition 6 ((Extended) matching problems). *A* matching problem *is a pair* $(\mathcal{M},\sigma)$ *where* $\mathcal{M}$ *is a set of pairs* $s \gtrdot \ell$. *It is* solvable *iff there is a* solution (k,τ) *such that* $s\sigma^k = \ell\tau$ *is satisfied for all* $s \gtrdot \ell \in \mathcal{M}$.

An extended matching problem *is a quintuple* $(D \gtrdot \ell, C, t, \mathcal{M}, \sigma)$. *It is* solvable *iff there is a* solution (n,k,τ) *such that the equation* $D[t(C,\sigma)^n]\sigma^k = \ell\tau$ *is satisfied and* (k,τ) *is a solution to* $(\mathcal{M},\sigma)$.

To simplify presentation we write $(s \gtrdot \ell,\sigma)$ *instead of* $(\{s \gtrdot \ell\},\sigma)$ *and we write* $(D \gtrdot \ell, C, t, \sigma)$ *instead of* $(D \gtrdot \ell, C, t, \varnothing, \sigma)$. *We use the notion "matching problem" also for extended matching problems.*

Consider a reduction $t \rightarrow_q u$ of a loop. Looking at Fig. 1 one can see that for innermost loops the only interesting potential redexes are the subterms of $t|_q$ (i1) and those subterms that are introduced by iteratively applying σ on the variables $\mathcal{V}\mathsf{ar}(t|_q)$ of $t|_q$ (i2). For outermost loops all superterms of $t|_q$ have to be considered (o1) and one also has to analyze superterms that are constructed by iteratively adding the context C around t (o2).

Definition 7 (Initial matching problems). *The set of* initial matching problems *for a reduction* $t \rightarrow_q u$ *and a context-substitution* (C,σ) *with* $C|_p = \square$ *is defined as the set consisting of:*

$$(t|_{q'} \gtrdot \ell, \sigma) \text{ for each } \ell \to r \in \mathcal{R} \text{ and } q' \in \mathcal{P}\mathsf{os}(t) \text{ where } q' > q \quad \text{(i1)}$$

$$(x\sigma \gtrdot \ell, \sigma) \text{ for each } \ell \to r \in \mathcal{R} \text{ and } x \in \textstyle\bigcup_{i \in \mathbb{N}} \mathcal{V}\mathsf{ar}(t|_q \sigma^i) \quad \text{(i2)}$$

$$(t|_{p'} \gtrdot \ell, \sigma) \text{ for each } \ell \to r \in \mathcal{R} \text{ and } p' < q \quad \text{(o1)}$$

$$(C|_{p'} \gtrdot \ell, C\sigma, t\sigma, \sigma) \text{ for each } \ell \to r \in \mathcal{R} \text{ and } p' < p \quad \text{(o2)}$$

Note that the set of variables in (i2) is finite and can easily be computed.

Example 8. *Consider the loop* $t_1 = \mathsf{minus}(x,\mathsf{inf}) \rightarrow_2 \mathsf{minus}(x,\mathsf{s}(\mathsf{inf})) = t_2 \rightarrow_\varepsilon \mathsf{p}(\mathsf{minus}(x,\mathsf{inf})) = t_1(C,\sigma)$ *of Ex. 1 where* $C = \mathsf{p}(\square)$ *and* $\sigma = \{\}$. *For the first reduction in the outermost case we get the matching problems* $MP_{1\ell} = (\mathsf{minus}(x,\mathsf{inf}) \gtrdot \ell, \sigma)$ *for all left-hand sides* ℓ *of* $\mathcal{R}$. *Moreover, we build the extended matching problems* $MP_{2\ell} = (\mathsf{p}(\square) \gtrdot \ell, \mathsf{p}(\square), \mathsf{minus}(x,\mathsf{inf}), \sigma)$. *For the second reduction at root position we only build the extended matching problems* $MP_{3\ell} = (\mathsf{p}(\square) \gtrdot \ell, \mathsf{p}(\square), \mathsf{minus}(x,\mathsf{s}(\mathsf{inf})), \sigma)$.

Theorem 9 (Loops and matching problems). *Let* $t \rightarrow_q u$ *and* (C,σ) *be given such that* $C|_p = \square$. *All reductions* $t(C,\sigma)^n \rightarrow_{p^n q} u(C,\sigma)^n$ *are innermost / outermost iff none of the initial matching problems (i1,i2) / (o1,o2) for* $t \rightarrow_q u$ *and* (C,σ) *is solvable.*

In the next section we provide a decision procedure for matching problems from left-linear TRSs, and therefore also obtain a decision procedure to decide innermost and outermost loops for these TRSs.[2]

4 Deciding Solvability of Matching Problems

To decide matching problems we give convergent rewrite rules to simplify matching problems to $\top$ and $\bot$, representing solvability and non-solvability respectively. Since extended matching problems $(D \gtrdot \ell, C, t, \mathcal{M}, \sigma)$ are only generated if $C \neq \square$, we always assume that $C \neq \square$ for these problems. The reason is that otherwise the simplification rules would not terminate.

[2]How to decide solvability of matching problems in the general (non-linear) case is shown in [6,7]. Then the simplification rules (1) and (6) for matching problems become unsound, and one has to solve (extended) identity problems instead.

Definition 10 (Simplification of matching problems)**.** *Let $MP = (\mathcal{M}, \sigma)$ be a matching problem where $\mathcal{M} = \{s_1 \gtrdot \ell_1, \dots, s_m \gtrdot \ell_m\}$. Let $\mathcal{V}_{incr} = \{x \in \mathcal{V} \mid \exists n : x\sigma^n \notin \mathcal{V}\}$ be the set of increasing variables.*

The relation $\Rightarrow$ is defined as follows. If $m = 0$ then $MP \Rightarrow \top$. Otherwise, $\mathcal{M} = \{s_1 \gtrdot \ell_1\} \cup \mathcal{M}'$ and

$$MP \Rightarrow (\mathcal{M}', \sigma) \text{ if } \ell_1 \in \mathcal{V} \tag{1}$$
$$MP \Rightarrow (\mathcal{M}' \cup \{t_i \gtrdot \ell'_i \mid 1 \le i \le n\}, \sigma) \text{ if } \ell_1 = f(\ell'_1 \dots \ell'_n) \text{ and } s_1 = f(t_1 \dots t_n) \tag{2}$$
$$MP \Rightarrow \bot \text{ if } \ell_1 = f(\dots) \text{ and } s_1 = g(\dots) \text{ where } f \neq g \tag{3}$$
$$MP \Rightarrow \bot \text{ if } \ell_1 = f(\dots) \text{ and } s_1 \in \mathcal{V} \setminus \mathcal{V}_{incr} \tag{4}$$
$$MP \Rightarrow (\{s_i\sigma \gtrdot \ell_i \mid s_i \gtrdot \ell_i \in \mathcal{M}\}, \sigma) \text{ if } \ell_1 = f(\dots) \text{ and } s_1 \in \mathcal{V}_{incr}. \tag{5}$$

For extended matching problems $MP = (D \gtrdot \ell, C, t, \mathcal{M}, \sigma)$ the rules of $\Rightarrow$ are as follows.

$$MP \Rightarrow (\mathcal{M}, \sigma) \text{ if } \ell \in \mathcal{V} \tag{6}$$
$$MP \Rightarrow (D_i \gtrdot \ell_i, C, t, \mathcal{M} \cup \{t_j \gtrdot \ell_j \mid 1 \le j \le n, i \neq j\}, \sigma) \text{ if } \ell = f(\ell_1 \dots \ell_n) \text{ and } D = f(t_1 \dots D_i \dots t_n) \tag{7}$$
$$MP \Rightarrow \bot \text{ if } \ell = f(\dots) \text{ and } D = g(\dots) \text{ where } f \neq g \tag{8}$$
$$MP \Rightarrow \top \text{ if } \ell = f(\dots),\ D = \Box, \text{ and } (\mathcal{M} \cup \{t \gtrdot \ell\}, \sigma) \Rightarrow^* \top \tag{9}$$
$$MP \Rightarrow (C \gtrdot \ell, C\sigma, t\sigma, \mathcal{M}, \sigma) \text{ if } \ell = f(\dots),\ D = \Box, \text{ and } (\mathcal{M} \cup \{t \gtrdot \ell\}, \sigma) \Rightarrow^* \bot \tag{10}$$

Rules (1,6) indicate that a variable matches everything. Rules (2,7) and (3,8) are the standard decomposition and clash rules. Rules (4,5) handle the case that the variable s_1—where σ can be applied arbitrarily often—has to be matched by a non-variable. If $s_1\sigma^n$ will always be a variable one cannot match (4), and otherwise (5) instantiates x one more time. For extended matching problems one has to handle the case $D = \Box$ where one has to choose the n in $t(C,\sigma)^n$. Essentially, it is just chosen whether $n = 0$ is possible (9) or not (10). In the latter case, (C, σ) is just applied one more time.

Theorem 11 (Soundness of the simplification rules)**.** *Let $MP = (s \gtrdot \ell, \sigma)$ or $MP = (D \gtrdot \ell, C, t, \sigma)$ be a matching problem, let ℓ be linear term. Then MP is solvable iff $MP \Rightarrow^* \top$. Moreover, $\Rightarrow$ is convergent.*

Example 12. *None of the matching problems of Ex. 8 is solvable. $MP_{1\ell}$ is reduced to $\bot$ by (2), (1), (3) if $\ell = \mathsf{minus}(\dots)$, or directly by (3), otherwise. For $MP_{2\ell}$ and $MP_{3\ell}$ an application of (8) suffices if $\ell \neq \mathsf{p}(\dots)$. Otherwise, (7), (10), (8) result in $\bot$. Hence, the loop of Ex. 8 is an outermost loop.*

References

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

[2] J. Giesl and A. Middeldorp. Transformation techniques for context-sensitive rewrite systems. *Journal of Functional Programming*, 14(4):379–427, 2004.

[3] S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1:1–61, 1998.

[4] M. Raffelsieper and H. Zantema. A transformational approach to prove outermost termination automatically. In *Proc. WRS '08*, ENTCS 237, pages 3–21, 2009.

[5] R. Thiemann. From outermost termination to innermost termination. In *Proc. SOFSEM '09*, LNCS 5404, pages 533–545, 2009.

[6] R. Thiemann, J. Giesl, and P. Schneider-Kamp. Deciding innermost loops. In *Proc. RTA '08*, LNCS 5117, pages 366–380, 2008.

[7] R. Thiemann and C. Sternagel. Loops under strategies. In *Proc. RTA '09*, LNCS 5595, pages 17–31, 2009. To appear. Available at `http://cl-informatik.uibk.ac.at/~griff/experiments/lus.php`.

A new approach to non-termination analysis of Logic Programs

Dean Voets[*]
Dept. of Computer Science
K.U.Leuven, Belgium
Dean.Voets@cs.kuleuven.be

Danny De Schreye
Dept. of Computer Science
Celestijnenlaan 200A, 3001 Heverlee
Danny.DeSchreye@cs.kuleuven.be

1 Introduction

The main advantage of Logic Programming (LP) is that a declarative programming style leads to less error-prone and more understandable programs, but it can also lead to inefficient program execution and possibly to non-termination. Therefore, analyzing the termination behavior of a logic program is one of the most important steps in proving its correctness. The latter problem has received considerable attention within the community.

Termination analysis is a well developed field in LP. Many different approaches and automated tools have been developed in the last two decades. However, non-termination analysis received little attention. As far as we know, [3] presents the only automated non-termination analyzer for LP, *NTI*.

Very recently, a technique to predict the termination behavior of a logic program was presented in [4]. The authors show that the termination behavior of queries described using modes, can be predicted based on a finite symbolic derivation tree.

In this paper, we show that these symbolic derivation trees can also be used for proving non-termination. We present this approach and show, by means of empirical evaluation, that it is more powerful than the approach of *NTI* [3].

2 Preliminaries

We assume the reader is familiar with standard terminology of logic programs, in particular with SLDNF-resolution, as described in [2]. We will use a special form of SLDNF-resolution where the goals are labeled and are denoted with $N_i : G_i$, with N_i the label of goal G_i. At each node, the leftmost literal is selected. The leftmost literal at node N_i is denoted with L_i^1. An ancestor-relation is defined on these nodes. Basically, a node N_i is an ancestor of a node N_j, if the proof of L_i^1 goes through the proof of L_j^1.

In [4], this generalized form of SLDNF-resolution is extended for moded queries. An input mode denotes an unknown variable-free term. Therefore, it can be instantiated with either a constant or a compound term, during concrete query evaluation. Such an input mode is represented with an *input variable*, denoted by underlining the variables name. *Moded-generalized SLDNF-resolution* allows to define symbolic derivation trees, representing all derivations of the concrete queries denoted by the moded query .This set of concrete queries is called the *denotation* of the moded query. In the following definition, $Term_P$ represents all terms constructible from constant and function symbols in the program and $A(t_1 \to \underline{I_1}, \ldots, t_n \to \underline{I_n})$ represents the concrete atom obtained by replacing the input variables $\underline{I_1}, ..., \underline{I_n}$ by the terms $t_1, \ldots, t_n$.

Definition 1. *Let A be an atom or goal with $\underline{I_1}, \ldots, \underline{I_n}$ as its input variables. The* ***denotation*** *of A is*

$$Den(A) = \{A(t_1 \to \underline{I_1}, \ldots, t_n \to \underline{I_n}) \mid t_i \in Term_P, t_i \text{ is ground}\}. \qquad \square$$

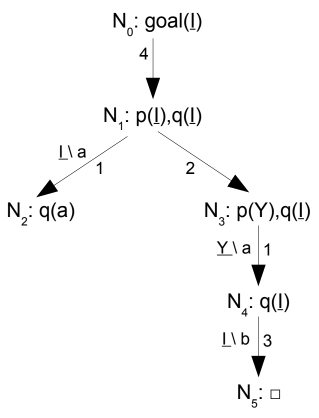
[*]Supported by the Fund for Scientific Research - FWO-project G0561-08

```
p(a).
p(X) :- p(Y).

q(b).

goal(X) :- p(X), q(X).
```

(a) the program



(b) the moded SLDNF-tree

For example, in a language that only contains the constant symbols *a* and *b* and no function symbols, $Den(p(\underline{I})) = \{p(a), p(b)\}$ and $Den(p(X)) = \{p(X)\}$, since this atom does not contain input variables.

Because these moded SLDNF-trees are in general infinite, they cannot be used to analyze the termination behavior of a program. Therefore, a complete loop check for moded generalized SLDNF-resolution, *LP-check*, is defined in [4]. A complete loop check cuts all infinite branches from the tree and thus, constructs finite symbolic derivation trees. For more information about loop checking, we refer to [1].

In the remainder of the paper, we will illustrate the introduced concepts using the program depicted in Figure (a), with $goal(\underline{I})$ as a moded query. Figure (b) shows the symbolic derivation tree for this program and moded query. At node N_3, clause 2 is cut by *LP-check*.

3 A new non-termination condition

To prove non-termination based on moded SLDNF-trees, we look for a path in the tree that can be applied infinitely often. It turns out that we can identify such paths, using three basic properties. There should be no substitutions on input variables in the path, the ancestor-relation must hold between the first and the last selected literal and a special more general relation must hold between these literals.

A moded atom *A* is *moded more general* than a moded atom *B*, if any atom in the denotation of *A* is more general than some atom in the denotation of *B*.

Definition 2. *A moded atom A is* ***moded more general*** *than a moded atom B,* $A \rhd B$*, iff:*

$$\forall I \in Den(A), \exists I' \in Den(B) : I \text{ is more general than } I' \qquad \square$$

For example, $p(X) \rhd p(\underline{I})$, because $Den(p(X)) = \{p(X)\}$ and $p(X)$ is more general than any of the elements of $Den(p(\underline{I})) = \{p(a), p(b)\}$.

If a path in the moded SLDNF-tree satisfies the three properties mentioned earlier, we call this path a *moded more general loop*.

Definition 3. *In a moded SLDNF-derivation D, nodes* $N_i : G_i$ *and* $N_j : G_j$ *are a* ***moded more general loop****,* $N_i : G_i \overset{mmg}{\rightarrow} N_j : G_j$*, iff:*

- *No substitutions on input variables occur in the path from* N_i *to* N_j*.*
- *Node* N_i *is an ancestor of node* N_j*.*
- $L_j^1 \rhd L_i^1$ $\square$

In [5], we have proven that a moded more general loop, $N_i : G_i \overset{mmg}{\rightarrow} N_j : G_j$, corresponds to an infinite loop for every concrete goal in the denotation of G_i.

Theorem 1 (Sufficiency of the moded more general loop)**.** *Let $N_i : G_i \overset{mmg}{\rightarrow} N_j : G_j$ be a moded more general loop in a moded SLDNF-derivation D of program P and moded query I. The sequence of clauses from N_i to N_j, $\langle C_1,\ldots,C_n\rangle$, can be repeated infinitely often for any goal in $Den(G_i)$.* □

We will illustrate this with our running example.

Example 1. *We show that the path between N_1 and N_3 is a moded more general loop:*

- *The ancestor-relations obviously holds because clause 2 is recursive.*
- *There are no substitutions in the path.*
- *As shown previously, $p(X)$ is moded more general than $p(\underline{I})$.*

Therefore, every goal in the denotation of $(p(\underline{I}),q(\underline{I}))$ is non-terminating. Because no substitution occurs between N_0 and N_1, every goal in the denotation of $goal(\underline{I})$ is non-terminating as well. □

3.1 Input-generalizations

For many non-terminating programs, non-termination can be proven using the moded more general loop. However, when a normal variable is replaced by a compound term, non-termination cannot be proven with Definition 3. To overcome this problem, we define *input generalizations*. Informally, an atom A^α is an input generalization of A, if we can obtain A^α by replacing subterms of A by new input variables. In [5], we have proven that non-termination of an input generalized goal implies non-termination of the original goal. Intuitively, this is because the new input variables represent more instantiated terms as the replaced subterms.

We will illustrate the use of these input generalizations with a small example.

Example 2. *The following clause is the recursive clause of reverse list with accumulator.*

```
reverse([H|T],Temp,Res) :- reverse(T,[H|Temp],Res).
```

The moded SLDNF-tree for this program and a query $reverse(X,Y,\underline{I})$, does not contain a moded more general loop. To prove non-termination for this query, we use an input generalization of the original query: $reverse(X,\underline{Y},\underline{I})$.

The constructed derivation for his query contains a moded more general loop. Therefore, this clause is non-terminating w.r.t. the goals in $Den(reverse(X,Y,\underline{I}))$. □

4 Experimental evaluation

We implemented our condition in a non-termination analyzer *P2P*[1]. We tested *P2P* on a benchmark of 48 non-terminating programs, taken from the Termination Competition of 2007[2]. We compared our analyzer with the only other non-termination analyzer, *NTI* [3]. The results are shown in Table 1. In columns *P2P* and *NTI*, a *V* denotes that a non-termination proof was found, while an *X* denotes that no proof was found. The columns *Size* and *Time* give the size in number of nodes of the moded SLDNF-tree and the analysis time in seconds, respectively.

The results are very satisfactory. For all programs in the benchmark, *P2P* proves non-termination and thus, improves on the results of *NTI*. The analyzer is very efficient. Any benchmark program is analyzed in less than a second and the memory use never exceeds a few megabytes.

[1] Available at www.cs.kuleuven.be/˜dean/p2p.html

[2] Available at www.lri.fr/˜marche/termination-competition/

Name program	*P2P*	Size	Time	*NTI*	Name program	*P2P*	Size	Time	*NTI*
ackermann-ioi	V	9	0.33	V	permutation-fb	V	22	0.26	V
bad sublist	V	33	0.29	V	pl1.1	V	8	0.25	V
binary4	V	12	0.27	V	pl3.1.1	V	12	0.30	V
delete-bff	V	13	0.31	V	pl3.5.6	V	13	0.31	V
der-fb	V	22	0.29	V	pl4.0.1-oooi	V	33	0.27	V
doublehalfpred	V	38	0.28	V	pl4.5.2	V	481	0.36	V
example4-2	V	4	0.23	V	pl4.5.3a	V	10	0.29	V
flatlength-fbf	V	14	0.23	V	pl4.5.3b	V	10	0.24	V
flatlength-ffb	V	19	0.23	V	pl4.5.3c	V	11	0.27	V
flat-oi	V	9	0.26	X	pl5.2.2	V	59	0.27	V
frontier-fb	V	12	0.27	V	pl7.6.2.a	V	39	0.27	X
ifdiv	V	19	0.29	V	pl7.6.2.b	V	45	0.33	X
in-bf	V	18	0.29	V	quicksort-fb	V	72	0.26	V
inorder-fb	V	4	0.27	V	quicksort-oi	V	74	0.26	V
insert-bff	V	22	0.29	V	reverse-fb	V	9	0.32	V
log2a-oi	V	35	0.25	V	select-bff	V	8	0.32	V
log2b-oi	V	29	0.28	V	slowsort-fb	V	123	0.27	V
mapcolor	V	23	0.31	V	slowsort-oi	V	26	0.26	V
member-bf	V	8	0.27	V	sublist-bf	V	30	0.21	V
mergesort	V	171	0.28	V	subset-bf	V	21	0.23	V
mergesort-oi	V	54	0.28	V	subset-fb	V	14	0.26	V
mergesort_variant	V	15	0.23	V	suffix-bf	V	9	0.25	V
minimum-fb	V	8	0.29	V	transpose2	V	6	0.28	V
naive reverse-fb	V	8	0.37	V	tree_member-bf	V	12	0.28	V

Table 1: Benchmark of non-terminating pure logic programs.

5 Conclusion and future work

We introduced a new approach to non-termination analysis of logic programs based on a finite, symbolic derivation trees for a moded queries. These symbolic trees represent the derivation trees of all concrete queries denoted by the moded query. To prove non-termination we look for a path in this symbolic derivation tree, that can be repeated infinitely often. This approach has been implemented in a non-termination analyzer *P2P*. This analyzer proves non-termination of all our benchmark programs.

In the future, we want to extend our approach in multiple ways. The first extension is an extension for typed goals instead of moded goals. The main advantage of this extension is that more complex classes of queries can be described. Also, this extension makes it possible to use information from existing non-failure analysis tools. Because many Prolog programs contain non-logical features, we want to extend our technique so that programs containing integer arithmetics can be analyzed. This can be done by using existing finite domain solvers to find domains for the integers for which the moded more general loop is indeed an infinite loop.

References

[1] R. N. Bol. *Loop checking in logic programming*. CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 1995.

[2] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[3] Étienne Payet and Frédéric Mesnard. Nontermination inference of logic programs. *ACM Transactions on Programming Languages and Systems*, 28(2):256–289, 2006.

[4] Yi-Dong Shen, Danny De Schreye, and Dean Voets. Termination prediction for general logic programs. K.U.Leuven report CW 536, 2009. Accepted for TPLP.

[5] Dean Voets and Danny De Schreye. A new approach to non-termiation analysis of logic programs. K.U.Leuven report CW 537, 2009. Accepted for ICLP 2009.

Report on the Termination Competition 2008

Johannes Waldmann (editor)
Fakultät Informatik, Mathematik und Naturwissenschaften,
Hochschule für Technik, Wirtschaft und Kultur, Leipzig, Germany
http://www.imn.htwk-leipzig.de/~waldmann/

Abstract

The Termination Competition is a yearly event where software for proving termination automatically is applied to a variety of termination problems. The fifth Termination Competition was hosted by the Computational Logic group at the University of Innsbruck, Austria, and took place in November 2008. We present the results of this competition, and some conclusions.

1 Introduction

The *Termination Competition* is an annual event to showcase and compare software that automatically proves termination of rewriting systems and logic and functional programs.

The goal of the competition is, among the community, to encourage research for new termination techniques, as well as efficient implementation and combination of know techniques; and at the same time to show outside the community that automated termination provers are mature, reliable and ready to be used in applications.

The precursor of the current termination competition was an exhibition of termination provers, organized by Albert Rubio, during the Workshop on Termination 2003 in Valencia. There, the participants decided to make this a regular event, using specific sets of termination problems, a specific procedure for submitting provers and running them on an execution platform, as well as a unified presentation of results on the web.

Past termination competitions were run by Claude Marche and Albert Rubio in 2004 [1], and by Claude Marche and Hans Zantema in 2005, 2006, and 2007 [3, 4, 2]. Since 2008, the competition is hosted by the Computational Logic Group at the University of Innsbruck. The host provides a hardware and software platform for the automated submission and execution of termination provers. See Section 2 for technical detail.

A *Termination problem data base* (TPDB) was established in 2003, based on termination problems collected from publications in the field, and growing continually by submissions of new termination problems, mostly by authors of termination provers, but also from applications areas. E.g., TPDB contains most of the functions from the Haskell Prelude. In Section 3 we give some detail on the semantics and current contents of TPDB.

The termination competition of 2008 was executed in three stages at the end of the year (and beginning of the next year), and this paper gives an account of the participants (in Section 4) and results (following sections). In particular we discuss some recent developments: 2008 was the first competition that included automated investigations of derivational complexity (see Section 9), and the second competition with (optional) automated verification of termination proofs (see Section 10). We conclude with some comments on the future of the competition in Section 11.

Strategic decisions for the competition and its future are made by a steering committee, consisting of representatives of the participating research groups. Developers and users of automated termination provers subscribe to the `termtools` mailing list, and there is a central information resource for the termination community `http://termination-portal.org/`, hosted by Lehr- und Forschungsgebiet Informatik 2 of RWTH Aachen.

This report is based on contributions by Simon Bailey (Platform), Frederic Blanqui (Certification), Jürgen Giesl (LP/FP/SRS/TRS), and Georg Moser (Complexity). The views expressed in Section 11 are the editor's.

2 Platform

After deciding to move the competition to Innsbruck, the Computational Logic Group in Innsbruck designed and implemented a platform for managing and running the competition. The main capabilities required were:

- Author-driven submission of termination tools
- Automated competition scheduling and running
- Web-interface for viewing results

The first version of the competition platform consists of two components: the web- interface itself and a scheduler with which the web-interface interacts. The competition run in November 2008 showed that the platform per se is robust and fulfils the requirements, but also has a lot of room for improvement.

2.1 Software Environment

The platform was programmed using JavaEE, the web-framework used is JBoss Seam. The complete interface runs on a dedicated JBoss server on the competition server. The scheduling component is also written in Java and uses Quartz as the job management framework. The database back-end is a PostgreSQL server.

2.2 Hardware Platform

The competition platform currently runs on a Sunfire x4600 with 8 Dual-Core AMD Opteron CPUs and 64GB of RAM. This system is a 64-bit system and offers tools the possibility of implementing and using parallel algorithms as compared to the previous versions of the competition which were limited to single CPUs.

2.3 Current Development

Currently, two new components for the execution platform are in development: a component allowing users to submit custom-built experiments on the reference competition hardware and a component which will allow statistical analysis of the results from previous competitions.

3 Termination Problems

We give an overview of the kind of termination problems that are present in the Termination Problem Data Base (TPDB), and are being used in competitions.

In the most general view, each entry in the data base specifies a relation. The task of the termination prover is to analyze this relation.

3.1 Specifying Computations

The relation describes one step of the computation in the following models:

- a rewriting system (given by a set of rewrite rules),
- a logic program (given in Prolog notation),

- a functional program (given in Haskell notation).

These relations are classified by additional criteria. For term rewriting systems, we have the syntactic criterion on the signature:

- string rewriting (all function symbols are unary),
- term rewriting (arbitrary signatures).

Then, several variants are used in defining the relation from the set(s) of rules. The rewrite relation can be restricted by *strategies*:

- no strategy (reduce any redex), • innermost, • outermost,
- context-sensitive (using replacement maps).

The relation can also be restricted by prescribing *start terms*.

- no restrictions,
- start term is constructor-based. (The resulting complexity notion is called *run time* complexity.)
- start term is a Prolog query, or a Haskell expression. (This is mandatory for the respective categories.)

The relation $\to_1$ defined by one set of rules can be combined with some other relation $\to_2$:

- no combination (just one relation $\to_1$)
- *relative* rewriting: consider $\to_1 \circ \to_2^*$,
- *equational* rewriting: consider $\to_1 \circ \leftrightarrow_2^*$.

  In particular, $\leftrightarrow_2$ may encode commutativity, associativity or idempotency of function symbols, and this is called “rewriting modulo theories”.

This gives a huge design space, and not all combinations are actually present (and some would have no reasonable semantics).

3.2 Properties of Computations

For relations that are given by the above specifications, two types of questions are considered by the provers:

- *Termination*: there is no infinite derivation
- if so, what is the *derivational complexity*? (the maximal length of a derivation, as function of the size of the start configuration)

The answers produced by the provers (YES, NO, or complexity bounds) are to be accompanied by a proof that is given in a form

- that can be verified by a mechanical proof checker,
- or at least be believed after human inspection.

We remark that not all of the 2×2 combinations of the above features do occur at present. The ultimate goal is indeed that a prover’s output is as detailed as possible (i.e., it contains complexity information), and as trustworthy as possible (i.e., mechanically verifiable) but this is certainly a long-term process.

3.3 Current Contents of TPDB

The termination competition 2008 used version 5 of the Termination Problems Data Base. Ignoring the refinements discussed above (strategies, theories, etc.) its summarized contents is:

combined categories	FP	LP	SRS	TRS
number of problems	1676	351	777	2036

.

The data base is freely available from `http://termination-portal.org/wiki/TPDB`.

4 Participants of the Competition

A termination prover could register for a subset of the categories, and was then applied to all problems from the respective categories.

We list here the provers and their categories. In some cases, specific versions (per category) were used. We ignore these in this overview, and give a detailed description in the following sections.

- AProVE (Automated Program Verification Environment)
  - developed at RWTH Aachen, Germany by Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Swiderski, Carsten Fuhs, Carsten Otto et al.
  - home page: `http://aprove.informatik.rwth-aachen.de`
  - categories: FP, LP, SRS and TRS both non-certified and certified
- CaT (Complexity and Termination)
  - developed at LFU Innsbruck, Austria, by Martin Korp, Christian Sternagel and Harald Zankl
  - home page: `http://cl-informatik.uibk.ac.at/software/cat/`
  - categories: Complexity
- Cime3 (Complet*i*on Modulo E)
  - developed at Cédric (CNAM/ENSIIE) and LRI (Univ. Paris-Sud, CNRS), France, by Évelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, Xavier Urbain,
  - home page: `http://a3pat.ensiie.fr/pub/index.en.html`
  - categories: SRS and TRS certified
- Jambox (Java Matchbox)
  - developed at Vrije Universiteit Amsterdam, Netherlands, by Jörg Endrullis
  - home page: `http://joerg.endrullis.de/`
  - categories: SRS and TRS non-certified
- matchbox
  - developed at HTWK Leipzig, Germany, by Johannes Waldmann,
  - home page: `http://dfa.imn.htwk-leipzig.de/matchbox/`
  - categories: SRS and TRS certified
- nonloop

  - developed at HTWK Leipzig, Germany, by Martin Oppelt,
  - home page: `http://dfa.imn.htwk-leipzig.de/nonloop/`
  - categories: SRS non-certified

- Polytool (Proving Termination Automatically based on Polynomial Interpretations)
  - developed at KU Leuven, Belgium, by Manh Thang Nguyen and Danny De Schreye, Peter Schneider-Kamp and Jürgen Giesl,
  - home page: `http://www.cs.kuleuven.be/~manh/polytool/`
  - categories: LP

- $\mathsf{T_CT}$ (Tyrolean Complexity Tool)
  - developed at LFU Innsbruck, Austria, by Martin Avanzini, Georg Moser, Andreas Schnabl,
  - home page: `http://cl-informatik.uibk.ac.at/software/tct/`
  - categories: Complexity

- $\mathsf{T_TT_2}$ (Tyrolean Termination Tool 2)
  - developed at LFU Innsbruck, Austria, by Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp
  - home page: `http://colo6-c703.uibk.ac.at/ttt2`
  - categories: SRS and TRS non-certified

In the following sections we report on the results of the competition, by category.

5 Term Rewriting

In these categories, the tools try to prove or disprove termination of term rewrite systems. Unfortunately, this year several of the well-known tools in the area did not participate. In the future, one should try to motivate their authors to join this category again. Moreover, one should also encourage the tools to join as many of the different TRS-sub-categories as possible.

5.1 Full Termination of Term Rewriting

In the category "*TRS Standard*", one regards ordinary TRSs. Here, three tools participated on 1391 examples. The winner was AProVE which could prove termination for 995 TRSs and disprove termination for 231 examples. The second best tool was $\mathsf{T_TT_2}$ which proved termination for 792 examples and disproved it for 178 TRSs. Finally, Jambox proved termination of 750 examples and disproved it for 60 TRSs. New techniques employed by the tools in this competition include polynomial interpretations that also allow functions like "maximum" and "minimum" as well as rational polynomial interpretations (where the search for the interpretations is based on SAT-solving).

It is remarkable that $\mathsf{T_TT_2}$ was using at most 5 seconds per problem, and its total time (for YES answers) is an eighth of the winner's. The authors wanted to show that even with severely limited resources, a considerable score could be achieved.

5.2 Innermost Termination of Term Rewriting

The categories "*TRS Innermost*" and "*TRS Outermost*" regard term rewriting under the innermost resp. the outermost evaluation strategy. In the innermost category, only AProVE participated (proving innermost termination of 241 examples and disproving it for 4 examples out of 358). Here, due to new techniques developed for disproving innermost termination, it would be interesting in the future to try to also disprove innermost termination for the other TRS-examples that are known to be non-terminating for full rewriting.

5.3 Outermost Termination of Term Rewriting

In the outermost category, a competition took place a month later than the rest of the competition. Here, several new results had been developed. In this category, JamboxGoesOut was the winner for proving termination due to its new transformation from outermost to context-sensitive rewriting. It proved outermost termination for 72 out of 291 examples (but it did not disprove outermost termination for any example). Concerning disproving outermost termination, $\mathsf{T_TT_2}$ was the winner due to its new transformation from outermost to innermost rewriting. It disproved outermost termination for 158 examples (but it did not prove outermost termination for any example). The remaining two participants were TrafO (which proved 47 and disproved 30 examples) and AProVE (which proved 6 and disproved 21 examples).

5.4 Termination Modulo Theory

Here, one proves termination modulo AC. Unfortunately, only AProVE participated and proved termination for 57 and disproved it for 2 out of 71 examples.

5.5 Relative Termination

Here, one proves relative termination. Only Jambox participated and proved termination for 24 out of 40 examples.

The low number of participants seems strange. It is safe to assume that each termination prover contains some mechanism for reducing termination problems by "removing rules". By making this mechanism explicit, one should obtain a prover for relative termination with little extra work. In turn, progress in relative termination methods and implementations would benefit all provers.

5.6 Context-Sensitive Termination of Term Rewriting

Here, a lot of new developments took place in the last years. In 2006, a dependency pair approach for context-sensitive rewriting was developed and implemented in Mu-Term. In 2008, this approach was improved considerably to a context-sensitive dependency pair *framework*. This framework is implemented in AProVE which was the only participant in this category this year. This is a pity, since both Mu-Term and Jambox also support context-sensitive rewriting (see the outermost category above). In the future, these other tools should participate in this category as well. In 2008, AProVE proved termination of 94 out of 109 examples.

6 String Rewriting

These categories are similar to the TRS categories, but here one only regards TRSs where the function symbols have arity 1. Again, it is a pity that many of the tools that have been working on string rewrite systems in the past did not participate in the competition in 2008.

The previous winner *Matchbox* did participate in the certified string rewriting category instead. Recent progress in certification has narrowed the gap between the results in these two categories, see the discussion in Section 10.

For full termination of SRSs, the tools $\mathsf{T_TT_2}$ and AProVE that used to be tailored more to general TRSs than to SRSs were the most powerful ones. It is unclear whether this is due to the fact that tools which are more tailored to SRSs did not participate in this category in 2008 or whether the TRS-tools really improved so much on SRSs.

The winner for proving termination in this category was $\mathsf{T_TT_2}$ which proved termination for 512 SRSs and disproved it for 40 followed by AProVE which proved termination for 501 and disproved it for 22 SRSs. Jambox proved termination of 252 SRSs. For disproving termination of SRSs, the new tool nonloop was the winner by disproving termination of 92 SRSs. In particular, this tool is the first to also prove non-termination of some non-looping SRSs, by enumerating patterns in (forward) closures, and checking whether they are recurrent. $\mathsf{T_TT_2}$ finds loops by encoding derivations as a Boolean constraint system.

For relative termination of SRSs, only Jambox participated and proved termination for 32 out of 42 examples.

7 Logic Programming

In this category, the tools have to prove termination of Prolog programs for given sets of queries. Although there exist many tools for proving (and even disproving) termination of logic programs, only two tools joined the competition 2008. In the future, we should try to improve this and motivate the authors of the many other tools to participate as well.

Essentially, there are two approaches to prove termination of logic programs. Either one proves termination directly on the basis of the logic program or one transforms the logic program into a term rewrite system and proves termination of the TRS afterwards. In the competition, the tool Polytool used the direct approach and the tool AProVE used the transformational approach. The main feature of Polytool is that it contains several termination techniques that originate from term rewriting, but which were adapted to logic programs. The main feature of AProVE is that it can handle arbitrary logic programs via transformation (not just “well-moded” logic programs as in earlier transformational tools). Both Polytool and AProVE use SAT-solving when searching for suitable orders (in fact, Polytool uses AProVE as a back-end for this search).

In the competition, AProVE was a little more powerful than Polytool (AProVE could prove termination of 246 examples and Polytool proved termination for 238 examples out of 349). This is most likely due to the fact that AProVE offers more termination techniques than Polytool. But of course, these termination techniques might also adapted be to logic programming and integrated into Polytool in the future.

8 Functional Programming

In this category, the goal is to prove termination of Haskell programs for a given class of *start terms*. The data base currently consists of 1676 termination problems taken from standard libraries of the Hugs-interpreter. So this category is currently the only one in the termination competition where the termination problems are indeed taken from existing libraries in existing programming languages.

To indicate the importance of putting termination analysis into practical applications, this category was included in the competition although only one tool (AProVE) participated. Thus, currently this category is more a demonstration of the state of the art in this domain and it should serve as a motivation for other tools to join this category as well.

In the competition, AProVE could show termination of 1294 problems and it could disprove termination for 19 problems.

9 Complexity

In this year's termination competition for the first time a *complexity category* was present. It is a challenging topic to automatically determine upper bounds on the complexity of rewrite systems. Here complexity is measured as the maximal length of derivations, where either no restrictions on the initial terms are present (*derivational complexity*) or only constructor based terms are considered (*runtime complexity*). Additionally one distinguishes between complexities induced by full rewriting as opposed to complexities induced by specific strategies, as for example innermost rewriting. In this year this variety was represented trough four sub-categories: *derivational complexity–full rewriting*, *derivational complexity–innermost rewriting*, *runtime complexity–full rewriting*, and *runtime complexity–innermost rewriting*.

As test-bed we used the TRSs of the categories *TRS Standard* and *TRS Innermost* where TRSs that occur in both categories where considered once. For the sub-category *derivational complexity–full rewriting*, only *non duplicating* systems were tested as the current focus of the complexity category is on polynomial derivational/runtime complexity.

As test-bed the collection of all "standard" TRSs together with all TRSs with flag "(STRATEGY INNERMOST)" were used. Here TRSs in the current TPDB which only differ by the flag are only considered once. However for the sub-category *derivational complexity–full rewriting*, only *non duplicating* systems were tested as the current focus of the complexity category is on polynomial derivational/runtime complexity. In sum this amounts to 1404 examples for the full test-bed and 446 TRSs for the restricted test-bed.

In the sub-categories for derivational complexity, CaT and TCT ran a competition, which was won by CaT. In the sub-categories for runtime complexity, TCT ran as a demonstration.

The most challenging part in the design of this category was the definition of the competition semantics. While it was clear from the beginning that the current focus should be on *polynomial bounds*, it turned out to be not so easy to suit the existing competition semantics suitably to complexity analysers: while for the "standard" categories it has always been understood that the power of a tool should be measured by the number of successful runs the tool manages, the situation is not so clear for complexity analysis. What is a "successful run", if the prover is supposed to measure the complexity of a given TRS?

An ad-hoc solution would have been to test for polynomial bounds and the competing tools simply answer "yes" if this bound is verified. However this somehow blurs the potential of complexity analysis and does not really represent the research in this area. Hence we decided to extend the output certificate so that an (asymptotic) complexity function is rendered as output. Thus the tool providing the more accurate output (either as a lower or as an upper bound) should be considered the winner for the specific TRS tested.

In this year's competition only upper bounds have been tested as not much is currently known on how to (automatically) test for lower bounds on the complexity. Still the output format is already equipped to handle such an analysis as well. The following grammar is used:

$$\begin{aligned} \mathsf{S} &\to \mathtt{NO} \mid \mathtt{MAYBE} \mid \mathtt{YES}(\mathsf{F},\mathsf{F}) \mid \mathtt{YES}(?,\mathsf{F}) \mid \mathtt{YES}(\mathsf{F},?) \\ \mathsf{F} &\to \mathtt{O}(1) \mid \mathtt{O}(n^m) \mid \mathtt{POLY}\,, \end{aligned}$$

where m is a natural number.

As said, we currently focus on (polynomial) upper bounds. Firstly, for each TRS the competing tools are ranked, where the tool with the lowest upper bound wins with respect to the given TRS. For each place in this ranking, points are given. Finally the obtained points are summed up and the tool that achieved the most points in total wins the the competition, see `http://termination-portal.org/wiki/Complexity` for more details. This more refined competition semantics caused some overhead in its implementation, but it has proven its functionality in the execution.

Let us come back to the actual competition between CaT (short for Complexity And Termination) and TCT (short for Tyrolean Complexity Tool). Both tools are partially built up on basic libraries of TTT2 which explains the common output format. Both tools implement direct termination methods, sometimes restricted, that induce polynomial derivational complexity; for example *match-bound techniques*, *triangular matrix interpretations*, and *root labeling* are implemented.

In addition CaT uses the *arctic matrix method* as another direct termination technique and TCT incorporates transformation techniques like *weak dependency pairs*, *weak dependency graphs*, *polynomial path orders*, and *finite semantic labeling*. Note that the mentioned transformation techniques are only used in the sub-category "runtime complexity". The main reason why CaT won amounts (i) to a cleverer strategy that uses (triangular) matrix interpretations also for higher dimensions and (ii) the use of the arctic matrix method.

For the future it seems to be of utmost importance to increase the number of participants. Moreover fresh methods are required in all sub-categories to extend the number of TRSs that can be analysed, with respect to lower and upper bounds, respectively. Clearly at the moment the TPDB is not an ideal test-bed for complexity analysis, but we expect that this will change due to addition of new examples to the TPDB in the course of future competitions.

For the moment we suggest to keep the focus on *polynomial* complexity analysis. Within this focus there are still a vast number of challenging problems, foremost the proper incorporation of incremental termination techniques like the dependency pair framework. Currently the extension to higher complexity classes and/or certification, although envisaged, feels premature.

10 Certification

In the certified categories of the termination competition 2008, the termination proofs are expressed in a formal language and checked by machine. All participants used Coq as the underlying proof checker, and one of two libraries that contain formalized knowledge on rewriting and termination:

- CoLoR (Coq Library on Rewriting and termination)
  - developed at INRIA, France, and TU Eindhoven, Netherlands, by Frederic Blanqui and Adam Koprowski,
  - home page: `http://color.inria.fr/`
- A3PAT (Assister Automatiquement les Assistants de Preuve avec des Traces)
  - developed at Cedric, INRIA Sophia-Antipolis, LaBRI, LRI-PCRI, France, by Xavier Urbain et al.
  - home page: `http://a3pat.ensiie.fr/`

Matchbox uses CoLoR, Cime3 uses A3PAT, and AProVE submitted three versions: one that use each library alone, and one running these two in parallel.

Of the total 1391 termination problems for term rewriting, AProVE/CoLoR produced 558 certified proofs, followed by AProVE/mixed with 520 and Cime3/A3PAT with 485.

Of the total 732 termination problems in string rewriting, Matchbox/CoLoR produced 466 certified proofs, followed by AProVE/CoLoR with 406 and AProVE/mixed with 371.

Let us compare the results with the certified category of the previous competition. The problems used there were a strict subset of this years'.

For term rewriting, the percentage of solved (by the winner) problems goes up from 36 % to 40 %.

This increase is mainly due to the availability of the certified arctic matrix method.

In string rewriting, certification was first applied in 2008. The percentage of uncertified solutions in 2007 is 65 %, and this is nearly equal to the percentage of certified solutions in 2008, namely 64 %.

Let us now compare the number of certified and uncertified proofs (obtained by the respective winners) in 2008. In term rewriting, the certified winner got 56 % of the number of the uncertified winner, while in string rewriting, the corresponding ratio is 93 %.

This indicates that a state-of-the-art termination prover for term rewriting needs to rely on techniques that are (not yet) formally certified "half of the time", while in string rewriting, this is necessary only in much fewer cases.

Current developments in certified termination are

- to provide alternative certification libraries and back-ends (e.g., using Isabelle),
- to improve efficiency (e.g., by having the termination certificates checked by a Haskell or ML program that is derived automatically from an Isabelle or Coq text),
- to make more methods available for the standard categories (e.g., sharper dependency graph approximations for proving termination, and certification of loops for proving non-termination),
- to extend certification to other competition categories (e.g., relative termination).

The long-term goal is to have *all* statements produced by termination provers in any category verified automatically.

We note that with the current execution platform, the actual certification of termination proofs has a serious shortcoming: the platform checks validity of a proof but not that it proves the right theorem. Indeed it happened that a prover was working on different rewriting systems than intended because of some trivial programming error in the the input parser.

It is difficult to handle such problems because different termination provers do not even agree on the formalization of the statements and theorems, since they are using different libraries and proof checkers. The Rainbow project aims to solve this problem by providing a common format for termination proofs that is independent of the prover and of the verifier. Currently, Rainbow is used as a high-level interface to the CoLoR library.

11 Discussion

We give a few observations and derive some suggestions for future development. These are necessarily biased, and need further discussion within the community

The Termination Competition was moved successfully to the new site. The new execution platform provides better process automation, greater computing power and more flexibility.

Still, the competition faces several challenges:

- the competition needs to be more visible outside the community,
- inside the community, the competition should be made more interesting, to win new participants, and re-activate the retired ones.

- the available computing power should be utilized better.

Let us briefly discuss the technical item first.

11.1 Multi-core

For decades now, the computing power of CPUs increased steadily. Laws of nature seem to limit the speed of instruction execution on an electronic device, but the computing power is still increasing because of CPUs containing increasing numbers of computing units (cores) that operate in parallel. It is a challenge for software to match the increased hardware capabilities.

The Termination Competition community realized that and encourages participating provers to employ parallelism. The execution service for the competition runs on a 16 core machine, and the prover can use 1 minute "wall clock" time per problem. In other words, using only one of the cores (as a classical single threaded program would do) is wasting 15/16 of the available resources.

Several termination provers indeed employed parallelism, among them AProVE and Matchbox. The basic idea is that branches of the proof search tree are explored in parallel. E.g., Matchbox is looking for standard and arctic matrix interpretations for the original and the reversed string rewriting system, all at the same time. The execution of each node in the search tree involves the production of a constraint system, its treatment by an external solver, and reading back the results.

While the external solver processes will be handled by the operating system, the production of their input seems more of a challenge, as this has to be done in the prover properly. Parallelism must therefore be handled by the run time environment of the implementation language of the prover: e.g., the Java virtual machine for AProVE, and the Glasgow Haskell runtime for Matchbox.

This does not seem to scale so easily, and a particular obstacle (for Matchbox) seems to be the current "stop the world" mode of memory management (allocation and garbage collection). It is hoped that there will be progress in models for conceptually high level parallelism with efficient run-time implementations, so that a low-level "parallelizing rewrite" of termination provers can be avoided.

Another way of avoiding programming efforts is to keep a single-threaded control structure for the termination prover, but rely on constraint solvers that exploit parallelism. Constructing such solvers is an area of ongoing research in the SAT and SMT community, but the publically available constraint solvers do not seem to reflect this at the moment.

11.2 Modular and re-useable software

To increase participation in future termination competitions, new entrants need help in passing the "entry barrier".

Current participants are therefore encouraged to publish re-usable libraries for common but "boring" tasks like parsing the problem input, printing proof output, applying standard transformations, and calling standard constraint solvers. Based on that, it is easier for new participants to concentrate on implementing fresh ideas.

E.g., in the complexity categories, the new participants CaT and $\mathsf{T_CT}$ use libraries from $\mathsf{T_TT_2}$, so they benefitted from previous implementation efforts. Continuing in this fashion, both tools are (partly) open source under the GNU Lesser General Public License, and their source code is available from the respective web pages.

11.3 Organisation

In 2007, the community decided to move from yearly competitions to "ongoing" competitions. The goal was to have more flexibility: it should be easier to update provers and show their results.

Yet, viewing all visible executions of provers on data base problems as competitions, creates management overhead: all decisions have to be approved by the competition committee. This slowed down the process, and that is the reason why "ongoing" competitions did not really happen.

This will be improved by the planned separation of the Competition from the Execution service. This is modelled after the Lib/Exec/Comp structure in the SMT (satisfiability modulo theory) community.

Then indeed the run time of the competition could be reduced, by restricting to certain benchmarks of problems. The competition then could be run during some conference, with results being announced at the end, increasing visibility. The ranking of participants (as applied in the Complexity category) could add more excitement. Runs over the full data base could still be done at different times.

To implement such changes, the competition steering committee needs to act much more efficiently than it does presently.

References

[1] Claude Marche and Albert Rubio. The rise of the tools. `http://www.lri.fr/~marche/termination-competition/2004/slides-1jun2004.ps`, 2004.

[2] Claude Marche, Johannes Waldmann, and Hans Zantema. The termination competition 2007. `http://www.imn.htwk-leipzig.de/~waldmann/talk/07/wst/competition/`, 2007.

[3] Claude Marche and Hans Zantema. Termination competition 2005. `http://www.lri.fr/~marche/termination-competition/2005/TC.ppt`, 2005.

[4] Claude Marché and Hans Zantema. The termination competition. In Franz Baader, editor, *RTA*, volume 4533 of *Lecture Notes in Computer Science*, pages 303–313. Springer, 2007. This is the report on the 2006 competition. `http://www.lri.fr/~marche/termination-competition/2006/reportCompetition2006.pdf`.

Equational Reasoning for Termination of Rewriting*

Harald Zankl and Aart Middeldorp
Institute of Computer Science
University of Innsbruck
6020 Innsbruck, Austria
{harald.zankl,aart.middeldorp}@uibk.ac.at

Abstract

In this note we suggest to use equational reasoning for obtaining finer approximations of dependency graphs.

1 Introduction

The dependency graph is an essential ingredient to make the dependency pair framework modular and powerful. This graph is not computable in general but sound approximations exist [1, 5, 7, 8, 10]. In this note we propose a new method based on equational reasoning to further improve existing approximations. We first recall the basic concepts of the dependency pair framework in Section 2. Then the relationship to equational reasoning is sketched in Section 3 before experimental results are reported in Section 4. Related ideas and possible future work is discussed in Section 5.

2 Dependency Pair Framework

We assume familiarity with term rewriting [2, 11] and the termination competition.[1] Next we recall the key concepts that are essential for the remainder of this note.

Definition 1. The nodes of the *dependency graph* $\mathsf{DG}(\mathcal{R})$ of a TRS $\mathcal{R}$ are the dependency pairs of $\mathcal{R}$ and there is an edge from node $s \to t$ to node $u \to v$ if there exist substitutions σ and τ such that $t\sigma \to_{\mathcal{R}}^{*} u\tau$.

To make use of this definition a corresponding DP processor is formulated.

Theorem 2. *The following processor [6, 7] is sound and complete. For a DP problem* $(\mathcal{P},\mathcal{R})$ *the processor returns* $\{(\mathcal{P}_1,\mathcal{R}),\ldots,(\mathcal{P}_n,\mathcal{R})\}$ *where* $\mathcal{P}_1,\ldots,\mathcal{P}_n$ *are the SCCs of* $\mathsf{DG}(\mathcal{R})$. □

The condition in Definition 1 is not decidable but sound approximations exist [1, 5, 7, 8, 10]. In the sequel we show that Waldmeister [4] allows a finer approximation of the condition from Definition 1.

3 Equational Reasoning

Next we sketch how Waldmeister can be used to get finer approximations of the dependency graph: The existence of substitutions σ and τ such that for two terms t and u with $t\sigma \to_{\mathcal{R}}^{*} u\tau$ clearly implies $t\sigma \leftrightarrow_{\mathcal{R}}^{*} u\tau$. The latter condition can be tested with Waldmeister.

Let $\mathcal{E}$ denote the set of equations resulting from removing the orientation of rules in $\mathcal{R}$. To test an edge $(s \to t, u \to v)$ from $\mathsf{DG}(\mathcal{R})$, the input for Waldmeister consists of $\mathcal{E}$ and the equation (called conclusion) $t = u$. The theorem prover performs a variant of ordered completion [3] until it succeeds in proving or refuting $t\sigma \leftrightarrow_{\mathcal{R}}^{*} u\tau$. Thus an edge $(s \to t, u \to v)$ in the dependency graph can be removed if Waldmeister manages to refute $t\sigma \leftrightarrow_{\mathcal{R}}^{*} u\tau$. In all other cases (Waldmeister proves the claim or does not terminate) nothing can be said about the edge.

*This research is supported by FWF (Austrian Science Fund) project P18763.

[1] http://termcomp.uibk.ac.at

```
NAME         various11
MODE         PROOF
SORTS        ANY
SIGNATURE    g: ANY ANY -> ANY
             1: -> ANY
             0: -> ANY
             h: ANY -> ANY
             F: ANY ANY ANY -> ANY
             f: ANY ANY ANY -> ANY
ORDERING     LPO g > 1 > 0 > h > F > f
VARIABLES    z,w,y,x : ANY
EQUATIONS    f(0, 1, x) = f(h(x), h(x), x)
             h(0) = 0
             h(g(x, y)) = y
             F(x, x, x) = F(x, x, x)
CONCLUSION   F(h(z), h(z), z) = F(0, 1, w)
```

Listing 1: Input to Waldmeister

```
Initial equations:
---------------------

(    1)  x1 = h(g(x2,x1))
(    2)  h(0) = 0
(    3)  f(h(x1),h(x1),x1) = f(0,1,x1)
(    4)  F(x1,x1,x1) = F(x1,x1,x1)
(    5)  eq(x1,x1) = true
(    6)  eq(F(h(x1),h(x1),x1),F(0,1,x2)) = false

Goals:
-------

(    1)  true ?= false
         using narrowing to prove F(h(x1),h(x1),x1) ?= F(0,1,x2)

Detected structure: Orkus
****************************************************************************
**************************** COMPLETION - PROOF ****************************
****************************************************************************
new rule:              1  F(h(x1),h(x1),x1) ? F(0,1,x2)
new rule:              2  h(g(x1,x2)) -> x2
new rule:              3  F(x1,x1,g(x2,x1)) ? F(0,1,x3)
new rule:              4  h(0) -> 0
new rule:              5  F(0,0,0) ? F(0,1,x1)
new rule:              6  f(h(x1),h(x1),x1) -> f(0,1,x1)
new rule:              7  eq(x1,x1) -> true
new rule:              8  f(0,0,0) -> f(0,1,0)
new equation:          1  f(0,1,g(x1,x2)) = f(x2,x2,g(x1,x2))

Refuted Goals:
No.  1: true ?= false, current true ?= false
        using narrowing to prove F(h(x1),h(x1),x1) ?= F(0,1,x2)

1 goal was specified, which was refuted.

Waldmeister states: System completed.
```

Listing 2: Output of Waldmeister

Example 3. Consider the TRS various/11[2] consisting of the following three rules

$$\begin{aligned} \mathsf{f}(\mathsf{0},\mathsf{1},x) &\to \mathsf{f}(\mathsf{h}(x),\mathsf{h}(x),x) \\ \mathsf{h}(\mathsf{0}) &\to \mathsf{0} \\ \mathsf{h}(\mathsf{g}(x,y)) &\to y \end{aligned}$$

admitting the dependency pairs

$$\begin{aligned} \mathsf{F}(\mathsf{0},\mathsf{1},x) &\to \mathsf{F}(\mathsf{h}(x),\mathsf{h}(x),x) \\ \mathsf{F}(\mathsf{0},\mathsf{1},x) &\to \mathsf{H}(x) \end{aligned}$$

Typical (combinations of) dependency graph approximations [1, 5, 7] (sometimes referred to as edg***) cannot figure out that no instance of $\mathsf{F}(\mathsf{h}(x),\mathsf{h}(x),x)$ rewrites to $\mathsf{F}(\mathsf{0},\mathsf{1},x)$ whereas our approach can. Thus this system can successfully be proved terminating due to an empty dependency graph. The input for above edge to Waldmeister is depicted in Listing 1 most of which is self explanatory. We stress some facts that we learned during our experiments:

- Although Waldmeister is used in its automatic mode, one has to specify a precedence for LPO which is ignored.
- The root function symbols from the **CONCLUSION** must appear among the equations to get consistent results. This explains the peculiar looking $\mathsf{F}(x,x,x) = \mathsf{F}(x,x,x)$.
- Waldmeister does not terminate quite frequently. Thus we impose a time limit of 0.1 seconds per edge.

When calling Waldmeister on the input from Listing 1 it produces the refutation shown in Listing 2.

4 Experimental Results

We integrated our approach into $\mathsf{T_TT_2}$ [9] with the help of Waldmeister. In our tests we considered the 1391 TRSs from the Termination Problems Data Base version 5.0. All tests have been performed on a server equipped with eight dual-core AMD Opteron® processors 885 running at a clock rate of 2.6 GHz and on 64 GB of main memory. A time limit of 5 seconds was enforced. Thus the configuration of $\mathsf{T_TT_2}$ and the test environment is similar to the latest termination competitions.

For Table 1 we compare the existing approximation (edg***) with our approach (wdg) in different settings. Here wdg just tests edges that could not be removed by edg***. The column none indicates that no additional DP processors are employed and for competition the refinement is integrated within the strategy used by $\mathsf{T_TT_2}$ in the 2008 edition of the termination competition. The numbers in the table indicate how many systems could be proved terminating. With edg*** 60 systems can be shown terminating due to a lack of SCCs in the approximated dependency graph. This number can be increased to 71 if additionally wdg is applied. Most remarkably this simple method allows to prove two systems (secret06/tpa01 and various/11) which $\mathsf{T_TT_2}$ could not show terminating in the November 2008 competition. These systems have been solved by other tools (AProVE, Jambox, and TPA) before, based on semantic labeling and/or narrowing. The results in the right part of Table 1 do not look so convincing at first since the number is worse for wdg. The reason is due to the small timeout of 5 seconds in $\mathsf{T_TT_2}$'s competition mode. Currently we lose three systems (two from the TRCSR directory and various/14)

[2]Systems from the Termination Problems Data Base are written in sans − serif font.

Table 1: Experiments for 1391 TRSs.

	none	competition
edg***	60	779
wdg	71	778

when additionally allowing wdg which could be handled (closely) within 5 seconds beforehand. Since wdg has to call Waldmeister for each single edge in the dependency graph one should not forget about the additional costs. However, we anticipate that further experimentation will provide the necessary insights to build an efficient strategy including wdg.

5 Related and Future Work

Apart from the known approximations for dependency graphs [1, 5, 7, 10] recently some more refinements evolved. Korp and Middeldorp [8] use tree automata techniques to remove edges from the dependency graph. However, they cannot solve systems like various/11 since their approach must first linearize right-hand sides of rules. Zankl and Middeldorp [12] combined (increasing) interpretations with cycle analysis but are also unable to solve Example 3.

Concerning future work we plan to investigate whether other tools than Waldmeister can be used for our needs. Furthermore we want to extend our approach to innermost termination. Here the main benefit is that one can restrict the input to Waldmeister to the usable rules of $\mathcal{R}$ instead of $\mathcal{R}$.

References

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

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

[3] L. Bachmair, N. Dershowitz, and D. Plaisted. *Completion without Failure*, chapter 1 of volume II in Resolution of Equations in Algebraic Structures, pages 1–30. Academic Press, 1989.

[4] J.-M. Gaillourdet, T. Hillenbrand, B. Löchner, and H. Spies. The new WALDMEISTER loop at work. In F. Baader, editor, *CADE*, volume 2741 of *Lecture Notes in Computer Science*, pages 317–321, 2003.

[5] J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In B. Gramlich, editor, *FroCoS*, volume 3717 of *Lecture Notes in Artificial Intelligence*, pages 216–231, 2005.

[6] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

[7] N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1-2):172–199, 2005.

[8] M. Korp and A. Middeldorp. Beyond dependency graphs. In R. Schmidt, editor, *CADE*, 2009. To appear.

[9] M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In R. Treinen, editor, *RTA*, volume 5595 of *Lecture Notes in Computer Science*, pages 295–304, 2009.

[10] A. Middeldorp. Approximations for strategies and termination. *Electronic Notes in Theoretical Computer Science*, 70(6):1–20, 2002.

[11] TeReSe. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 2003.

[12] H. Zankl and A. Middeldorp. Increasing interpretations. *Annals of Mathematics and Artificial Intelligence*, 2009. To appear.

Combinatorial Problems by Termination of Rewriting

Hans Zantema[1,2]

[1] Department of Computer Science, TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, email: H.Zantema@tue.nl
[2] Institute for Computing and Information Sciences, Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands

Abstract. We give an overview of some combinatorial problems that can be transformed to termination problems on rewriting. Some of them then can be solved by applying state-of-the-art-tools for proving termination automatically, others can not, in this way creating new challenges for future versions of termination tools.

1 Collatz problem

The Collatz problem is called after L. Collatz who posed this problem in 1937; it is also called Syracuse problem or $3n+1$ problem. The problem is whether for every positive natural number a_0 there exists n such that $a_n = 1$, where

$$a_n = \begin{cases} a_{n-1}/2 & \text{if } a_{n-1} \text{ is even} \\ 3a_{n-1}+1 & \text{if } a_{n-1} \text{ is odd} \end{cases}$$

for $n = 1, 2, 3, \ldots$. It is conjectured that this holds for every positive natural number a_0; it has been proved in Mathematica that it holds for all positive natural numbers $< 10^{15}$ ([7]). An overview on this open problem is given in [4].

Obviously the problem is equivalent to termination of the system of computation rules after removing the rule for a_{n-1} being 1. So it is not surprising that this problem can be expressed as a termination problem. A way to do so by term rewriting was given in [5]. A way to do so by string rewriting was given in [8]: there it was proved that the Collatz property holds if and only if C is terminating. Here C is the SRS consisting of the seven rules

$$\begin{array}{lll} h11 \to 1h & 11hb \to 11sb & h1b \to t11b \\ & 1s \to s1 & 1t \to t111 \\ & bs \to bh & bt \to bh \end{array}$$

over the five symbols b (blank), h (half), s (shift), t (triple) and 1. This SRS can be seen as a kind of a Turing machine with an elastic tape: the tape alphabet consists of 1 and b where b is the blank symbol, and h, s and t are the machine states. By h the head is shifted to the right while contracting two tape cells to one; by s and t the head is shifted to the left while by t every cell is tripled.

As expected, termination of C cannot be proved. We agree with Paul Erdös that "mathematics is not yet ready for such problems".

The above problem admits several variants which are simple in mathematical sense, but for which standard termination tools fail. For instance, consider the following game. Start by a arbitrary number. If it is of the shape $2n$ for $n > 0$ it is replaced by n. If it is of the shape $2n + 1$ for $n \geq 0$ it is replaced by $4n$. For an even number > 1 it strictly decreases after one step, for an odd number it strictly decreases after two steps. So it is obviously terminating. This game can be encoded in the following SRS

$$\begin{array}{lcr}
h11 \to 1h & 1hb \to 1sb & h1b \to tb \\
 & 1s \to s1 & 1t \to t1111 \\
 & bs \to bh & bt \to bh
\end{array}$$

It is not hard to show that any infinite reduction in this SRS gives rise to an infinite play in the above game, so this SRS is terminating. However, this can neither be proved by AProVE [1] nor by TTT2 [3]. Here the mathematical argument is much simpler than for `tpdb-5.0/SRS/Zantema/z079.srs` in which $2n$ is replaced by $3n$ until all factors 2 have been removed.

Another variant

$$\begin{array}{lcr}
h11 \to 1h & 11hb \to 11sb & h1b \to t11b \\
 & 1s \to s1 & 1t \to t11 \\
 & bs \to bh & bt \to bh
\end{array}$$

describes the terminating game in which $2n$ for $n > 1$ is replaced by n and $2n+1$ for $n \geq 0$ is replaced by $2n+2$. Termination of this SRS can be proved by TTT2 [3], but not by AProVE [1].

2 Braids

A result from Garside [2] states that starting from two arbitrary braids one can always continue by only positive moves on both braids such that the resulting braids are equivalent. A rewriting approach to this problem can be found in [6], page 467 - 474. Essentially this is a confluence problem, in a setting where local confluence is straightforward. It can be shown that this confluence problem is equivalent to termination of the process of filling the confluence diagram by applying the local confluence property. This process can be expressed by string rewriting, by which the whole result boils down to termination of an SRS. For braiding with n strands the corresponding SRS is over the $2n - 2$ symbols $1, 2, \ldots, n-1, \overline{1}, \overline{2}, \ldots, \overline{n-1}$, and consists of the rules

$$\begin{array}{ll}
i\,\overline{i} \to \epsilon & \text{for } i = 1, \ldots, n-1 \\
i\,\overline{j} \to \overline{j}\,\overline{i}\,j\,i & \text{for } i, j = 1, \ldots, n-1,\ |i-j| = 1 \\
i\,\overline{j} \to \overline{j}\,i & \text{for } i, j = 1, \ldots, n-1,\ |i-j| > 1
\end{array}$$

For three strands, that is $n = 3$, termination of this SRS is easily proved by standard termination tools. For higher n, however, tools like AProVE [1] and TTT2

[3] fail. We propose the cases with $n > 3$ as a challenge for future termination provers.

For $n = 4$ and $n = 5$ a termination proof can be given as follows. By a finite automaton construction a finite set W of words over $1, 2, \ldots, n-1$ can be found such that

– the singleton string i is in W for every $i = 1, 2, \ldots, n-1$, and
– for every $w \in W$ and every $i = 1, 2, \ldots, n-1$, the string $w\overline{i}$ rewrites to $\overline{u}w'$ for some $w' \in W$ and a string $\overline{u}$ over $\overline{1}, \overline{2}, \ldots, \overline{n-1}$.

For $n = 4$ the minimal set W satisfying these properties consists of 35 strings, for $n = 5$ it consists of 712 strings. Using the properties of W and writing $\overline{u}, \overline{v}$ for strings over $\overline{1}, \overline{2}, \ldots, \overline{n-1}$ one proves by induction on the length of $\overline{u}$ that for every $w \in W$ the string $w\overline{u}$ rewrites to normal form $\overline{v}w'$. As $i \in W$ for every $i = 1, 2, \ldots, n-1$, any string of the shape $i\overline{u}$ rewrites to normal form. Using this property one proves by induction on the number of non-over-lined symbols in the starting string that every string rewrites to normal form. So the SRS is weakly normalizing. Since it is orthogonal, termination can be concluded.

3 Bowls and beans

Suppose an unbounded sequence of bowls each containing a number of beans, in total a finite number of beans. We have the following rule:

> If a bowl contains two or more beans, pick any two beans in it and move one of them to the bowl on its left and the other to the bowl on its right.

The problem is to show that this cannot go on forever, and the resulting configuration is independent of the chosen strategy. This problem has been described and solved by Vincent van Oostrom in
`http://www.phil.uu.nl/~oostrom/publication/pdf/bowls3.pdf`.
The main problem is proving termination of this process; proving strategy-independence of the result easily follows from the diamond property that is easily checked.

The termination problem data base TPDB contains several encodings of this problem in rewriting: `tpdb-5.0/SRS/Zantema06/beans`i`.srs` for i running from 1 to 7. These encodings also contain generalizations: at least two beans are taken from one of the bowls, and this group of beans is split into two non-empty parts; one of them is moved to the left and the other is moved to the right. Also a bowl containing at least two beans may create a new bowl, where the beans are divided into two non-empty groups over the original and the new bowl. For all of the 7 encodings termination is proved fully automatically by both AProVE and TTT2.

Harder is this problem where there are n bowls put in a circle, and the same rule applies. It is easily seen that for a total number $\geq n$ of beans this can go on forever. We conjecture that for a total number k of beans with $k < n$, this will always terminate.

References

1. J. Giesl et al. Automated program verification environment (AProVE). Available at `http://aprove.informatik.rwth-aachen.de/`.
2. F. A. Garside. The braid group and other groups. *Quarterly Journal of Mathematics*, 20:235–254, 1969.
3. M. Korp, C. Sternagel, H. Zankl, and Aart Middeldorp. Tyrolean termination tool 2. In R. Treinen, editor, *Proceedings of the 20th Conference on Rewriting Techniques and Application s (RTA)*, Lecture Notes in Computer Science. Springer, 2009. Tool available at `http://colo6-c703.uibk.ac.at/ttt2/`.
4. J. C. Lagarias. The $3n + 1$ problem and its generalizations. *Amer. Math. Monthly*, 92:3–23, 1985.
5. P. Lescanne. Rewrite orderings and termination of rewrite systems. In *Proceedings Mathematical Foundations of Computer Science*, volume 520 of *Lecture Notes in Computer Science*, pages 17–27. Springer Verlag, 1991.
6. Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.
7. I. Vardi. *Computational recreations in Mathematica*. Addison-Wesley, 1991.
8. H. Zantema. Termination of string rewriting proved automatically. *Journal of Automated Reasoning*, 34:105–139, 2004.