Workshop Proceedings

WST 2006

Eighth International Workshop on Termination

Seattle, WA, USA

15–16 August 2006

Alfons Geser and Harald Søndergaard, editors

WST 2006 — Proceedings of the Eighth International Workshop on Termination, Seattle, WA, USA, 15–16 August 2006. Alfons Geser and Harald Søndergaard, editors.

These proceedings are available from the Computing Research Repository, `http://www.acm.org/corr/`.

Preface

This volume contains the proceedings of the *Eighth International Workshop on Termination (WST 2006)*, held in Seattle, WA, August 15–16, 2006, as part of the 2006 Federal Logic Conference (FLoC). The sponsoring conference was the Seventeenth International Conference on Rewriting Techniques and Applications (RTA 2006).

There were 15 submissions to WST 2006, from Australia, Austria, Belgium, France, Germany, Israel, The Netherlands, Japan, and the USA. Each paper was reviewed by at least three referees. The average number of reviews was four. All 15 submissions were accepted.

The submissions cover various languages, and deal with various aspects of termination: direct termination proofs (D), transformations (T), encoding of termination problems (E), termination proof certificates (C), growth functions (G), and non-termination (N). The following table gives an overview of the distribution of the submissions with respect to language (row) and aspect (column):

Language	D	T	E	C	G	N
String Rewriting	#7					
Term Rewriting	#5	#16	#4,#8,#10	#1	#14	#15
Typed λ-Calculus	#2	#6		#1		
Narrowing		#9				
Logic Programming		#12,#13				
Other		#3				#3

Every row and every column of this table is inhabited. This shows a healthy variety of interests among the participants. The clearest trends this year are SAT encoding for efficient termination proofs (3 submissions) and more transformations (6 submissions) than direct termination proof methods (3 submissions).

Invited talks have been given by Byron Cook and Yuri Gurevich, both from Microsoft Research. The 2006 workshop has continued the tradition from 2003 and 2004 by including a demonstration of termination tools and a report on the 2006 termination competition, organised by Claude Marché and Hans Zantema (`http://www.lri.fr/~marche/termination-competition/2006/`).

We wish to thank the FLoC organisers for their tremendous effort and congratulate them on a job well done. Moshe Vardi and Gopal Gupta provided clear guidelines for workshops and attended promptly to every request. Tom Ball managed the impressive FLoC web site. Ashish Tiwari, for RTA 2006, supported us all the way. We also thank the WST program committee for the excellent and conscientious job of refereeing all the papers. Paper submission, refereeing, author notification, and collation of final versions—all was managed by Andrei Voronkov's EasyChair system—a fantastic tool for anyone who has to organize a conference.

Alfons Geser and Harald Søndergaard
WST 2006 Program Chairs
August 2006

Contents and Schedule

Tuesday, 15 August, 2006

Wednesday, 16 August, 2006

Invited Talk **09:30 – 10:30**

Break **10:30 – 11:00**

Session 5 **11:00 – 12:30**

Lunch **12:30 – 14:00**

Session 6 **14:00 – 15:30**

Break **15:30 – 16:00**

Session 7 **16:00 – 16:30**

System Demonstrations **16:30 – 18:30**

WST 2006 Program Committee

Thomas Arts	IT Universitet Göteborg, SE
Alfons Geser	HTWK Leipzig, DE
Dieter Hofbauer	Universität Kassel, DE
Claude Marché	Université Paris-Sud, FR
Andreas Podelski	Max-Planck-Institut für Informatik, DE
Henny Sipma	Stanford University, US
Harald Søndergaard	University of Melbourne, AU
Andreas Weiermann	Universiteit Utrecht, NL

2006 Termination Competition Organisers

Claude Marché	Université Paris-Sud, FR
Hans Zantema	TU Eindhoven, NL

Program Termination Analyses for Free!

Byron Cook

Microsoft Research, Cambridge, UK
bycook@microsoft.com

I will describe an easy-to-follow recipe for the construction of automatic program termination analyses. The recipe's primary ingredient is an abstract-interpretation-based program analysis for invariance properties - any one will do. If we change the underlying program analysis, we get a different program termination analysis.

Termination Analysis for Logic Programs by Term Rewriting Revisited*
(extended abstract)

P. Schneider-Kamp[1], J. Giesl[1], A. Serebrenik[2], and R. Thiemann[1]

[1] LuFG Informatik 2, RWTH Aachen, Ahornstr. 55, 52074 Aachen, Germany, {psk,giesl,thiemann}@informatik.rwth-aachen.de
[2] Dept. of Mathematics and Computer Science, TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, a.serebrenik@tue.nl

1 Introduction

Techniques for automated termination analysis of logic programs can be classified into "direct" and "transformational" approaches. Direct approaches try to prove termination directly on the basis of the logic program, whereas transformational approaches transform logic programs into term rewrite systems (TRSs) and try to prove termination of the resulting TRS instead. We refer to [8] for an overview and to [6,9,13,18] for more recent work on "direct" approaches. "Transformational" approaches have been developed in [1,4,5,10,12,15,16,21] and a comparison of these approaches is given in [19]. Transformational methods

(I) should be *applicable* for a class of logic programs as large as possible and
(II) should produce TRSs whose termination is *easy to analyze automatically*.

Concerning (I), the above transformations can only be used for certain subclasses of logic programs. More precisely, all approaches except [15,16] are restricted to *well-moded* [2] logic programs. The transformations of [15,16] may be applied on a larger class of logic programs, but they generate very complex TRSs whose termination can hardly be proved automatically by existing termination provers, cf. [19].

Concerning (II), one needs an implementation and an empirical evaluation to find out whether termination of the transformed TRSs can indeed be verified automatically for a large class of examples. Unfortunately, to our knowledge there is only a single other termination tool available which implements a transformational approach. This tool TALP [20] is based on the transformations of [4,5,10] which are shown to be equally powerful in [19]. So these transformations are indeed suitable for automated termination analysis, but consequently, TALP only accepts well-moded logic programs.

We present a new transformation which, in contrast to all previous transformations, is applicable for *any* (definite) logic program, i.e. it satisfies (I).

* Supported by the Deutsche Forschungsgsmeinschaft DFG under grant GI 274/5-1.

We implemented our approach in our termination prover AProVE [11]. In the full version of this paper [22] we perform experiments on large collections of examples which show that our transformation indeed produces TRSs that are suitable for automated termination analysis and that using this approach AProVE is currently among the most powerful termination provers for logic programs. Thus, our approach also satisfies (II).

2 A New Transformation from Logic Programs to TRSs

Our transformation is inspired by the transformation of [4,5,10,19]. In this classical transformation, each argument position of each predicate is either labelled as *input* or *output*. As mentioned, the labelling must be such that the labelled program is *well-moded* [2]. Well-modedness guarantees that each atom is "sufficiently" instantiated during any derivation with a query that is ground on all input positions.

Example 1 *We consider a variant of an example from [19]. Let* p*'s first argument position be* input *and the second be* output *resulting in a well-moded program.*

$$\begin{array}{l}\mathsf{p}(X, X)\\ \mathsf{p}(\mathsf{f}(X), \mathsf{g}(Y)) \;:\!-\; \mathsf{p}(\mathsf{f}(X), \mathsf{f}(Z)), \mathsf{p}(Z, \mathsf{g}(Y))\end{array}$$

In the classical transformation from logic programs to TRSs [19], two new function symbols p_{in} and p_{out} are introduced for each predicate p. We write "$p(\boldsymbol{s}, \boldsymbol{t})$" to denote that $\boldsymbol{s}$ and $\boldsymbol{t}$ are the sequences of terms on p's in- and output positions, respectively.

- For each fact $p(\boldsymbol{s}, \boldsymbol{t})$, the TRS contains the rule $p_{in}(\boldsymbol{s}) \to p_{out}(\boldsymbol{t})$.
- For each clause c of the form $p(\boldsymbol{s}, \boldsymbol{t}) :\!- p_1(\boldsymbol{s}_1, \boldsymbol{t}_1), \ldots, p_k(\boldsymbol{s}_k, \boldsymbol{t}_k)$, the resulting TRS contains the following rules:

$$\begin{aligned} p_{in}(\boldsymbol{s}) &\to u_{c,1}(p_{1_{in}}(\boldsymbol{s}_1), \mathcal{V}(\boldsymbol{s}))\\ u_{c,1}(p_{1_{out}}(\boldsymbol{t}_1), \mathcal{V}(\boldsymbol{s})) &\to u_{c,2}(p_{2_{in}}(\boldsymbol{s}_2), \mathcal{V}(\boldsymbol{s}) \cup \mathcal{V}(\boldsymbol{t}_1))\\ &\ldots\\ u_{c,k}(p_{k_{out}}(\boldsymbol{t}_k), \mathcal{V}(\boldsymbol{s}) \cup \mathcal{V}(\boldsymbol{t}_1) \cup \ldots \cup \mathcal{V}(\boldsymbol{t}_{k-1})) &\to p_{out}(\boldsymbol{t}) \end{aligned}$$

  Here, $\mathcal{V}(\boldsymbol{s})$ are the variables in $\boldsymbol{s}$. Moreover, if $\mathcal{V}(\boldsymbol{s}) = \{x_1, ..., x_n\}$, then "$u_{c,1}(p_{1_{in}}(\boldsymbol{s}_1), \mathcal{V}(\boldsymbol{s}))$" denotes the term $u_{c,1}(p_{1_{in}}(\boldsymbol{s}_1), x_1, ..., x_n)$, etc.

If the resulting TRS is terminating, then the logic program terminates for any query with ground terms on all input positions of the predicates, cf. [19].

Example 2 *For Ex. 1, the transformation results in the following TRS* $\mathcal{R}$*.*

$$\begin{array}{ll} \mathsf{p}_{in}(X) \to \mathsf{p}_{out}(X) & \mathsf{u}_1(\mathsf{p}_{out}(\mathsf{f}(Z)), X) \to \mathsf{u}_2(\mathsf{p}_{in}(Z), X, Z)\\ \mathsf{p}_{in}(\mathsf{f}(X)) \to \mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X)), X) & \mathsf{u}_2(\mathsf{p}_{out}(\mathsf{g}(Y)), X, Z) \to \mathsf{p}_{out}(\mathsf{g}(Y)) \end{array}$$

The original logic program is terminating for any query $\mathsf{p}(t_1, t_2)$ *where* t_1 *is a ground term. However, the above TRS is not terminating:*

$$\mathsf{p}_{in}(\mathsf{f}(X)) \to_{\mathcal{R}} \mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X)), X) \to_{\mathcal{R}} \mathsf{u}_1(\mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X)), X), X) \to_{\mathcal{R}} \ldots$$

In the logic program, after resolving with the second clause, one obtains a query starting with $\mathsf{p}(\mathsf{f}(\ldots), \mathsf{f}(\ldots))$. *Since* p*'s output argument* $\mathsf{f}(\ldots)$ *is already partly instantiated, the second clause cannot be applied again for this atom. However, this information is neglected in the translated TRS. Here, one only regards the input argument of* p *in order to determine whether a rule can be applied. Note that current tools for termination proofs of logic programs like* cTI *[17],* Hasta-La-Vista *[23],* TALP *[20],* TermiLog *[14], and* TerminWeb *[7] fail on Ex. 1.*

So this example already illustrates a drawback of the classical transformation of [19]: there are several terminating well-moded logic programs which are transformed into non-terminating TRSs. In such cases, one fails in proving the termination of the logic program. Even worse, most of the existing transformations are not applicable for logic programs that are not well moded. A natural non-well-moded example is the append-program with the clauses $\mathsf{append}([\,], XS, XS)$ and $\mathsf{append}([X|XS], YS, [X|ZS]) \text{:-} \mathsf{append}(XS, YS, ZS)$. If one only considers append's first argument as input, then the program is not well moded but all queries $\mathsf{append}(t_1, t_2, t_3)$ are terminating if t_1 is ground.

Recently, several authors tackled the problem of applying termination techniques from term rewriting for (possibly non-well-moded) logic programs. A framework for integrating orders from term rewriting into direct termination approaches for logic programs is discussed in [9].[1] However, the automation of this framework is non-trivial in general.

Therefore, unlike [9], we choose a transformational approach. To obtain goal (I) we modify the classical transformation of logic programs into TRSs to make it applicable for *arbitrary* (possibly non-well-moded) programs as well. Instead of partitioning the positions of p/n into input and output positions, now we keep *all* arguments both for p_{in} and p_{out}.

- For each fact $p(\boldsymbol{s})$ in $\mathcal{P}$, the TRS contains the rule $p_{in}(\boldsymbol{s}) \to p_{out}(\boldsymbol{s})$.
- For each clause c of the form $p(\boldsymbol{s}) \text{:-} p_1(\boldsymbol{s}_1), \ldots, p_k(\boldsymbol{s}_k)$ in $\mathcal{P}$, the resulting TRS contains:

$$\begin{aligned} p_{in}(\boldsymbol{s}) &\to u_{c,1}(p_{1_{in}}(\boldsymbol{s}_1), \mathcal{V}(\boldsymbol{s})) \\ u_{c,1}(p_{1_{out}}(\boldsymbol{s}_1), \mathcal{V}(\boldsymbol{s})) &\to u_{c,2}(p_{2_{in}}(\boldsymbol{s}_2), \mathcal{V}(\boldsymbol{s}) \cup \mathcal{V}(\boldsymbol{s}_1)) \\ &\ldots \\ u_{c,k}(p_{k_{out}}(\boldsymbol{s}_k), \mathcal{V}(\boldsymbol{s}) \cup \mathcal{V}(\boldsymbol{s}_1) \cup \ldots \cup \mathcal{V}(\boldsymbol{s}_{k-1})) &\to p_{out}(\boldsymbol{s}) \end{aligned}$$

[1] But in contrast to [9], we also apply more recent powerful termination techniques from rewriting (e.g., *dependency pairs* [3]) for termination of logic programs.

With the new transformation we have achieved goal (I): it can handle programs like Ex. 1 where classical transformations like [19] fail; moreover it can also transform non-well-moded programs where most current transformational techniques are not even applicable.

Example 3 *For the logic program of Ex. 1, we obtain the following TRS.*

$$\begin{aligned}
\mathsf{p}_{in}(X,X) &\to \mathsf{p}_{out}(X,X)\\
\mathsf{p}_{in}(\mathsf{f}(X),\mathsf{g}(Y)) &\to \mathsf{u}_1(\mathsf{p}_{in}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y)\\
\mathsf{u}_1(\mathsf{p}_{out}(\mathsf{f}(X),\mathsf{f}(Z)),X,Y) &\to \mathsf{u}_2(\mathsf{p}_{in}(Z,\mathsf{g}(Y)),X,Y,Z)\\
\mathsf{u}_2(\mathsf{p}_{out}(Z,\mathsf{g}(Y)),X,Y,Z) &\to \mathsf{p}_{out}(\mathsf{f}(X),\mathsf{g}(Y))
\end{aligned}$$

Our new transformation results in TRSs where the notion of "rewriting" has to be slightly modified: we regard a restricted form of infinitary rewriting, called *infinitary constructor rewriting*. The reason is that logic programs use *unification*, whereas TRSs use *matching*. Therefore the logic program $\mathsf{p}(\mathsf{s}(X)) \text{ :- } \mathsf{p}(X)$ does not terminate for the query $\mathsf{p}(X)$ whereas the TRS $\mathsf{p}(\mathsf{s}(X)) \to \mathsf{p}(X)$ terminates for all finite terms. However, the infinite derivation of the logic program corresponds to an infinite reduction of the TRS with the *infinite* term $\mathsf{p}(\mathsf{s}(\mathsf{s}(\ldots)))$ containing infinitely many s-symbols. So to simulate unification by matching, we have to regard TRSs where the variables in rewrite rules may be instantiated by infinite terms. It turns out that this form of rewriting also analyzes the termination behavior of logic programs with infinite terms, i.e., of logic programming without occur check.

Moreover, in the second rule of Ex. 3 we have a variable Z in the right-hand side which does not occur in the left-hand side. Such rules are never terminating in standard rewriting as one can instantiate variables like Z with arbitrary terms. Therefore, we modify the rewrite relation to *constructor* rewriting where we may only instantiate variables by constructor terms. Constructor rewriting correctly simulates the behavior of logic programs, since variables in a logic program are only instantiated by "constructor terms".

So in contrast to the old transformation in Ex. 2, now all terms $\mathsf{p}_{in}(t_1,t_2)$ terminate for the TRS if t_1 is finite, since now the second argument of p_{in} prevents an infinite application of the second rule.

In the full version of the paper [22] we showed that existing termination techniques for TRSs (in particular, the dependency pair framework) can easily be adapted in order to prove termination of infinitary constructor rewriting. Moreover, the experiments in [22] have shown that goal (II) is achieved as well. In other words, the implementation of our approach can indeed compete with modern tools for direct termination analysis of logic programs and it succeeds for many programs where these tools fail.

Acknowledgements. We thank M. Codish, D. De Schreye, and F. Mesnard for helpful comments and R. Bagnara and S. Genaim for help with the experiments.

References

1. G. Aguzzi and U. Modigliani. Proving termination of logic programs by transforming them into equivalent term rewriting systems. In *Proc. 13th FST & TCS*, LNCS 761, pages 114–124, 1993.
2. K. R. Apt and S. Etalle. On the unification free Prolog programs. In *Proc. 18th MFCS*, LNCS 711, pages 1–19, 1993.
3. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
4. T. Arts and H. Zantema. Termination of logic programs using semantic unification. In *Proc. 5th LOPSTR*, LNCS 1048, pages 219–233, 1995.
5. M. Chtourou and M. Rusinowitch. Méthode transformationelle pour la preuve de terminaison des programmes logiques. Unpublished manuscript, 1993.
6. M. Codish, V. Lagoon, and P. Stuckey. Testing for termination with monotonicity constraints. In *Proc. 21st ICLP*, LNCS 3668, pages 326–340, 2005.
7. M. Codish and C. Taboch. A semantic basis for termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
8. D. De Schreye and S. Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, 19/20:199–260, 1994.
9. D. De Schreye and A. Serebrenik. Acceptability with general orderings. In *Comp. Logic. Logic Prog. and Beyond.*, LNCS 2407, pages 187–210, 2002.
10. H. Ganzinger and U. Waldmann. Termination proofs of well-moded logic programs via conditional rewrite systems. In *Proc. 3rd CTRS*, LNCS 656, 1993.
11. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the DP framework. In *Proc. 3rd IJCAR*, 2006. To appear.
12. M. Krishna Rao, D. Kapur, and R. Shyamasundar. Transformational methodology for proving termination of logic programs. *J. Log. Pr.*, 34(1):1–42, 1998.
13. V. Lagoon, F. Mesnard, and P. J. Stuckey. Termination analysis with types is more accurate. In *Proc. 19th ICLP*, LNCS 2916, pages 254–268, 2003.
14. N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. TermiLog: A system for checking termination of queries to logic programs. In *Proc. 9th CAV*, LNCS 1254, 1997.
15. M. Marchiori. Logic programs as term rewriting systems. In *Proc. 4th ALP*, LNCS 850, pages 223–241, 1994.
16. M. Marchiori. Proving existential termination of normal logic programs. In *Proc. 5th AMAST*, LNCS 1101, pages 375–390, 1996.
17. F. Mesnard and R. Bagnara. cTI: A constraint-based termination inference tool for ISO-Prolog. *Theory and Practice of Logic Prog.*, 5(1&2):243–257, 2005.
18. F. Mesnard and S. Ruggieri. On proving left termination of constraint logic programs. *ACM Transaction on Computational Logic*, 4(2):207–259, 2003.
19. E. Ohlebusch. Termination of logic programs: Transformational methods revisited. *Appl. Algebra in Engineering, Comm. and Computing*, 12:73–116, 2001.
20. E. Ohlebusch, C. Claves, and C. Marché. TALP: A tool for the termination analysis of logic programs. *Proc. 11th RTA*, LNCS 1833, pages 270–273, 2000.
21. F. van Raamsdonk. Translating logic programs into conditional rewriting systems. In *Proc. 14th ICLP*, pages 168–182. MIT Press, 1997.
22. P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated Termination Analysis for Logic Programs by Term Rewriting. In *Proc. 16th LOPSTR*, LNCS, 2006. To appear. Available at `http://aprove.informatik.rwth-aachen.de/eval/LP`.
23. A. Serebrenik and D. De Schreye. Inference of termination conditions for numerical loops in Prolog. *Theory and Practice of Logic Prog.*, 4:719–751, 2004.

Program Specialisation as a Preprocessing Step for Termination Analysis

Manh Thang Nguyen[1], Maurice Bruynooghe[1], Danny De Schreye[1], and Michael Leuschel[2]

[1] Department of Computer Science, K.U.Leuven, Belgium
{ManhThang.Nguyen, Maurice.Bruynooghe,Danny.DeSchreye}@cs.kuleuven.be
[2] University of Düesseldorf, Germany
leuschel@cs.uni-duesseldorf.de

1 Introduction

Previous works have shown that structural information can be used to transform a logic program into another one with the same termination behaviour but where the termination analysis becomes easier and more precise (i.e. it can prove termination for a larger class of queries) [1,2]. In [2], a transformation which is based on partial evaluation and predicate renaming was proposed. However, this work was done specifically for Terminweb[1] with its focus on binary clauses. Another limitation of the work is that only few experiments were performed.
In this paper, we present a transformation technique which consists of two simple steps:

- **Predicate Renaming.** Based on the different structural information of predicates collected in a given program, a predicate is split into different predicates and the program is transformed to a new one. This transformation process is termination preserving.
- **Program unfolding.** Partial evaluation is applied to unfold the program in a way such that termination is preserved. It is done by using ECCE, a well-known off-the-shelf partial evaluator.

The advantage of this technique is that it applies directly to logic programs without binarising them. Therefore, it can be considered as a pre-processing step for termination analysis in general and can be integrated into any termination analyser. First, we apply predicate renaming on the original program to generate a new program, and if the termination prover fails, then we apply program unfolding and do termination analysis again on the unfolded program. Note that these two steps are independent and can be applied individually and in any order. We have done an extensive experiment on a large number of benchmarks and the results show that the transformation makes the termination analysis considerably more precise and the running time for the analysis does not increase much.

[1] http://lvs.cs.bgu.ac.il/~mcodish/suexec/terminweb/bin/terminweb.cgi

2 Predicate Renaming

This specialisation method is an alternative to the approach in [2]. The difference is that this transformation is applicable not only for binary logic programs but also for any pure logic program. Moreover, it generates new logic programs with less clauses than the ones in [2]. The basic idea of this method is to rename predicates so that a predicate can be separated into different new predicates. We illustrate this approach by considering the following example which calculates the Fibonacci numbers according to its index.

Example 1 (Fibonacci).

$$p(add(X,0),X). \tag{1}$$
$$p(add(X,s(Y)),s(Z)) : -p(add(X,Y),Z). \tag{2}$$
$$p(f(0),s(0)). \tag{3}$$
$$p(f(s(0)),s(0)). \tag{4}$$
$$p(f(s(s(X))),Z) : -p(f(s(X)),U),p(f(X),V),p(add(U,V),Z). \tag{5}$$

Obviously, this program terminates w.r.t. the query set $Q = \{p(t_1, t_2)|$ where t_1 is a ground term and t_2 is a free variable$\}$, because both clauses (2) and (5) can be applied only a finite number of times. For clause (5), only the first and second atoms in the body of this clause are unifiable with its head and the first argument of each predicate call is decreased. For the third atom $p(add(U, V), Z)$, any call to this atom is followed by a finite number of applications of clause (2) (the decrease of the first argument). However, to the best of our knowledge, proving termination of this example is impossible for most current termination analysers, except TALP[2], a tool which proves termination of a logic program by first transforming it to a corresponding term rewriting system. The reason for failing on the example is that any linear or polynomial level mapping which guarantees termination for clause (2) also leads to the fact that the mapping of the third atom in clause (5) is greater than the mapping of its head. Since the third atom in the body of clause (5) and the atom in the body of clause (2) can only be unified with the heads of clauses (1) and (2), the first atom in the body of clause (5) can only be unified with the heads of clauses (4) and (5), the second atom of this clause can only be unified with the heads of clauses (3), (4) and (5), the idea is to split the predicate $p/2$ into different predicates corresponding to different cases. Formally, our predicate renaming consists of two steps:

1. For each predicate call to p/n that can unify with only the heads of a proper subset of clauses S defining the predicate p/n: replace that call by a new predicate p_S/n. This transforms the original program P to P'.
2. Extend P' with the definitions of each new predicate p_S/n: copy the clauses from S in P' and replace the head predicate p/n by p_S/n. If S is empty, a new clause $p_S(t_1, .., t_n) : -fail$ is generated and added to P'.

[2] http://bibiserv.techfak.uni-bielefeld.de/talp/

For example 1, the first step transforms the original program to a new one:

$p(add(X,0),X).$ $\quad$ $p(add(X,s(Y)),s(Z)) : -p_{1:2}(add(X,Y),Z).$
$p(f(0),s(0)).$ $\quad$ $p(f(s(0)),s(0)).$
$p(f(s(s(X))),Z) : -p_{4:5}(f(s(X)),U), p_{3:4:5}(f(X),V), p_{1:2}(add(U,V),Z).$

Applying the second step, the following new clauses which define the new predicates $p_{1:2}/2$, $p_{4:5}/2$ and $p_{3:4:5}/2$ are added:

$p_{1:2}(add(X,0),X).$ $\quad$ $p_{1:2}(add(X,s(Y)),s(Z)) : -p_{1:2}(add(X,Y),Z).$
$p_{4:5}(f(s(0)),s(0)).$
$p_{4:5}(f(s(s(X))),Z) : -p_{4:5}(f(s(X)),U), p_{3:4:5}(f(X),V), p_{1:2}(add(U,V),Z).$
$p_{3:4:5}(f(0),s(0)).$ $\quad$ $p_{3:4:5}(f(s(0)),s(0)).$
$p_{3:4:5}(f(s(s(X))),Z) : -p_{4:5}(f(s(X)),U), p_{3:4:5}(f(X),V), p_{1:2}(add(U,V),Z).$

It is easy to see that the transformation preserves the structure of the SLD-tree of the original program, and is termination preserving. If there exists a termination proof (a norm and level mapping) for the original program, there also exists at least a norm and a level mapping from which a termination proof for the transformed program is derived (the norm is kept the same as in the original program, the new level mappings for all renamed predicates are the same as that of the original predicate). Moreover, this transformation provides more space for searching for a suitable norm and level mapping since renamed predicates may have different level mappings (e.g. $p_{1:2}, p_{4:5}, p_{3:4:5}$ may have different level mappings).
We have tested the example before and after applying predicate renaming with Polytool[3], a tool for automated termination proof which is based on polynomial interpretations but in its most simple form: only linear interpretations are used. The result shows that the termination of the transformed program can be proved by Polytool (i.e. the system provides the following norms: $\|add(t_1,t_2)\| = \|t_2\|, \|s(t)\| = \|t\| + 1, \|f(t)\| = \|t\|$ and level mappings: $|p_{1:2}(t_1,t_2)| = \|t_1\|, |p_{4:5}(t_1,t_2)| = \|t_1\|, |p_{3:4:5}(t_1,t_2)| = \|t_1\|$), while its original cannot.

3 Program Unfolding

When applying program unfolding, we need to take into account the fact that the unfolding does not preserve non-termination in general. It is because partial evaluation may *"remove"* an infinite branch out of the SLD-tree of the program. Fortunately, the ECCE partial evaluator with the option **"termdet"** (only allowing one non-determinate unfolding step) is termination preserving and hence is suitable for termination analysis. Another option **"term"** is also termination preserving. However, this uses a quite aggressive unfolding rule (allowing leftmost non-determinate unfolding with homeomorphic embedding); this may lead to much larger specialised programs which

[3] http://www.cs.kuleuven.be/~manh/polytool/

require much more time for termination analysis. In this paper, we use ECCE with the option **"termdet"**.
Let us apply program unfolding on the following example *remainder* with the query set $Q = \{rem(t_1, t_2, t_3) | t_1$ and t_2 are ground and t_3 is a free variable$\}$. For this example, predicate renaming does not change the original program and Polytool fails to prove its termination.

Example 2 (remainder).

$$rem(X, Y, R) : -notZero(Y), sub(X, Y, Z), rem(Z, Y, R). \quad (6)$$
$$rem(X, Y, X) : -notZero(Y), geq(Y, X). \quad (7)$$
$$sub(s(X), s(Y), Z) : -sub(X, Y, Z). \quad (8)$$
$$sub(X, 0, X). \quad (9)$$
$$notZero(s(X)). \quad (10)$$
$$geq(s(X), s(Y)) : -geq(X, Y). \quad (11)$$
$$geq(X, 0).$$

If we look at clause (6), the decrease between the size of the head $rem(X, Y, R)$ and the recursive body atom $rem(Z, Y, R)$ can be established if the size of X is bigger than the size of Z. This may be done by means of a valid interargument relation of the predicate $sub(X, Y, Z)$ of the form: $||X|| > ||Z||$. However, Polytool is unable to capture such interargument relation because of clause (9) and therefore, cannot prove termination of this example.
After unfolding with clause (10) and then with clause (8), clauses (6) and (7) are replaced by the two following clauses while the other clauses remain the same:

$$rem(s(X), s(Y), R) : -sub(X, Y, Z), rem(Z, s(Y), R).$$
$$rem(X, s(Y), X) : -geq(s(Y), X).$$

Now the requirement for the interargument relation is relaxed. A valid interargument relation for the predicate $sub(X, Y, Z)$ such as $||X|| \geq ||Z||$ is enough for proving termination of the transformed program. Indeed, Polytool is able to prove termination of this example.

4 Experiments

We have fully implemented the technique and tested it with a large number of benchmarks[4] and three termination analysers: TerminWeb, CTi[5] and Polytool. Experiments on TerminWeb and CTi are done through their web interfaces with their default settings. For Polytool, only linear interpretations are used. The results[6] are summarised in Table 1. The table shows that both predicate renaming and program unfolding considerably improve termination analysis in terms of precision and are not very costly. The combination of both techniques provides us the best precision.

[4] The benchmarks are at http://www.cs.kuleuven.be/~manh/polytool/WST06-bmk.zip
[5] http://www.complang.tuwien.ac.at/cti/
[6] Detail results are at http://www.cs.kuleuven.be/~manh/polytool/WST06-result.zip

	Polytool	TerminWeb	CTi
Origin Programs	61	54	55
After Predicate Renaming	78	60	60
After Program Unfolding(ECCE)	76	66	68
After Program Renaming + Unfolding	81	69	72

Table 1. Number of successful proofs over 84 benchmarks

5 Conclusion and Discussion

We have introduced a simple specialisation technique through predicate renaming and program unfolding. We have also discussed how this technique may improve termination analysis. Note that although these issues are not new (as discussed in [2]), with a more general approach and extensive experiments, we have shown that this technique can significantly improve precision of the analysis and can be applied as a pre-processing step in any termination analyser. In addition, a number of difficult examples become easy after transformation and hence one could argue that they are not difficult at all and are a poor motivation for more complex termination analysis techniques. Note that a disadvantage of program unfolding by ECCE is that it tries to unfold every clause in the original program and the transformed program may be larger such that the analysis on it is much more costly. A solution for this problem is to apply partial evaluation only for clauses which are involved in the failure of termination analyser. In [2], a suggestion is to apply unfolding on a clause only if the sizes of the atoms in its body get smaller than the size of its head according to a selected norm. However, in Polytool, we don't select a particular norm and level mapping in advance. We search for a suitable norm and level mapping instead. A simple solution for this issue is to build proofs incrementally, based on the dependency graph for the program. One first proves termination for the bottom layer in the predicate dependency graph. Then one repeatedly works upwards for next mutually dependent predicates. Whenever the analyser fails to prove termination, it provides information on the causes of the failure. Only for the clauses that were involved in the problem, partial deduction would be applied. In future work, we will explore this further as well as the usefulness of the technique in the term rewriting context.

References

1. A. Serebrenik and D. De Schreye. Proving termination with adornments. In *LOPSTR'03: Preproceedings of International Symposium on Logic-based Program Synthesis and Transformation*, pages 113–148, 2003.
2. L. Tamary and M. Codish. Abstract partial evaluation for termination analysis. In M. Codish and A. Middeldorp, editors, *WST'04: 7th International Workshop on Termination*, pages 47–50, Aachen, Germany, June 2004.

Dependency Graph Method for Proving Termination of Narrowing *

Naoki Nishida and Koichi Miura

Graduate School of Information Science, Nagoya University, Nagoya, Japan
nishida@is.nagoya-u.ac.jp miura@trs.cm.is.nagoya-u.ac.jp

Abstract. Term rewriting systems with extra variables are useful in encoding operators for inverse computation. Their ground rewrite sequences can be simulated by narrowing sequences. In this paper, we refine the dependency pair method for proving termination of narrowing and extend the dependency graph method for proving termination of rewriting to a method for narrowing.

1 Introduction

Term rewriting systems with extra variables (*EV-TRSs*, for short), i.e., sets of rewrite rules that may have extra variables, are useful in encoding operators that compute inverse images of the corresponding functions [12,13]. Here, *extra variables* of a rewrite rule are variables appearing not in the left-hand side but in the right-hand side. For example, the following EV-TRS is a part of the system automatically generated by inversion compilers in [12,13] from a program for computing multiplication of natural numbers:

$$R_1 = \left\{ \begin{array}{ll} \overline{\mathsf{mul}}(\mathsf{0}) \to \mathsf{tp}_2(\mathsf{0}, y), & \overline{\mathsf{mul}}(\mathsf{0}) \to \mathsf{tp}_2(x, \mathsf{0}), \\ \overline{\mathsf{mul}}(\mathsf{s}(z)) \to \mathsf{u}_2(\overline{\mathsf{add}}(z)), & \mathsf{u}_2(\mathsf{tp}_2(w, y)) \to \mathsf{u}_3(\overline{\mathsf{mul}}(w), y), \\ \mathsf{u}_3(\mathsf{tp}_2(x, \mathsf{s}(y)), y) \to \mathsf{tp}_2(\mathsf{s}(x), \mathsf{s}(y)), & \cdots \end{array} \right\}.$$

Here $\overline{\mathsf{mul}}$ and $\overline{\mathsf{add}}$ are inverse operators of multiplication and addition of natural numbers, respectively, and tp_2 is a constructor representing pairs of two terms.

The rewrite relations by rewrite rules having at least an extra variable are infinitely branching and cause non-termination. However, it was shown that *narrowing* reduction can simulate ground rewrite sequences of EV-TRSs if either the sequences are *EV-safe* or the systems are right-linear [10,11]. Here, a rewrite sequence on an EV-TRS is called *EV-safe* if any redex introduced by means of extra variables is not reduced along the sequence. Typical instances of EV-safe sequences are rewrite sequences where extra variables are substituted with normal forms at every rewrite step. Moreover, the EV-safety is not restrictive for inverse computation. Proving termination of narrowing sequences starting from ground terms is required because termination of inverse computation by EV-TRSs generated in [12,13] coincides with that of the narrowing sequences. Termination of such narrowing seems to be very similar to that of rewriting, while that of narrowing starting from arbitrary terms is not. For example, narrowing of R_1 starting from meaningful terms $\overline{\mathsf{mul}}(\mathsf{s}^n(\mathsf{0}))$ $(n \geq 0)$ is terminating, while it is not terminating if starting from arbitrary ground terms. The only way to prove the termination is mathematical induction on n by hand.

To prove termination of inverse computation, the *dependency pair method* [1] for proving termination of rewriting has already been extended to narrowing, by adding a condition. The extended method is applicable for constructor or right-linear systems [11]. However, the method is slightly too weak to prove termination of $\overline{\mathsf{mul}}(\mathsf{s}^n(\mathsf{0}))$.

* This work is partly supported by MEXT. KAKENHI #17700009.

In this paper, we refine the dependency pair method extended in [11], that is, we remove the additional condition on the original method. This refinement fills a gap between the methods for rewriting and narrowing starting from ground terms. We also extend the *dependency graph method* [1] for proving termination of rewriting, which is stronger than the dependency pair method, for proving termination of the narrowing. These methods are also applicable to either constructor or right-linear systems. One of the differences between the dependency graph methods for rewriting and narrowing is that an additional condition is imposed on argument filterings used in the method to maintain the groundness of sequences obtained by applying the argument filterings to dependency chains of narrowing.

A similar method for proving termination of narrowing was proposed by J. Christian [3]. This method corresponds to the well-known termination proof method that R is terminating if and only if there exists a reduction order $>$ such that $R \subseteq >$ [8], where the left-hand sides of rules are *flat*. At every step of narrowing of the TRSs, the number of kinds of variables does not increase. Therefore, the method for rewriting works well for narrowing. However, it seems to be difficult to apply this method to the termination proof of EV-TRSs, since the existence of extra variables impedes the main feature of this method.

This paper follows the basic notions of term rewriting [2,14], and the definition of *argument filterings* in [6]. *Extra variables* of a rewrite rule $l \to r$ are variables not in l but in r. A *TRS with extra variables* (*EV-TRS*, for short) is a set of rewrite rules that may have extra variables. The set of all variables in a term t is denoted by $\mathcal{V}ar(t)$. *Narrowing* of EV-TRSs is defined similarly to that of TRSs [5]. Let R be an EV-TRS. A term s is said to be *narrowable* into a term t at a non-variable occurrence p, written as $s \ {}_{\sigma|_{\mathcal{V}ar(s)}}\!\!\rightsquigarrow^{p}_{R} \ t$, if there exist a variant $\rho' : l \to r$ of a rewrite rule ρ in R and a substitution σ such that $t \equiv (s[r]_p)\sigma$ and σ is a most general unifier of $s|_p$ and l. The relation $\rightsquigarrow_R$ is called *narrowing* of R. A term that is reachable from a ground term is said to be *with a ground antecedent*. We simply say that narrowing on terms with ground antecedents is *narrowing with ground antecedents* (*GA-narrowing*, for short), that is, the binary relation $\overset{\rightsquigarrow}{\mathrm{GA}}_R = \{(s,t) \mid s \rightsquigarrow_R t, (\exists s_0 \in \mathcal{T}(\mathcal{F}), s_0 \overset{*}{\rightsquigarrow}_R s)\}$. For example, the set $R_2 = \{\, \mathsf{f}(x,\mathsf{0}) \to \mathsf{s}(x), \mathsf{g}(x) \to \mathsf{h}(x,y), \mathsf{h}(\mathsf{0},x) \to \mathsf{f}(x,x) \,\}$ is an EV-TRS. The ground term $\mathsf{g}(\mathsf{0})$ is narrowable by three steps to $\mathsf{s}(\mathsf{0})$, that is, $\mathsf{g}(\mathsf{0}) \rightsquigarrow_{R_2} \mathsf{h}(\mathsf{0},y) \ {}_{\{y\mapsto\mathsf{0}\}}\!\!\rightsquigarrow_{R_2} \mathsf{f}(\mathsf{0},\mathsf{0}) \rightsquigarrow_{R_2} \mathsf{s}(\mathsf{0})$.

2 Dependency Pair Method for Narrowing

In this section, we refine the dependency pair method for narrowing [11].

Unlike the case of rewriting, the approach by dependency pairs is not applicable to all EV-TRSs [11]. It is applicable to the systems whose narrowing has the *TRAT property*, which is necessary to guarantee the existence of 'minimal' dependency chains for infinite sequences. Let R be an EV-TRS and $\to$ be a subset of either $\rightarrow_R$ or $\rightsquigarrow_R$. An infinite $\to$-sequence is said to be *almost terminating with respect to* $\to$ if every proper subterm of its initial term is terminating with respect to $\to$. An almost terminating $\to$-sequence is said to be *top-reduced* if it contains a reduction step at the root (top) position. An infinite $\to$-sequence *has a TRAT sequence* if there exists a subterm of its initial term, which causes a top-reduced almost terminating $\to$-sequence. The relation $\to$ has the *TRAT property* (*top-reduced almost terminating property*) if every infinite $\to$-sequence has a TRAT sequence. Unfortunately, narrowing does not generally have the TRAT property.

Example 1. Consider the TRS $R_3 = \{\, \mathsf{f}(\mathsf{f}(x)) \to x \,\}$ over a signature with a binary symbol c. We have the infinite $\rightsquigarrow_{R_3}$-sequence $\mathsf{c}(\mathsf{f}(x),x) \ {}_{\{x\mapsto\mathsf{f}(x')\}}\!\!\rightsquigarrow_{R_3} \mathsf{c}(x',\mathsf{f}(x')) \ {}_{\{x'\mapsto\mathsf{f}(x'')\}}\!\!\rightsquigarrow_{R_3}$

$\mathsf{c}(\mathsf{f}(x''), x'') \leadsto_{R_3} \cdots$. Since the above infinite almost-terminating sequence is not top-reduced, narrowing of R_3 does not have the TRAT property.

However, narrowing of EV-TRSs in some classes has the TRAT property.

Proposition 2 ([11]). *Let R be an EV-TRS.*

- *If R is right-linear, then $\leadsto_R$ has the TRAT property on linear terms.*
- *If R is a constructor system, then $\leadsto_R$ has the TRAT property.*

Dependency pairs of EV-TRSs are defined similarly to those of TRSs [1] (see [11]). The set of all dependency pairs of R is denoted by $\mathcal{DP}(R)$. *R-chains* are sequences of dependency pairs that are in turn connected with the rewrite relation of R. Chains for narrowing are defined similarly by connecting dependency pairs with the narrowing of R.

Definition 3 ([10,11]). Let R be an EV-TRS. A sequence $\langle s_1, t_1\rangle \langle s_2, t_2\rangle \cdots$ of dependency pairs of R is called an *R-narrowing-chain* if there exist terms $u_1, u_2, \cdots$ and substitutions $\sigma_1, \sigma_2, \cdots$ such that $u_1 \leadsto^{\varepsilon}_{\{s_1 \to t_1\}} t_1\sigma_1 \overset{*}{\leadsto}{}^{\varepsilon<}_{R} u_2 \leadsto^{\varepsilon}_{\{s_2 \to t_2\}} t_2\sigma_2 \overset{*}{\leadsto}{}^{\varepsilon<}_{R} \cdots$ where $\mathcal{V}ar(u_i) \cap \mathcal{V}ar(s_i, t_i) = \emptyset$ and σ_i is a most general unifier of s_i and u_i. In particular, the sequence is said to be *with a ground antecedent* if there exists a ground term s_0 such that $s_0 \overset{*}{\leadsto}{}^{\varepsilon<}_{R} u_1$.

To refine the method in [11], we prepare the following lemma.

Lemma 4. *Let R be an EV-TRS, $(\succsim, \succ)$ a reduction pair, π an argument filtering, s and t terms, and δ a substitution. If $s\ {}_{\delta}\overset{*}{\leadsto}_R\ t$ and $\pi(R) \subseteq \succsim$ (respectively $s\ {}_{\delta}\leadsto^{\varepsilon}_{\{l \to r\}}\ t$ and $l \succ r \subseteq \succsim$), then $\pi(s\delta) \succsim \pi(t)$ (respectively $\pi(s\delta) \succ \pi(t)$).*

The dependency pair method [1] for proving termination of rewriting was extended for GA-narrowing and narrowing in [11], and the extended method is refined as follows.

Theorem 5. *Let R be an EV-TRS, and suppose that $\leadsto_R$ has the TRAT property. R is terminating with respect to $\overset{\rightsquigarrow}{\text{GA}}_R$ if there exist a reduction pair $(\succsim, \succ)$ and an argument filtering π such that*

(a) $\pi(R) \subseteq \succsim$ and $\pi(\mathcal{DP}(R)) \subseteq \succ$.

R is terminating with respect to $\leadsto_R$ if $(\succsim, \succ)$ and π satisfy (a) and

(b) for all pairs $\langle s, t\rangle \in \mathcal{DP}(R)$, $\pi(t)$ is ground.

Proof (Sketch). By using Lemma 4 instead of Lemma 4.14 [11] in the proof of Theorem 4.16 [11], we can easily construct from an infinite R-narrowing-chain an infinite sequence on $\succ$. It follows from Theorem 4.9 in [11] that $\overset{\rightsquigarrow}{\text{GA}}_R$ is not terminating. □

Condition (a) is the same as that in the case of rewriting. The refined point from the method in [11] is the removal of the restriction against argument filterings and the condition "$\pi(R) \cup \pi(\mathcal{DP}(R)) \subseteq \supseteq_{\mathcal{V}ar}$" where $\supseteq_{\mathcal{V}ar} = \{(s, t) \mid \mathcal{V}ar(s) \supseteq \mathcal{V}ar(t)\}$. Condition (b) guarantees that every infinite narrowing-chain not starting from any non-ground term has a postfix sequence that is a narrowing-chain with a ground antecedent. This guarantee reduces termination of narrowing to that of GA-narrowing.

Example 6. Consider the EV-TRS R_2 again. The dependency pairs of R_2 are $\langle \mathsf{g}^\sharp(x), \mathsf{h}^\sharp(x, y)\rangle$ and $\langle \mathsf{h}^\sharp(\mathsf{0}, x), \mathsf{f}^\sharp(x, x)\rangle$. Let π_2 be an argument filtering such that $\pi_2(\mathsf{f}) = \pi_2(\mathsf{g}) = \pi_2(\mathsf{h}) = \pi_2(\mathsf{f}^\sharp) = \pi_2(\mathsf{h}^\sharp) = [\,]$ and $\pi_2(\mathsf{g}^\sharp) = [1]$, and $>$ be a precedence over $\{\mathsf{f}, \mathsf{g}, \mathsf{h}, \mathsf{s}, \mathsf{0}, \mathsf{f}^\sharp, \mathsf{g}^\sharp, \mathsf{h}^\sharp\}$ such that $\mathsf{g}^\sharp > \mathsf{h}^\sharp > \mathsf{f}^\sharp \leq \mathsf{f} = \mathsf{g} = \mathsf{h} = \mathsf{s} = \mathsf{0}$. Then we have $\pi_2(R_2) \subseteq \geq_{\mathsf{lpo}}$ and $\pi_2(\mathcal{DP}(R_2)) \subseteq >_{\mathsf{lpo}}$, and hence $\overset{\rightsquigarrow}{\text{GA}}_{R_2}$ is terminating. Moreover, $\leadsto_{R_2}$ is terminating because $\pi_2(\mathsf{h}^\sharp(\mathsf{0}, x))$ and $\pi_2(\mathsf{f}^\sharp(x, x))$ are ground.

The sufficient condition for termination of GA-narrowing in Theorem 5 corresponds to that for termination of rewriting. From this fact, the implementation of an automatic termination proof by Theorem 5 can be easily realized by relaxing the restriction "being TRSs" on inputs of existing tools, which are based on the dependency pair method for rewriting, to "being EV-TRSs". As we describe later, such implementation will be faulty if the existing tools are related to the dependency graph method.

3 Dependency Graph Method for Narrowing

As described in the previous section, we succeeded in extending the dependency pair method for rewriting to a method for GA-narrowing, without differences. This extension makes it also possible to extend the *dependency graph method* for rewriting [1] to a method for GA-narrowing. Unfortunately, unlike the case of the dependency pair method, we must add a condition to the original method. The dependency pair method cannot prove termination of the EV-TRS $R_4 = \{\mathsf{f}(x,x) \to y\}$, but the dependency graph method enables us to prove it.

We first define the dependency graph for narrowing.

Definition 7. Let R be an EV-TRS. The *narrowing dependency graph* of R, denoted by $\mathcal{NDG}(R)$, is the directed graph whose nodes are the dependency pairs of R, and there is an arc from $\langle s,t\rangle$ to $\langle u,v\rangle$ if $\langle s,t\rangle\langle u,v\rangle$ is an R-narrowing-chain.

The notion of *strongly connected subgraphs* (*SCSs*, for short) of $\mathcal{NDG}(R)$ is the same as the notion of 'cycle' in [14].

Theorem 8. *Let R be an EV-TRS. R is terminating with respect to $\overset{\rightsquigarrow}{\mathrm{GA}}_R$ if for every SCS $\mathcal{P}$ in $\mathcal{NDG}(R)$ there exist a reduction pair $(\succsim, \succ)$ and an argument filtering π such that*

(a) $\pi(R) \subseteq \succsim$, $\pi(\mathcal{P}) \subseteq \succsim$ and $\pi(\mathcal{P}) \cap \succ \neq \emptyset$, and
(b) either $\pi(P') \subseteq \succsim$ or $\pi(R) \cup \pi(P') \subseteq \sqsupseteq_{\mathcal{V}ar}$, where P' is a set of dependency pairs, each of which is reachable to a pair in $\mathcal{P}$ on the graph $\mathcal{NDG}(R)$.

R is terminating with respect to $\rightsquigarrow_R$ if for every SCS $\mathcal{P}$ in $\mathcal{NDG}(R)$, there exist a reduction pair $(\succsim, \succ)$ and an argument filtering π such that (a) and

(c) for a pair $\langle s,t\rangle$ in $\mathcal{P}$, $\pi(t)$ is ground.

Proof (Sketch). Similarly to the proof of this method for rewriting, we can easily show that an infinite narrowing-chain with a ground antecedent, which is caused by an SCS $\mathcal{P}$, implies an infinite sequence of $\succ$. To guarantee the existence of a ground antecedent for the sequence of $\succ$, condition (b) is used. Note that (c) implies (b). □

Condition (a) is similar to that in the case of rewriting, and (b) guarantees the existence of ground antecedents (with respect to $\succ$) for looping by $\mathcal{P}$. For example, consider the EV-TRS $R_5 = \{\, \mathsf{g}(x) \to \mathsf{add}(y,x),\ \mathsf{add}(\mathsf{0},y) \to y,\ \mathsf{add}(\mathsf{s}(x),y) \to \mathsf{s}(\mathsf{add}(x,y)) \,\}$. It is clear that $\overset{\rightsquigarrow}{\mathrm{GA}}_{R_5}$ is not terminating. If condition (b) in Theorem 8 is missing, then termination of $\overset{\rightsquigarrow}{\mathrm{GA}}_{R_5}$ is proved. To avoid such incorrect proof, condition (b) is necessary. Condition (b) makes it difficult to introduce this method to existing termination provers based on the dependency graph method. Furthermore, condition (c) corresponds to Theorem 5 (b).

Example 9. Proving termination of $\rightsquigarrow_{R_2}$ and $\rightsquigarrow_{R_4}$ is easy by the dependency graph method because there is no SCS in their dependency graphs. Consider the EV-TRS $R_6 = \{\, \mathsf{g}(x) \to \mathsf{add}(x, y),\ \mathsf{add}(\mathsf{0}, y) \to y,\ \mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) \,\}$. The only SCS in $\mathcal{NDG}(R_6)$ is $\{\langle \mathsf{add}^\sharp(\mathsf{s}(x), y), \mathsf{add}^\sharp(x, y)\rangle\}$. Let π_6 be an argument filtering such that $\pi_6(\mathsf{add}) = [1]$, and $>$ be a precedence such that $\mathsf{g} = \mathsf{add} = \mathsf{s} = \mathsf{0} = \mathsf{add}^\sharp$. Then we have $\pi_6(R_6) \subseteq \geq_{\mathsf{lpo}}$ and $\pi_6(\{\langle \mathsf{add}^\sharp(\mathsf{s}(x), y), \mathsf{add}^\sharp(x, y)\rangle\}) \subseteq \geq_{\mathsf{lpo}} \cap >_{\mathsf{lpo}}$. Therefore, $\overset{\rightsquigarrow}{\underset{\mathrm{GA}}{}}_{R_6}$ is terminating.

As with dependency graphs for rewriting, narrowing dependency graphs are not computable in general because it is undecidable whether two dependency pairs form a narrowing-chain. However, the dependency graphs are approximations of the corresponding narrowing dependency graphs. Here, we denote the dependency graph of R by $\mathcal{DG}(R)$.

Theorem 10. *For every EV-TRS R, $\mathcal{NDG}(R)$ is a subgraph of $\mathcal{DG}(R)$.*

Note that the converse of the above theorem does not hold in general. The above theorem makes it possible for cases of narrowing to use several (computable) approximation techniques for dependency graphs, such as the *estimated dependency graphs* [1], *s-, nv- and growing-approximations* [9], *approximations based on ω- and Ω-reductions* [7], and so on, as approximations for narrowing ones.

We expect to introduce the notion of *usable rules* [4] to Theorem 8. Such an extended method will enable us to prove the termination of $\overline{\mathsf{mul}}(\mathsf{s}^n(\mathsf{0}))$ on R_1.

References

1. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theoretical Computer Science **236**(1-2) (2000) 133–178
2. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
3. Christian, J.: Some termination criteria for narrowing and e-narrowing. In: Proceedings of the 11th International Conference on Automated Deduction. Volume 607 of Lecture Notes in Computer Science., Springer (1992) 582–588
4. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Improving dependency pairs. In: Proceedings of the 10th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. Volume 2850 of Lecture Notes in Computer Science., Springer (2003) 167–182
5. Hullot, J.M.: Canonical forms and unification. In: Proceedings of the 5th International Conference on Automated Deduction. Volume 87 of Lecture Notes in Computer Science. (1980) 318–334
6. Kusakari, K., Nakamura, M., Toyama, Y.: Argument filtering transformation. In: Proceedings of the International Conference on Principles and Practice of Declarative Programming. Volume 1702 of Lecture Notes in Computer Science., Springer (1999) 47–61
7. Kusakari, K.: Termination, AC-Termination and Dependency Pairs of Term Rewriting Systems. PhD thesis, JAIST (2000)
8. Manna, Z., Ness, S.: On the termination of Markov algorithms. In: Proceedings of the Third Hawaii International Conference on System Science. (1970) 789–792
9. Middeldorp, A.: Approximating dependency graphs using tree automata techniques. In: Proceedings of the first International Joint Conference on Automated Reasoning. Volume 2083 of Lecture Notes in Computer Science., Springer (2001) 593–610
10. Nishida, N., Sakai, M., Sakabe, T.: Computation model of term rewriting systems with extra variables. Computer Software **20**(5) (2003) 85–89 (in Japanese).
11. Nishida, N., Sakai, M., Sakabe, T.: Narrowing-based simulation of term rewriting systems with extra variables and its termination proof. Electric Notes in Theoretical Computer Science **86**(3) (2003) 1–18
12. Nishida, N., Sakai, M., Sakabe, T.: Partial inversion of constructor term rewriting systems. In: Proceedings of the 16th International Conference on Rewriting Techniques and Applications. Volume 3467 of Lecture Notes in Computer Science., Springer (2005) 264–278
13. Nishida, N., Sakai, M., Sakabe, T.: Generation of inverse computation programs of constructor term rewriting systems. The IEICE Transactions on Information and Systems **J88-D-I**(8) (2005) 1171–1183 (in Japanese).
14. Ohlebusch, E.: Advanced Topics in Term Rewriting. Springer (2002)

On Non-Looping Term Rewriting

Yi Wang* and Masahiko Sakai

Graduate School of Information Science, Nagoya University, Furo-cho, Chikusa-ku, Nagoya, 4648603 Japan
{ywang80@trs.cm.,sakai@}is.nagoya-u.ac.jp

Abstract. Proving non-termination is important for instance if one wants to decide termination for given TRSs. Although the usual method is to find looping reduction sequences, there are non-looping infinite reduction sequences. We find some new interesting non-looping examples and propose new definitions of inner-looping sequence and normal sequence to classify them. We also show the undecidability of the existence of inner-looping sequence.

1 Introduction

Termination is one of the central properties of term rewriting systems (TRSs for short). We say a TRS terminates if it does not admit any infinite reduction sequences. Termination guarantees that any expression cannot be infinitely rewritten, and hence the existence of a normal form for it. Thus most researches on termination are for proving termination or for clarifying decidable classes. However, proving non-termination is also important for instance if one wants to decide termination for given TRSs.

An infinite reduction sequence often loops, that is, an instance of the starting term re-occurs as a subterm in the sequence. It is rather easy to detect loops and to give a proof of non-termination. However, some infinite reduction sequence may have no loop [5]. Its known that one-rule TRS that is non-terminating and admits no loop [8].

We give some new interesting examples and present new definitions of inner-looping sequence and normal sequence to classify them. We also show the undecidability of the existence of non-looping sequence.

2 Preliminaries

We assume the reader is familiar with the standard definitions of term rewriting systems [1]. A *signature* $\mathcal{F}$ is a set of function symbols, where every $f \in \mathcal{F}$ is associated with a non-negative integer by an arity function: $arity\colon \mathcal{F} \to \mathbb{N}(= \{0, 1, 2, \ldots\})$. The set of all *terms* built from a signature $\mathcal{F}$ and a countable infinite set $\mathcal{V}$ of *variables* such that $\mathcal{F} \cap \mathcal{V} = \emptyset$, is represented by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We write $s = t$ when two terms s and t are identical.

The set of all *positions* in a term t is denoted by $\mathcal{P}os(t)$ and ε represents the root position. The *height* $|t|$ of a term t is 0 if t is a variable or a constant, and $1 + max(\{height(s_i) \mid i \in \{1, \ldots, m\}\})$ if $t = f(s_1, \ldots, s_m)$. Let C be a *context* with a hole $\Box$. We write $C[t]$ for the term obtained from C by replacing $\Box$ with a term t. A *substitution* θ is a mapping from $\mathcal{V}$ to $\mathcal{T}(\mathcal{F}, \mathcal{V})$ such that the set $\mathrm{Dom}(\theta) = \{x \in \mathcal{V} \mid \theta(x) \neq x\}$ is finite. We usually identify a substitution θ with the set $\{x \mapsto \theta(x) \mid x \in \mathrm{Dom}(\theta)\}$ of variable bindings. We write $t\theta$ instead of $\theta(t)$.

A *rewrite rule* $l \to r$ is a directed equation which satisfies $l \notin \mathcal{V}$ and $\mathrm{Var}(r) \subseteq \mathrm{Var}(l)$. A *term rewriting system* TRS is a finite set of rewrite rules. The *reduction relation* $\to_R \subseteq \mathcal{T}(\mathcal{F}, \mathcal{V}) \times \mathcal{T}(\mathcal{F}, \mathcal{V})$ associated with a TRS R is defined as follows: $s \to_R t$ if there exist a

* Presently, with Financial Services Dept., Accenture Japan Ltd.

rewrite rule $l \to r \in R$, a substitution θ, and a context C such that $s = C[l\theta]$ and $t = C[r\theta]$. We say that s is reduced to t. The transitive closure of $\to_R$ is denoted by $\to_R^+$. The transitive and reflexive closure of $\to_R$ is denoted by $\to_R^*$. We also denote k-step reduction by $\to_R^k$.

For a TRS R, a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ *terminates* if there is no infinite reduction sequence starting from t. We say that R terminates if every term terminates.

3 Loop and non-Loop

Infinite reductions are often composed of loops. A loop is a reduction where an instance of the starting term re-occurs as a subterm. It is obvious that a loop gives an infinite reduction. In fact, the usual way to deduce non-termination is to construct a loop.

Definition 1 (Loop). A *reduction sequence loops* if it contains $t \to_R^+ C[t\theta]$ for some context C, substitution θ and term t. A *TRS R admits a loop* if there is a looping reduction sequence of R.

Example 2. Let $R_1 = \{f(x) \to h(f(g(x)))\}$. We can construct the following reduction sequence: $t = f(x) \to h(f(g(x))) \to h(h(f(g(g(x))))) \to \cdots$ which loops with $C = h[\Box]$ and $\theta = \{x \mapsto g(x)\}$.

Definition 3 (Non-Looping TRS). A rewrite sequence is *non-looping* if it is infinite and does not contain any loop. A TRS is *non-looping* if it admits an non-looping sequence. A TRS is *properly non-looping* if it is non-looping and does not admit any looping sequence.

Example 4 ([4, 5]). The following TRS R_2 is non-looping.

$$R_2 = \begin{cases} b(c) \to d(c) \\ b(d(x)) \to d(b(x)) \\ a(d(x)) \to a(b(b(x))) \end{cases}$$

The TRS R_2 has an infinite rewrite sequence: $a(b(c)) \to^2 a(b(b(c))) \to^3 a(b(b(b(c)))) \to \cdots$.

4 Inner-Loop

We propose a new definition for a certain class of non-looping sequences which covers examples in the previous section. Moreover, we present some new interesting non-looping examples which also belongs to the class we proposed. Let $\Delta^i s\delta^i = \cdots \Delta[\Delta[\Delta[s\delta]\delta]\delta]\delta \cdots$, where context Δ and substitution δ repeat i times.

Definition 5 (Inner-Looping Sequence). Given a TRS R, let s be a term, an *inner-looping sequence* is of the form:

$$C[\Delta^{l_1} s\delta^{l_1}] \to_R^+ C[\Delta^{l_2} s\delta^{l_2}] \to_R^+ \cdots \qquad (*)$$

where C and Δ are contexts, δ is a substitution, $\{l_i\}$ is an infinite sequence of natural numbers.

Obviously, a looping sequence is an inner-looping sequence where $C = \Box$,

Definition 6 (Inner-Looping TRS). A TRS R is *inner-looping* if R admits an inner-looping sequence. A TRS R is *properly inner-looping* if R is inner-looping and does not admit any looping sequence.

A modified *Post's Correspondence Problem* (mPCP for short) is defined as follows:

Let $\{\langle u_i, v_i\rangle \in \Sigma^+ \times \Sigma^+ \mid 1 \leq i \leq n\}$ be a finite set of mPCP pairs. Does there exist an solution $u_1 u_{e_1} u_{e_2} \cdots u_{e_m} = v_1 v_{e_1} v_{e_2} \cdots v_{e_m}$ where $m \geq 0$ and $e_1, e_2, \ldots, e_m \in \{1, \ldots, n\}$?

Proposition 7. *The following TRSs are properly inner-looping.*

1. R_2 *in Example 4.*
2. $R_3 = \begin{cases} f(c, a(x), y) \to f(c, x, a(y)) \\ f(c, a(x), y) \to f(x, y, a^2(c)) \end{cases}$ *[8].*
3. *Let* $\{\langle u_i, v_i\rangle \in \Sigma^+ \times \Sigma^+ \mid 1 \leq i \leq n\}$ *be an instance of mPCP having a solution. Let* $h_i \in \Sigma$*. We write* $h_1 h_2 h_3 \cdots h_n(c)$ *for* $h_1(h_2(h_3(\cdots h_n(c))))$*. Let* $u = h_1 h_2 h_3 \cdots h_n(c)$*. We use notations* $\overline{u} = h'_n h'_{n-1} h'_{n-2} \cdots h'_1(c)$ *where* h'_i *is a fresh symbol that corresponds to* h_i*.*

$$\begin{aligned} R_4 = \{&f(\overline{u_1}(c), \overline{v_1}(c), z) \to f(u_1(z), u_1(z), u_1(z))\} \\ &\cup \{f(\overline{u_i}(x), \overline{v_i}(y), z) \to f(x, y, u_i(z)) \mid 2 \leq i \leq n\} \\ &\cup \{f(u_i(x), v_i(y), z) \to f(x, y, \overline{u_i}(z)) \mid 1 \leq i \leq n\} \end{aligned} .$$

4. $$R_5 = \begin{cases} f(a(x), y, w, z) \to f(x, y, w, a(z)) \\ f(x, a(y), w, z) \to f(x, y, w, a(z)) \\ f(a(c), a(c), w, z) \to f(w, z, z, a^2(c)) \end{cases} .$$

Proof. 1. We have an inner-looping sequence in the $(*)$ form: $C = a(\Box)$, $\Delta = b(\Box)$, $s = c$, $\delta = \emptyset$, $l_i = i$. The proof for non-existence of a looping sequence is found in [5].

2. We have an inner-looping sequence in the $(*)$ form: $C = f(c, a(c), \Box)$, $\Delta = a(\Box)$, $s = c$, $\delta = \emptyset$, $l_i = i$. The proof for non-existence of a looping sequence is found in [8].

3. Let $u_1 u_{e_1} u_{e_2} \cdots u_{e_m}(c)$ be a term corresponding to a solution, denote $t_p = u_{e_1} u_{e_2} \cdots u_{e_m}$ $\overline{u_{e_m}} \cdots \overline{u_{e_2}}\, \overline{u_{e_1}}(\Box)$, then we have the following inner-looping sequence of R_5 in the $(*)$ form: $C = f(\overline{u_1}(c), \overline{v_1}(c), t_p[\overline{u_1}(\Box)])$, $\Delta = u_1(t_p[\overline{u_1}(\Box)])$, $s = c$, $\delta = \emptyset$, $l_i = 2^{i-1} - 1$.
Consider an instance of mPCP $\{\langle a, aa\rangle, \langle ab, b\rangle\}$ having a solution aab, which leads to TRS R_6:

$$R_6 = \begin{cases} f(a'(c), a'a'(c), z) \to f(a(z), a(z), a(z)) & (1) \\ f(b'a'(x), b'(y), z) \to f(x, y, ab(z)) & (2) \\ f(a(x), aa(y), z) \to f(x, y, a'(z)) & (3) \\ f(ab(x), b(y), z) \to f(x, y, b'a'(z)) & (4) \end{cases} .$$

We have $t_p = abb'a'(\Box)$, $C = f(a'(c), a'a'(c), abb'a'a'(\Box))$, $\Delta = aabb'a'a'(\Box)$ and $s = c$. Indeed, it admits an inner-looping sequence:

$$\begin{aligned} &C[\Delta^0 s\delta] = f(a'(c), a'a'(c), abb'a'a'(c)) \\ &\to f(aabb'a'a'(c), aabb'a'a'(c), aabb'a'a'(c)) \\ &\to f(abb'a'a'(c), bb'a'a'(c), a'aabb'a'a'(c)) \\ &\to f(b'a'a'(c), b'a'a'(c), b'a'a'aabb'a'a'(c)) \\ &\to f(a'(c), a'a'(c), abb'a'a'aabb'a'a'(c)) = C[\Delta^1 s\delta] \\ &\to f(aabb'a'a'aabb'a'a'(c), aabb'a'a'aabb'a'a'(c), aabb'a'a'aabb'a'a'(c)) \\ &\to \cdots . \end{aligned}$$

The non-existence of a looping sequence of R_6 is shown as follows. Since non-innermost f's in a term do not contribute to infinite sequences, it is enough to consider terms in form of $t = f(t_1, t_2, t_3)$ with no f symbol inside. Rules except (1) decrease $|t_1|$ and increase $|t_3|$ but they do not change $|t_1|+|t_3|$. Hence, the rule (1) must be used infinitely many, which also requires the groundness of t. Since the rule (1) increases $|t_1| + |t_3|$ by $|t_3|$, we have no looping sequence.

4. An inner-looping sequence of R_5 is

$$C = f(a(c), a(c), \Box_p, \Box_q),\ \Delta = a[\Box],\ s = c,\ \delta = \emptyset,\ \{l_i\} = 1, 1, 2, 3, 5, 8, \ldots$$
$$C[\Delta^{l_i} s \delta^{l_i}]_p[\Delta^{l_{i+1}} s \delta^{l_{i+1}}]_q \to^+_{R_6} C[\Delta^{l_{i+1}} s \delta^{l_{i+1}}]_p[\Delta^{l_{i+2}} s \delta^{l_{i+2}}]_q \ .$$ [1]

□

It is worth pointing out that $\{l_i\}$ for R_5 is a Fibonacci Sequence.

Note that the TRS R_3 [8] is a little bit complex because the left-hand sides are the same for constructing an one-rule example $R_7 = \{f(c, a(x), y) \to g(f(c, x, a(y)), f(x, y, a^2(c)))\}$. The TRS $R_8 = \{f(a(x), y) \to f(x, a(y)),\ f(c, y) \to f(y, a(c))\}$ is a simpler example, which has a straightforward infinite inner-looping sequence $f(c, c) \to f(c, a(c)) \to^2 f(c, a^2(c)) \to^3 f(c, a^3(c)) \to^4 \cdots$. Here, we observe that the numbers of intermediate reduction steps increase in inner-looping sequences.

So far, we defined "inner-looping" property and showed the existence of properly inner-looping TRSs. It is easy to see that either looping or inner-looping property has some *special patterns* in its infinite rewrite sequence. So naturally we want to be able to answer the following question: is there some non-looping rewrite sequence *without any patterns at all*? We give the following definition inspired by normal numbers in mathematics [2]; real numbers whose digits show a random distribution with all digits appearing equally.

Definition 8 (Normal Sequence). Given TRS R, let $t_0 \to_R t_1 \to_R \cdots \to_R t_n \to_R \cdots$ be an infinite sequence starting from t_0, denoted by $\mathcal{S}$. Let $s \in B^*$ be a context of finite symbols in base $B \subseteq \mathcal{F}$. We say context s occurs in term t if $t = C[s[t']]$ for a context C and a term t'. Denote function $N(s, n)$ to be the number of times the context s occurs in t_n. We say the sequence $\mathcal{S}$ is *normal* in base B if $\lim_{n\to\infty} \frac{N(s,n)}{n} = \frac{1}{|B|^k}$ for every s with height k ($k = 1, 2, \ldots$). A TRS R is *normal* if R admits a normal sequence.

Here "normal" says that when $n \to \infty$, in t_n every function symbol (context) shows a *random distribution* with all function symbols (contexts) appearing equally. Next proposition shows the existence of such a normal TRS.

Proposition 9. *TRS* $R_9 = R_{base} \cup R_{repeat} \cup R_{successor}$ *is normal.*

$$R_{base} = \begin{cases} f(a, x, 1(y)) \to 1(f(a, 1(x), y)) \\ f(a, x, 0(y)) \to 0(f(a, 0(x), y)) \\ f(a, x, \varepsilon) \to f(c, x, \varepsilon) \end{cases}$$
$$R_{repeat} = \begin{cases} f(b, 1(x), y) \to f(b, x, 1(y)) \\ f(b, 0(x), y) \to f(b, x, 0(y)) \\ f(b, \varepsilon, y) \to f(a, \varepsilon, y)) \end{cases}$$
$$R_{successor} = \begin{cases} f(c, 1(x), y) \to f(c, x, 0(y)) \\ f(c, 0(x), y) \to f(b, x, 1(y)) \\ f(c, \varepsilon, y) \to f(a, \varepsilon, 1(y))) \end{cases}$$

[1] This is a general case for inner-looping sequence by allowing context C to have "multi-holes".

Proof. Set base $B = \{0, 1\}$, $t_0 = f(a, \varepsilon, \varepsilon)$ starts a normal sequence of the form:

$$f(a,\varepsilon,\varepsilon) \to^* 1(f(a,1(\varepsilon),\varepsilon)) \to^* 110(f(a,01(\varepsilon),\varepsilon)) \to^* 11011(f(a,11(\varepsilon),\varepsilon))$$
$$\to^* 11011100(f(a,001(\varepsilon),\varepsilon)) \to^* 11011100101(f(a,101(\varepsilon),\varepsilon)) \to^* \cdots .$$

□

It is well known that *Champernowne's Constant* [3]: $C_2 = 0.\underline{1}\,\underline{10}\,\underline{11}\,\underline{100}\,\underline{101}\cdots$ is a normal number. Notice that the sequence in TRS R_9 is imitating C_2.

At the end of this section, we state a negative result on the decidability of the existence of inner-looping sequences.

Theorem 10. *The inner-looping property and the properly inner-looping property for TRSs R are undecidable.*

Proof. It is known that the mPCP is undecidable. Considering strictly inner-looping TRS R_4 in Proposition 7, it can be proved that there is a non-looping sequence if and only if there exists a term $u_1 u_{e_2} u_{e_2} \cdots u_{e_m}(c)$ that is corresponding to a solution of mPCP. Consider the case that the given mPCP has no solution. As stated in the proof of Proposition 7, the first rule in R_4 must be used infinitely many for an inner-looping sequence. Thus it is easy to see its impossibility. Therefore, the theorem follows from the undecidability of mPCP. □

Note that the non-looping property is undecidable [7]. The existence of proper loops, $t \to^* C[t]$ with $C \neq \Box$, is shown to be undecidable by Otto [6].

Acknowledgement

We thank the anonymous referees for giving valuable comments. This work is partly supported by MEXT.KAKENHI #18500011 and #16300005.

References

1. F. Baader and T. Nipkow. Term rewriting and all that. *Cambridge University Press*, 1998.
2. D. H. Bailey and R. E. Crandall. Random Generators and Normal Numbers. *Experimental Mathematics*, 11:527–546, 2002.
3. D. G. Champernowne. The Construction of Decimals Normal in the Scale of Ten. *J. London Mathematical Society*, 8:254–260, 1933.
4. N. Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3:69–115, 1987.
5. A. Geser and H. Zantema. Non-looping String Rewriting. *Theoret. Informatics and Appl.*, 33:279–301, 1999.
6. F. Otto. The undecidability of self-embedding for finite semi-Thue and Thue systems. *Theoretical Computer Science*, 47:225–232, 1986.
7. D. Plaisted. The undecidability of self-embedding for term rewriting systems. *Information Processing Letters*, 20:61-64, 1985.
8. H. Zantema and A. Geser. Non-looping rewriting. *Technical Report Utrecht University*, UU-CS-1996-03, 1996.

Higher-Order Dependency Pairs

Frédéric Blanqui

LORIA*, Campus Scientifique, BP 239, 54506 Vandoeuvre-lès-Nancy, France

Abstract. Arts and Giesl proved that the termination of a first-order rewrite system can be reduced to the study of its "dependency pairs". We extend these results to rewrite systems on simply typed λ-terms by using Tait's computability technique.

1 Introduction

Let $\mathcal{F}$ be a set of function symbols, $\mathcal{X}$ be a set of variables and $\mathcal{R}$ be a set of rewrite rules over the set $\mathcal{T}(\mathcal{F},\mathcal{X})$ of first-order terms. Let $\mathcal{D}$ be the set of symbols occuring at the top of a rule left hand-side and $\mathcal{C}=\mathcal{F}\setminus\mathcal{D}$. The set $\mathcal{DP}(\mathcal{R})$ of *dependency pairs* of $\mathcal{R}$ is the set of pairs (l,t) such that l is the left hand-side of a rule $l\to r\in\mathcal{R}$ and t is a subterm of r headed by some symbol $f\in\mathcal{D}$. The term t represents a potential recursive call. The chain relation is $\to_{\mathrm{C}}=\to_{\mathcal{R}i}^*\to_{\mathcal{DP}h}$, where $\to_{\mathcal{R}i}^*$ is the reflexive and transitive closure of the restriction of $\to_{\mathcal{R}}$ to non-top positions and $\to_{\mathcal{DP}h}$ is the restriction of $\to_{\mathcal{DP}}$ to top positions. Arts and Giesl prove in [1] that $\to_{\mathcal{R}}$ is strongly normalizing (SN) (or terminating, well-founded) iff the chain relation so is. Moreover, $\to_{\mathrm{C}}$ is terminating if there is a weak reduction ordering $>$ such that $\mathcal{R}\subseteq\geq$ and $\mathcal{DP}(\mathcal{R})\subseteq>$ (only dependency pairs need to strictly decrease).

We would like to extend these results to higher-order rewriting. There are several approaches to higher-order rewriting. In Higher-order Rewrite Systems (HRSs) [7], terms and rules are simply typed λ-terms in β-normal η-long form, left hand-sides are patterns à la Miller and matching is modulo $\beta\eta$. An extension of dependency pairs for HRSs is studied in [10,9]. In Combinatory Reduction Systems (CRSs) [6], terms are λ-terms, rules are λ-terms with meta-variables, left hand-sides are patterns à la Miller and matching uses α-conversion and some variable occur-checks. The relation between the two kinds of rewriting is studied in [12]. It appears that the matching algorithms are similar and that, in HRSs, one does more β-reductions after having applied the matching substitution. But, in both cases, β-reduction is used at the meta-level for normalizing right hand-sides after the application of the matching substitution. So, a third more atomic approach is to have no meta-level β-reduction and add β-reduction at the object level. This is the approach that we consider in this paper.

So, we assume given a set $\mathcal{R}$ of rewrite rules made of simply typed λ-terms and study the termination of $\to_\beta\cup\to_{\mathcal{R}}$ when using CRS-like matching. This

* UMR 7503 CNRS-INPL-INRIA-Nancy2-UHP

clearly implies the termination of $\to_{\mathcal{R}}$ in the corresponding CRS or HRS. Another advantage of this approach is that we can rely on Tait's technique for proving termination [11,3]. This paper explores its use with dependency pairs. This is in contrast with [10,9].

In Tait's technique, to each type T, one associates a set $[\![T]\!]$ of terms of type T. Terms of $[\![T]\!]$ are said *computable*. Before giving some properties of computable terms, let us introduce a few definitions. The sets $\mathrm{Pos}^+(T)$ and $\mathrm{Pos}^-(T)$ of *positive and negative positions* in T are defined as follows:

- $\mathrm{Pos}^+(B) = \{\varepsilon\}$ and $\mathrm{Pos}^-(B) = \emptyset$ if B is a base type,
- $\mathrm{Pos}^\delta(T \Rightarrow U) = 1 \cdot \mathrm{Pos}^{-\delta}(T) \cup 2 \cdot \mathrm{Pos}^\delta(U)$.

We use $\boldsymbol{T}$ to denote a sequence of types $T_1, \ldots, T_n$ of length $|\boldsymbol{T}| = n$. The i-th argument of a function symbol $f : \boldsymbol{T} \Rightarrow B$ is *accessible* if B occurs only positively in T_i. Let $\mathrm{Acc}(f)$ be the set of indexes of the accessible arguments of f. A base type B is *basic* if, for all $f : \boldsymbol{T} \Rightarrow B$ and $i \in \mathrm{Acc}(f)$, T_i is a base type. After [3,4], given a relation R, *computability wrt* R can be defined so that the following properties are satisfied:

(1) A computable term is strongly normalizable wrt $\to_\beta \cup R$.
(2) A term of basic type is computable if it is SN wrt $\to_\beta \cup R$.
(3) A term $v^{T \Rightarrow U}$ is computable if, for all t^T computable, vt is computable.
(4) If t is computable then every reduct of t is computable.
(5) A term $f\boldsymbol{t}$ is computable if all its reducts wrt $\to_\beta \cup R$ are computable.
(6) If $f\boldsymbol{t}$ is computable then, for all $i \in \mathrm{Acc}(f)$, t_i is computable.
(7) If t contains no $f \in \mathcal{D}$ and σ is computable, then $t\sigma$ is computable.
(8) Every term is computable whenever every $f \in \mathcal{D}$ is computable.

2 Admissible rules

An important property of the first-order case is that, given a term t, a substitution σ and a variable $x \in \mathcal{V}(t)$, $x\sigma$ is strongly normalizable whenever $t\sigma$ so is. This is not always true in the higher-order case. So, we need to introduce some restrictions on rules to keep this property.

Definition 1 (Admissible rules) A rule $f\boldsymbol{l} \to r$ is *admissible* if $\mathrm{FV}(r) \subseteq \mathrm{PCC}(\boldsymbol{l})$, where PCC is defined in Figure 1.

The Pattern Computability Closure (PCC) is called accessibility in [2]. It includes most usual higher-order patterns [8].

Lemma 2 If $f\boldsymbol{l} \to r$ is admissible, $\mathrm{dom}(\sigma) \subseteq \mathrm{FV}(\boldsymbol{l})$ and $\boldsymbol{l}\sigma$ is computable, then $\sigma|_{\mathrm{FV}(r)}$ is computable.

Proof. We prove by induction that, for all $u \in \mathrm{PCC}(\boldsymbol{t})$ and computable substitution θ such that $\mathrm{dom}(\theta) \subseteq \mathrm{FV}(u) \setminus \mathrm{FV}(\boldsymbol{t})$, $u\sigma\theta$ is computable.

Fig. 1. Pattern Computability Closure [2]

$$\text{(arg)}\quad t_i \in \mathrm{PCC}(\boldsymbol{t})$$

$$\text{(acc)}\quad \frac{g\boldsymbol{u} \in \mathrm{PCC}(\boldsymbol{t}) \quad i \in \mathrm{Acc}(g)}{u_i \in \mathrm{PCC}(\boldsymbol{t})}$$

$$\text{(lam)}\quad \frac{\lambda yu \in \mathrm{PCC}(\boldsymbol{t}) \quad y \notin \mathrm{FV}(\boldsymbol{t})}{u \in \mathrm{PCC}(\boldsymbol{t})}$$

$$\text{(app-left)}\quad \frac{uy \in \mathrm{PCC}(\boldsymbol{t}) \quad y \notin \mathrm{FV}(\boldsymbol{t}) \cup \mathrm{FV}(u)}{u \in \mathrm{PCC}(\boldsymbol{t})}$$

$$\text{(app-right)}\quad \frac{y^{U \Rightarrow \boldsymbol{T} \Rightarrow U} u \in \mathrm{PCC}(\boldsymbol{t}) \quad y \notin \mathrm{FV}(\boldsymbol{t}) \cup \mathrm{FV}(u)}{u \in \mathrm{PCC}(\boldsymbol{t})}$$

(arg) Since $\mathrm{dom}(\theta) = \emptyset$, $l_i\sigma\theta = l_i\sigma$ is computable by assumption.

(acc) By induction hypothesis, $g\boldsymbol{u}\sigma$ is computable. Thus, by property (6), $u_i\sigma$ is computable.

(lam) Let $\theta' = \theta|_{\mathrm{dom}(\theta)\setminus\{y\}}$. Wlog, we can assume that $y \notin \mathrm{codom}(\sigma\theta)$. Hence, $(\lambda yu)\sigma\theta' = \lambda yu\sigma\theta'$. Now, since $\mathrm{dom}(\theta) \subseteq \mathrm{FV}(u)\setminus\mathrm{FV}(\boldsymbol{t})$, $\mathrm{dom}(\theta') \subseteq \mathrm{FV}(\lambda yu) \setminus \mathrm{FV}(\boldsymbol{t})$. Thus, by induction hypothesis, $\lambda yu\sigma\theta'$ is computable. Since $y\theta$ is computable, by (3), $(\lambda yu\sigma\theta')y\theta$ is computable and, by (4), $u\sigma\theta'\{y \mapsto y\theta\}$ is computable. Finally, since $y \notin \mathrm{dom}(\sigma\theta') \cup \mathrm{codom}(\sigma\theta')$, $u\sigma\theta'\{y \mapsto y\theta\} = u\sigma\theta$.

(app-left) Let $v : T_y$ computable and $\theta' = \theta \cup \{y \mapsto v\}$. Since $\mathrm{dom}(\theta) \subseteq \mathrm{FV}(u) \setminus \mathrm{FV}(\boldsymbol{t})$ and $y \notin \mathrm{FV}(\boldsymbol{t})$, $\mathrm{dom}(\theta') = \mathrm{dom}(\theta) \cup \{y\} \subseteq \mathrm{FV}(uy) \setminus \mathrm{FV}(\boldsymbol{t})$. Thus, by induction hypothesis, $(uy)\sigma\theta' = u\sigma\theta'v$ is computable. Since $y \notin \mathrm{FV}(u)$, $u\sigma\theta' = u\sigma\theta$. Thus, $u\sigma\theta$ is computable.

(app-right) Let $v = \lambda x^U \lambda \boldsymbol{y}^{\boldsymbol{T}} x$ and $\theta' = \theta \cup \{y \mapsto v\}$. By (3), v is computable. Since $\mathrm{dom}(\theta) \subseteq \mathrm{FV}(u) \setminus \mathrm{FV}(\boldsymbol{t})$ and $y \notin \mathrm{FV}(\boldsymbol{t})$, $\mathrm{dom}(\theta') \subseteq \mathrm{FV}(yu) \setminus \mathrm{FV}(\boldsymbol{t})$. Thus, by induction hypothesis, $(yu)\sigma\theta' = vu\sigma\theta'$ is computable. Since $y \notin \mathrm{FV}(u)$, $u\sigma\theta' = u\sigma\theta$. Thus, by (4), $u\sigma\theta$ is computable. □

3 Higher-order dependency pairs

In the following, we assume given a set $\mathcal{R}$ of admissible rules. The sets $\mathrm{FAP}(t)$ of *full application positions* of a term t and the *level* of a term t are defined as follows:

- $\mathrm{FAP}(x) = \emptyset$ and $\mathrm{level}(x) = 0$
- $\mathrm{FAP}(\lambda xt) = 1 \cdot \mathrm{FAP}(t)$ and $\mathrm{level}(\lambda xt) = \mathrm{level}(t)$

If $f \in \mathcal{D}$ then:

- $\mathrm{level}(ft_1 \ldots t_n) = 1 + max\{\mathrm{level}(t_i) \mid 1 \leq i \leq n\}$

– $\mathrm{FAP}(ft_1 \dots t_n) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} 1^{n-i}2 \cdot \mathrm{FAP}(t_i)$

If $t \neq ft_1 \dots t_n$ with $f \in \mathcal{D}$, then $\mathrm{FAP}(tu) = 1 \cdot \mathrm{FAP}(t) \cup 2 \cdot \mathrm{FAP}(u)$ and $\mathrm{level}(tu) = max\{\mathrm{level}(t), \mathrm{level}(u)\}$.

Definition 3 (Dependency pairs) The set of *dependency pairs* is $\mathcal{DP} = \{l \to r|_p \mid l \to r \in \mathcal{R}, p \in \mathrm{FAP}(r)\}$. The *chain relation* is $\to_{\mathrm{C}} = \to_{\mathcal{R}i}^{*} \to_{\mathcal{DP}h}$, where $\to_{\mathcal{R}i}$ is the restriction of $\to_{\mathcal{R}}$ to non-top positions, and $\to_{\mathcal{DP}h}$ is the restriction of $\to_{\mathcal{DP}}$ to top positions.

Since $\to_{\mathrm{C}} \subseteq \to_{\mathcal{R}}^{+} \trianglerighteq$, $\to_{\beta\mathrm{C}}$ is terminating whenever $\to_{\beta\mathcal{R}}$ so is.

Theorem 4 Assume that, for all $l \to r \in \mathcal{R}$ and $p \in \mathrm{FAP}(r)$, $\mathrm{FV}(r|_p) \subseteq \mathrm{FV}(r)$ and $r|_p$ has the type of l (*). Then, $\to_{\beta\mathcal{R}}$ is terminating if $\to_{\beta\mathrm{C}}$ so is.

Proof. By (1), it suffices to prove that every term is computable. By (8), it suffices to prove that every $f^{\boldsymbol{T} \Rightarrow B} \in \mathcal{D}$ is computable. By (3), it suffices to prove that, for all $\boldsymbol{t} : \boldsymbol{T}$ computable, $f\boldsymbol{t}$ is computable. We prove it by induction on $(\boldsymbol{t}, f\boldsymbol{t})$ with $(\to_{\beta\mathcal{R}}, \to_{\beta\mathrm{C}})_{\mathrm{lex}}$ as well-founded ordering (H1). Indeed, by (1), $\boldsymbol{t}$ are strongly normalizable wrt $\to_{\beta\mathcal{R}}$. By (5), it suffices to prove that every reduct of $f\boldsymbol{t}$ is computable. If $\boldsymbol{t} \to_{\beta\mathcal{R}} \boldsymbol{t}'$ then, by (H1), $f\boldsymbol{t}'$ is computable since, by (4), $\boldsymbol{t}'$ are computable. Now, assume that there is $f\boldsymbol{l} \to r \in \mathcal{R}$ and σ such that $\boldsymbol{t} = \boldsymbol{l}\sigma$. Since rules are admissible, by Lemma 2, $\sigma' = \sigma|_{\mathrm{FV}(r)}$ is computable. We now prove that $r\sigma'$ is computable by induction on the level n of r (H2). Let $p_1, \dots, p_k$ be the positions in r of the subterms of level $n-1$; $\boldsymbol{y}^i$ be the variables of $\mathrm{FV}(r|_{p_i}) \setminus \mathrm{FV}(r)$; $x_1, \dots, x_k$ be distinct variables not occuring in r; r' be the term obtained by replacing $r|_{p_i}$ by $x_i\boldsymbol{y}^i$ in r; and $\theta = \{x_i \mapsto \lambda\boldsymbol{y}^i r|_{p_i}\sigma'\}$. We have $\mathrm{level}(r') = 0$ and $r'\sigma'\theta \to_{\beta}^{*} r\sigma'$. If θ is computable then, by (7), $r'\sigma'\theta$ is computable and we are done. By (*), $\{\boldsymbol{y}^i\} = \emptyset$ and it suffices to prove that $r_{p_i}\sigma'$ is computable. For all $i \leq k$, $r|_{p_i}$ is of the form $g\boldsymbol{u}$ with $\mathrm{level}(u_j) < n$. By (H2), $\boldsymbol{u}\sigma'$ are computable and, since $f\boldsymbol{t} \to_{\mathrm{C}} r|_{p_i}\sigma'$, by (H1), $x_i\theta$ is computable. □

The condition on free variables is an important restriction since it is not satisfied by function calls with bound variables like in $(lim\, F) + x \to lim\, \lambda n(Fn + x)$.

Theorem 5 An higher-order reduction pair is two relations $(>, \geq)$ such that:

– $>$ is well-founded and stable by substitution,

– $\geq$ is a reflexive and transitive rewrite relation containing $\to_{\beta}$,

– $\geq \circ > \subseteq >$.

If $\mathcal{R} \subseteq \geq$ and $\mathcal{DP} \subseteq >$, then $\to_{\beta\mathrm{C}}$ is well-founded.

Proof. By (1), it suffices to prove that every term is computable wrt $\to_{\mathrm{C}}$. By (8), it suffices to prove that every $f^{\boldsymbol{T} \Rightarrow B} \in \mathcal{D}$ is computable. By (3), it suffices to prove that, for all $\boldsymbol{t} : \boldsymbol{T}$ computable, $f\boldsymbol{t}$ is computable.

We prove it by induction on $(\boldsymbol{t}, f\boldsymbol{t})$ with $(\rightarrow_\beta, >)_{\text{lex}}$ as well-founded ordering (H1). Indeed, by (1), $\boldsymbol{t}$ are strongly normalizable wrt $\rightarrow_\beta$. By (5), it suffices to prove that every reduct of $f\boldsymbol{t}$ is computable. If $\boldsymbol{t} \rightarrow_\beta \boldsymbol{t'}$ then, by (H1), $f\boldsymbol{t'}$ is computable since, by (4), $\boldsymbol{t'}$ are computable. Now, assume that there is $\boldsymbol{t'}$, $f\boldsymbol{l} \rightarrow r \in \mathcal{DP}$ and σ such that $\boldsymbol{t} \rightarrow^*_{\beta\mathcal{R}} \boldsymbol{t'}$ and $\boldsymbol{t'} = \boldsymbol{l}\sigma$. Since rules are admissible, by Lemma 2, $\sigma' = \sigma|_{\text{FV}(r)}$ is computable. By (4), $\boldsymbol{t'}$ are computable. Since $\rightarrow_\beta \subseteq \geq$, $\mathcal{R} \subseteq \geq$, $\geq$ is a reflexive and transitive rewrite relation, $f\boldsymbol{t} \geq f\boldsymbol{t'}$. Since $\mathcal{DP} \subseteq >$ and $>$ is stable by substitution, $f\boldsymbol{t'} > r\sigma'$. Since $\geq \circ > \subseteq >$, $f\boldsymbol{t} > r\sigma'$. Thus, by (H1), $r\sigma'$ is computable. □

An example of reduction pair can be given by using the higher-order recursive path ordering $>_{\text{horpo}}$ [5]. Take $>= (\rightarrow_\beta \cup >_{\text{horpo}})^+$ and $\geq= (\rightarrow_\beta \cup >_{\text{horpo}})^*$. The study of these two relations has to be done. However, $>_{\text{horpo}}$ does not take advantage of the fact that $>$ does not need to be monotonic. Such a relation is given by the weak higher-order recursive computability ordering $>_{\text{whorco}}$, whose monotonic closure strictly contains $>_{\text{horpo}}$ [4]. Moreover, $>_{\text{whorco}}$ is transitive, which is not the case of $>_{\text{horpo}}$. It would therefore be interesting to look for reduction pairs built from $>_{\text{whorco}}$.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. F. Blanqui. Termination and confluence of higher-order rewrite systems. In *Proc. of RTA'00*, LNCS 1833.
3. F. Blanqui. Definitions by rewriting in the Calculus of Constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005.
4. F. Blanqui. (HO)RPO revisited, 2006. Manuscript.
5. J.-P. Jouannaud and A. Rubio. The Higher-Order Recursive Path Ordering. In *Proc. of LICS'99*.
6. J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems. *Theoretical Computer Science*, 121:279–308, 1993.
7. R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192(2):3–29, 1998.
8. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In *Proc. of ELP'89*, LNCS 475.
9. M. Sakai and K. Kusakari. On dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, E88-D(3):583–593, 2005.
10. M. Sakai, Y. Watanabe, and T. Sakabe. An extension of dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, E84-D(8):1025–1032, 2001.
11. W. W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.
12. V. van Oostrom and F. van Raamsdonk. Comparing Combinatory Reduction Systems and Higher-order Rewrite Systems. In *Proc. of HOA'93*, LNCS 816.

Normalization of Intuitionistic Set Theories

Wojciech Moczydłowski *

Department of Computer Science
Cornell University
Ithaca, NY, 14853, USA
wojtek@cs.cornell.edu

Abstract. IZF is a well-investigated impredicative constructive counterpart of Zermelo-Fraenkel set theory. We define a weakly-normalizing lambda calculus λZ corresponding to proofs in an intensional version of IZF with Replacement according to the Curry-Howard isomorphism principle. By adapting a counterexample invented by M. Crabbé, we show that λZ does not strongly normalize. Moreover, we prove that the calculus corresponding to a non-well-founded IZF does not even weakly normalize. Thus λZ and IZF are positioned on the fine line between weak, strong and lack of normalization.

1 Introduction

There are various lambda calculi corresponding to constructive theories via the propositions-as-types principle, also called the Curry-Howard isomorphism. Some famous examples are Girard's System F for second-order intuitionistic logic and F_ω for higher-order intuitionistic logic. Many powerful calculi exist which do not correspond to the existing theory, instead providing it implicitly. Prominent examples are Martin-Löf's type theory with extensions, Coquand and Huet's Calculus of Constructions and Luo's Extended Calculus of Constructions implemented in Nuprl, Coq and Lego, respectively.

Normalization of a calculus, weak or strong, is a standard tool used to justify its consistency and to provide program extraction capabilities. The more powerful the system is, the more difficult it is to prove its normalization. Girard's proof for System F cannot be formalized in the first-order arithmetic. Calculus of Inductive Constructions requires at least the proof-theoretical power of Zermelo's set theory for its normalization.

The gap between weak and strong normalization for the calculi we mentioned seems minimal. Apart from Martin-Löf's type theory, all of them strongly normalize. Many of them can be specified in the framework of Pure Type Systems and it's been conjectured by Barendregt, Geuvers and Klop that for Pure Type Systems, weak normalization entails strong normalization. Reasonable calculi have also the property that their inconsistency implies the existence of a non-normalizing term, which violates even weak normalization of the calculus.

One natural class of constructive theories for which no propositions-as-types interpretation has been known until recently ([4]) are constructive set theories. They have been investigated thoroughly. Results and bibliography concerning predicative CZF (Constructive Zermelo-Fraenkel) with variants can be found in [1] and on the authors' webpages. More powerful, impredicative IZF (Intuitionistic Zermelo-Fraenkel) has been investigated mainly before 1990 and information can be found in [5] and [2].

We introduce a lambda calculus λZ corresponding to proofs in IZF_R^-, an impredicative intensional version of IZF containing in particular the Power Set, unrestricted Separation and Replacement axioms. λZ has several interesting properties:

* Partly supported by NSF grants DUE-0333526 and 0430161.

- It corresponds to a natural constructive theory.
- It weakly normalizes. The proof method combines realizability with reduction-preserving erasure map and uses all proof-theoretic power of ZF set theory.
- It does not strongly normalize. A very small fragment of a theory suffices to exhibit the counterexample to the strong normalization property.
- A slight semantic modification to IZF_R^-, namely making it non-well-founded, makes the calculus not even weakly normalizing.

Therefore λZ and IZF_R^- are positioned on the fine line between weak, strong and lack of normalization.

Normalization of λZ makes it possible to extract programs from proofs in IZF_R^-. We describe program extraction from IZF proofs in details in [3]. More detailed proofs of our results can be found in [4].

2 $\mathbf{IZF}_R$

Intuitionistic set theory IZF_R is a first-order theory, equivalent to ZF when extended with excluded middle. The signature consists of one binary relational symbol $\in$ and function symbols used in the axioms below. The relational symbol $t = u$ is an abbreviation for $\forall z.\ z \in t \leftrightarrow z \in u$. Function symbols 0 and $S(t)$ are abbreviations for $\{x \in \omega \mid \bot\}$ and $\bigcup\{t, \{t, t\}\}$. Bounded quantifiers and the quantifier $\exists! a$ (there exists exactly one a) are also abbreviations defined in the standard way.

- (PAIR) $\forall a, b, c.\ c \in \{a, b\} \leftrightarrow c = a \vee c = b$
- (INF) $\forall c.\ c \in \omega \leftrightarrow c = 0 \vee \exists b \in \omega.\ c = S(b)$
- ($\mathrm{SEP}_{\phi(a,\overline{f})}$) $\forall \overline{f}, a, c.\ c \in S_{\phi(a,\overline{f})}(a, \overline{f}) \leftrightarrow c \in a \wedge \phi(c, \overline{f})$
- (UNION) $\forall a, c.\ c \in \bigcup a \leftrightarrow \exists b \in a.\ c \in b$
- (POWER) $\forall a, c.\ c \in P(a) \leftrightarrow \forall b.\ b \in c \to b \in a$
- ($\mathrm{REPL}_{\phi(a,b,\overline{f})}$) $\forall \overline{f}, a, c.\ c \in R_{\phi(a,b,\overline{f})}(a, \overline{f}) \leftrightarrow (\forall x \in a \exists! y. \phi(x, y, \overline{f}))) \wedge ((\exists x \in a.\ \phi(x, c, \overline{f})))$
- ($\mathrm{IND}_{\phi(a,\overline{f})}$) $\forall \overline{f}.\ (\forall a. (\forall b \in a. \phi(b, \overline{f})) \to \phi(a, \overline{f})) \to \forall a.\ \phi(a, \overline{f})$
- ($\mathrm{L}_{\phi(a,\overline{f})}$) $\forall \overline{f}, a, b.\ a = b \to \phi(a, \overline{f}) \to \phi(b, \overline{f})$

These axioms correspond very closely to the standard axiomatization of Zermelo-Fraenkel set theory ZF. The Separation axiom schema ($\mathrm{SEP}_{\phi(a,\overline{f})}$) asserts, for a given set a, the existence of the set $\{c \in a \mid \phi(c, \overline{f})\}$. Our Replacement axiom schema ($\mathrm{REPL}_{\phi(a,b,\overline{f})}$) is equivalent to the more standard formulation: "If for all $x \in a$ there is exactly one y such that $\phi(x, y, \overline{f})$ holds, then there is a set D such that $\forall x \in a \exists y.\ y \in D \wedge \phi(x, y, \overline{f})$ and for all $c \in D$ there is $x \in a$ such that $\phi(x, c, \overline{f})$". The Induction axiom schema ($\mathrm{IND}_{\phi(a,\overline{f})}$) states the principle of $\in$-induction, which in ZF follows from the fact that the membership relation $\in$ is well-founded. The Leibniz axiom schema ($\mathrm{L}_{\phi(a,\overline{f})}$) is usually present in the logic as a proof rule. However, as there is no clear way to assign a computational meaning to the rule, we instead make it a part of the axiom system.

The intensional version, which we call IZF_R^-, arises by removing the Leibniz axiom ($\mathrm{L}_{\phi(a,\overline{f})}$). It is also possible to define a normalizing lambda calculus for full, extensional IZF_R, but the presentation becomes less clear.

3 λZ

The lambda terms in λZ will be denoted by letters M, N, O, P. Letters x, y, z and a, b, c will be used for lambda variables. There are two kinds of lambda abstractions, one used for proofs of implications, the other for proofs of universal quantification. Letters t, s, u are reserved for IZF_R^- terms. The types in the system are IZF_R formulas.

$$M ::= x \mid M\ N \mid \lambda a.\ M \mid \lambda x : \phi.\ M \mid \mathrm{inl}(M) \mid \mathrm{inr}(M) \mid \mathrm{fst}(M) \mid \mathrm{snd}(M)$$

$$[t, M] \mid M\ t \mid \langle M, N\rangle \mid \mathrm{case}(M, x.N, x.O) \mid \mathrm{magic}(M) \mid \mathrm{let}\ [a, x : \phi] = M\ \mathrm{in}\ N$$

$$\mathrm{ind}_{\phi(a,\bar{b})}(M, \bar{t}) \mid \mathrm{pairProp}(t, u_1, u_2, M) \mid \mathrm{pairRep}(t, u_1, u_2, M)$$

$$\mathrm{unionProp}(t, u, M) \mid \mathrm{unionRep}(t, u, M) \mid \mathrm{sep}_{\phi(a,\overline{f})}\mathrm{Prop}(t, u, \overline{u}, M) \mid \mathrm{sep}_{\phi(a,\overline{f})}\mathrm{Rep}(t, u, \overline{u}, M)$$

$$\mathrm{powerProp}(t, u, M) \mid \mathrm{powerRep}(t, u, M) \mid \mathrm{infProp}(t, M) \mid \mathrm{infRep}(t, M)$$

$$\mathrm{repl}_{\phi(a,b,\overline{f})}\mathrm{Prop}(t, u, \overline{u}) \mid \mathrm{repl}_{\phi(a,b,\overline{f})}\mathrm{Rep}(t, u, \overline{u})$$

The ind terms correspond to the (IND) axiom, and Prop and Rep terms correspond to the respective axioms. To avoid listing all of them every time, we adopt a convention of using axRep and axProp terms to tacitly mean all Rep and Prop terms, for ax being one of pair, union, sep, power, inf and repl. With this convention in mind, we can summarize the definition of the Prop and Rep terms as:

$$\mathrm{axProp}(t, \overline{u}, M) \mid \mathrm{axRep}(t, \overline{u}, M),$$

The rest of terms correspond to the rules of the first-order logic. The nature of the correspondence is expressed by the typing system in Section 3.1.

The deterministic call-by-need reduction relation $\to$ arises from the following reduction rules and evaluation contexts. In the reduction rules for ind terms, the variable x is new.

$$(\lambda x : \phi.\ M)N \to M[x := N] \quad (\lambda a.\ M)t \to M[a := t] \quad \mathrm{fst}(\langle M, N\rangle) \to M \quad \mathrm{snd}(\langle M, N\rangle) \to N$$

$$\mathrm{case}(\mathrm{inl}(M), x.N, x.O) \to N[x := M] \qquad \mathrm{case}(\mathrm{inr}(M), x.N, x.O) \to O[x := M]$$

$$\mathrm{let}\ [a, x : \phi] = [t, M]\ \mathrm{in}\ N \to N[a := t][x := M] \qquad \mathrm{axProp}(t, \overline{u}, \mathrm{axRep}(t, \overline{u}, M)) \to M$$

$$\mathrm{ind}_{\phi(a,\bar{b})}(M, \bar{t}) \to \lambda c.\ M\ c\ (\lambda b.\lambda x : b \in c.\ \mathrm{ind}_{\phi(a,\bar{b})}(M, \bar{t})\ b)$$

$$[\circ] ::= \mathrm{fst}([\circ]) \mid \mathrm{snd}([\circ]) \mid \mathrm{case}([\circ], x.M, x.N) \mid \mathrm{axProp}(t, \overline{u}, [\circ]) \mid \mathrm{let}\ [a, y : \phi] = [\circ]\ \mathrm{in}\ N \mid [\circ]\ M$$
$$\mid \mathrm{magic}([\circ])$$

3.1 Types

The type system for λZ is constructed according to the principle of the Curry-Howard isomorphism for IZF_R^-. Types are IZF formulas. Contexts Γ are finite sets of pairs (x_1, ϕ_i). The *range* of a context Γ is the corresponding intuitionistic first-order logic context that contains only formulas and is denoted by $rg(\Gamma)$. The notation $FV_L(\Gamma)$ denotes the free logic variables of a context. The proof rules follow:

$$\frac{}{\Gamma, x : \phi \vdash x : \phi} \qquad \frac{\Gamma \vdash M : \phi \to \psi \quad \Gamma \vdash N : \phi}{\Gamma \vdash M\ N : \psi} \qquad \frac{\Gamma, x : \phi \vdash M : \psi}{\Gamma \vdash \lambda x : \phi.M : \phi \to \psi} \qquad \frac{\Gamma \vdash M : \phi \quad \Gamma \vdash N : \psi}{\Gamma \vdash \langle M, N\rangle : \phi \wedge \psi}$$

$$\frac{\Gamma \vdash M : \phi \wedge \psi}{\Gamma \vdash \mathrm{fst}(M) : \phi} \qquad \frac{\Gamma \vdash M : \phi \wedge \psi}{\Gamma \vdash \mathrm{snd}(M) : \psi} \qquad \frac{\Gamma \vdash M : \phi}{\Gamma \vdash \mathrm{inl}(M) : \phi \vee \psi} \qquad \frac{\Gamma \vdash M : \psi}{\Gamma \vdash \mathrm{inr}(M) : \phi \vee \psi}$$

$$\frac{\Gamma \vdash M : \phi \vee \psi \quad \Gamma, x : \phi \vdash N : \vartheta \quad \Gamma, x : \psi \vdash O : \vartheta}{\Gamma \vdash \mathrm{case}(M, x.N, x.O) : \vartheta} \qquad \frac{\Gamma \vdash M : \phi}{\Gamma \vdash \lambda a.\ M : \forall a.\ \phi}\ a \notin FV_L(\Gamma)$$

$$\frac{\Gamma \vdash M : \phi[a := t]}{\Gamma \vdash [t, M] : \exists a.\ \phi} \qquad \frac{\Gamma \vdash M : \bot}{\Gamma \vdash \mathrm{magic}(M) : \phi} \qquad \frac{\Gamma \vdash M : \exists a.\ \phi \quad \Gamma, x : \phi \vdash N : \psi}{\Gamma \vdash \mathrm{let}\ [a, x : \phi] := M\ in\ N : \psi}\ a \notin FV_L(\Gamma, \psi)$$

$$\frac{\Gamma \vdash M : \phi_A(t, \overline{u})}{\Gamma \vdash \mathrm{axRep}(t, \overline{u}, M) : t \in t_A(\overline{u})} \qquad \frac{\Gamma \vdash M : t \in t_A(\overline{u})}{\Gamma \vdash \mathrm{axProp}(t, \overline{u}, M) : \phi_A(t, \overline{u})} \qquad \frac{\Gamma \vdash M : \forall a.\ \phi}{\Gamma \vdash M\ t : \phi[a := t]}$$

$$\frac{\Gamma \vdash M : \forall c.\ (\forall b.\ b \in c \to \phi(b, \overline{t})) \to \phi(c, \overline{t})}{\Gamma \vdash \mathrm{ind}_{\phi(b,\overline{c})}(M, \overline{t}) : \forall a.\ \phi(a, \overline{t})}$$

Lemma 1 (Curry-Howard isomorphism). *If $\Gamma \vdash O : \phi$ then $rg(\Gamma) \vdash_{IZF_R^-} \phi$. If $\Gamma \vdash_{IZF_R^-} \phi$, then there exists a term M such that $\{(x_\phi, \phi) \mid \phi \in \Gamma\} \vdash M : \phi$.*

Progress and Preservation (Subject Reduction) can be proved for λZ in a standard way.

4 Normalization

Theorem 1 (Normalization of λZ). *If $\vdash M : \phi$, then M normalizes.*

Proof (Sketch). First, a first-order erasure map $M \to \overline{M}$ is defined from terms of λZ to untyped lambda terms. It erases all information regarding quantifiers, logic variables and terms, while preserving computational behavior. For example, $\overline{\lambda a.M} = \overline{M}$, and $\overline{[t, M]} = \overline{M}$. One can prove, in a standard way, that if $\overline{M}$ normalizes, then so does M. Second, a realizability relation $M \Vdash \phi$ is defined in such a way that the following properties hold:

- If $M \Vdash \phi$, then M normalizes.
- If $\vdash M : \phi$, then $\overline{M} \Vdash \phi$.

By combining these definitions and facts, the theorem can be proved.

If the reduction system is extended to allow reductions anywhere in terms, Theorem 1 shows only weak normalization — the existence of a terminating reduction sequence. Strong normalization (every reduction sequence terminates) does not hold. Only a weak form of the separation axiom is needed for this effect to arise:

Theorem 2 (Crabbé's counterexample). *There is a formula ϕ and term M such that $\vdash M : \phi$ and M does not strongly normalize.*

Proof. Let $t = \{x \in 0 \mid x \in x \to \bot\}$. Consider the terms:

$$N \equiv \lambda y : t \in t.\ \mathrm{snd}(\mathrm{sepProp}(t, 0, y))\ y \qquad M \equiv \lambda x : t \in 0.\ N\ (\mathrm{sepRep}(t, 0, \langle x, N \rangle))$$

Then it is easy to check that $\vdash N : t \in t \to \bot$, $\vdash M : t \in 0 \to \bot$ and that M does not strongly normalize.

This counterexample can be presented in a simpler way using higher-order rewriting on top of simply-typed lambda calculus. Consider lambda calculus with simple types, two type constants A, B, corresponding to $t \in t$ and $t \in 0$, two typed constants:

$$\text{prop} : A \to (B \wedge (A \to \bot)) \qquad \text{rep} : (B \wedge (A \to \bot)) \to A$$

and a higher-order rewriting rule $\text{prop}(\text{rep}(x)) \to x$. Let $N = \lambda y : A.\ \text{snd}(\text{prop}(y))\ y$, $M = \lambda x : B.\ N\ (\text{rep}(\langle x, N\rangle))$. Then the infinite reduction sequence is exhibited by:

$$M = \lambda x : B.\ N\ (\text{rep}(\langle x, N\rangle)) = \lambda x : B.\ (\lambda y : A.\ \text{snd}(\text{prop}(y))\ y)\ \text{rep}(\langle x, N\rangle) \to$$

$$\to \lambda x : B.\ \text{snd}(\text{prop}(\text{rep}(\langle x, N\rangle)))\ \text{rep}(\langle x, N\rangle) \to \lambda x : B.\ \text{snd}(\langle x, N\rangle)\ \text{rep}(\langle x, N\rangle) \to M$$

This is essentially the computational behavior of $\lambda x.\ \Omega$, where $\Omega = (\lambda y.\ y\ y)(\lambda y.\ y\ y)$. Modulo extra type B, the constants rep and prop along with the rewriting rule establish isomorphism of A and $A \to \bot$, which makes it less surprising, as in a type system where $A = A \to \bot$ even Ω can be typed along the same lines.

Moreover, a slight semantic modification to IZF_R^-, namely making it non-well-founded, results in a system which is not even weakly normalizing. A very small fragment is sufficient for this effect to arise. Let T be an intuitionistic set theory consisting of 2 axioms:

- (C) $\forall a.\ a \in c \leftrightarrow a = c$
- (D) $\forall a.\ a \in d \leftrightarrow a \in c \wedge a \in a \to a \in a$.

The axiom (C) introduces the constant c, which denotes a non-well-founded set. The axiom (D) is an instance of the Separation axiom, the set d introduced is $\{a \in c \mid a \in a \to a \in a\}$. The lambda calculus corresponding to T is defined just as for IZF_R^-.

Theorem 3. *There is a formula ϕ and term M such that $\vdash_T M : \phi$ and M does not weakly normalize.*

Proof. It is relatively easy to find a term O such that $\vdash_T O : d \in c$. Take $\phi = d \in d \to d \in d$. The term M below proves the claim.

$$N \equiv \lambda x : d \in d.\ \text{snd}(\text{dRep}(d, c, x))\ x \qquad M \equiv N\ (\text{dProp}(d, c, \langle O, N\rangle)).$$

Again, putting this counterexample in the framework of higher-order rewriting we have:

$$\text{prop} : A \to (B \wedge (A \to A)) \qquad \text{rep} : (B \wedge (A \to A)) \to A \qquad \text{prop}(\text{rep}(x)) \to x$$

The term M now behaves exactly as Ω, as this time the term O of type B can be produced.

Acknowledgments

I would like to thank anonymous referees for helpful comments.

References

1. Peter Aczel and Michael Rathjen. Notes on constructive set theory. Technical Report 40, Institut Mittag-Leffler (The Royal Swedish Academy of Sciences), 2000/2001.
2. Michael Beeson. *Foundations of Constructive Mathematics*. Springer-Verlag, 1985.
3. Robert Constable and Wojciech Moczydłowski. Extracting Programs from Constructive HOL Proofs via IZF Set-Theoretic Semantics. In *Proceedings of 3rd International Joint Conference on Automated Reasoning (IJCAR 2006)*. Springer, 2006. To appear.
4. Wojciech Moczydłowski. Normalization of IZF with Replacement. In *Proceedings of 15th Annual Conference of the EACSL (CSL 2006)*. Springer, 2006. To appear.
5. Andre Ŝĉedrov. Intuitionistic set theory. In *Harvey Friedman's Research on the Foundations of Mathematics*, pages 257–284. Elsevier, 1985.

The Termination Competition 2006

Claude Marché[1] and Hans Zantema[2]

[1] LRI, Université Paris-sud 11
F-91405 ORSAY Cedex, France
Claude.Marche@lri.fr
[2] Department of Computer Science, Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
h.zantema@tue.nl

Abstract. The third Termination Competition took place in June 2006. We present the background, results and conclusions of this competition.

1 Motivation and history

In the past decades several techniques have been developed to prove termination of programs and rewrite systems. In the late nineties the emphasis in this research shifted towards automation: for a new technique the final goal was not to use it by hand in order to prove termination of a number of systems, but to implement it in such a way that termination proofs could be found fully automatically using a computer. Since around 2000 several tools were developed for this goal. In 2003 the idea came up to organize a competition on these tool by developing an extensive set of termination problems called TPDB (termination problem data base), and run the tools on them and compare the results. The main objectives for such a competition were and are:

- stimulate research in this area, shifting emphasis towards automation, and
- provide a standard to compare termination techniques.

At the Workshop on Termination in 2003 in Valencia a preliminary competition was organized by Albert Rubio, together with the initial development of TPDB. Stimulated by the enthusiasm of the participants it was decided to organize an annual competition, in which all tools run on one central computer and results are reported on-line. Claude Marché took care of the organization and the development of the required tools.

All details on the termination competition including past editions, rules and all results, are found on

http://www.lri.fr/~marche/termination-competition/

The first full competition in this style run in May 2004, the week before the Workshop on Termination in Aachen, where the results were reported. There were three categories, corresponding to different input syntax: term rewriting, string rewriting and logic programs; respectively having 5, 5 and 2 participating tools. Apart from standard rewriting the category of term rewriting had some subcategories: termination modulo a theory, innermost strategy, outermost strategy, context-sensitive rewriting and conditional rewriting. In standard term rewriting the tool AProVE had the highest score of 410 termination proofs, out of 514 rewrite systems, while TTT was second with 397. Not all of the 514 systems were terminating: for 22 of them AProVE found a proof of non-termination. In string rewriting the tool TORPA had the highest score of 88 termination proofs out of 104, directly followed

by AProVE with 87. In logic programs and all term rewriting subcategories AProVE had the highest score.

In 2005 Hans Zantema joined the organization of the competition, and the second competition run in April 2005, the week before RTA in Nara. Since there was no Workshop on Termination in 2005, the results of this competition were reported at RTA. This time there were only two categories: term rewriting and string rewriting, respectively having 6 and 8 participating tools. Since string rewriting can be seen as a special case of term rewriting, all tools for term rewriting participated for the string rewriting category too. A new sub-category for relative termination was introduced, both for term rewriting and string rewriting. On the other hand subcategories were restricted to subcategories having at least two participants — otherwise it is hard to speak about a competition. For term rewriting these were standard, relative, innermost and modulo theory; for string rewriting these were standard and relative. In standard term rewriting again the tool AProVE had the highest score of 576 termination proofs, out of 773 rewrite systems, while TTT was again second, with 509. In string rewriting again the tool TORPA had the highest score of 126 termination proofs out of 153, again followed by AProVE, with 114. For both categories the improvements were not only due to extensions of TPDB. Also several termination proofs were given for systems where the 2004 version failed, showing improvements of the tools.

For both term rewriting and string rewriting the tools ending at the first place were the same in 2004 and 2005: AProVE and TORPA. The same holds for the second place: TTT and AProVE. So one might expect that the strong tools will remain strong in next editions, and it is unlikely for new tools to take over the role of strongest tools. Surprisingly, due to strong new techniques and the strong new tool Jambox the 2006 termination competition resulted in this unlikely scenario. Before going into details first we describe the rules as they applied for the 2006 edition of the competition.

2 The rules

The following rules were applied to the 2006 termination competition. They were announced several months in advance.

- Submission of new problems for TPDB is open until a few weeks before the competition, when this new TPDB is publicly available for testing and tuning the tools.
- Just before the competition participants submit
  - final versions of the tools, and
  - secret problems, up to 10 per participant per category, that are added to the version of TPDB used for the competition, but not accessible for other participants before the competition.
- All tools apply on all problems in the corresponding TPDB categories, all on the same machine. The required output of every tool is
  - "YES", followed by the text of a termination proof, or
  - "NO", followed by the text of a non-termination proof, or
  - anything else, interpreted as "DON'T KNOW".
- Execution of more than one minute for any tool on any termination problem causes a time-out, interpreted as "DON'T KNOW".
- For termination problems that are not solved within 10 seconds by any tool, a second round is hold with the same rules except that the time-out is five minutes rather than one minute. The time-out may be used as a parameter for the tool, by which the tools may use different policies or heuristics for different time-outs.

- All results are reported on-line, including generated proof text, and statistics about scores and running time.
- Any tool generating a wrong answer will be disqualified.
- There are no formal rules and consequences of being a "winner", apart from the honour of having a high or the highest score in some (sub)category.

These rules were designed in such a way that participants also being organizer had no advantage of being organizer. Just like in 2004 and 2005 Claude Marché took care of the actual running of the competition. After a short delay it started on June 12, 2006. It took around ten days to run the full competition, due to ten participating tools and nearly 2000 termination problems.

3 The tools and the categories

There were eleven participating tools:

- AProVE, developed at RWTH in Aachen, Germany, coordinated by Jürgen Giesl.
- CiME, developed at LRI, Orsay, France, coordinated by Claude Marché.
- Jambox, developed by Jörg Endrullis, starting in Leipzig, Germany, and continued at Free University in Amsterdam, The Netherlands.
- Matchbox, developed by Johannes Waldmann, at HTWK in Leipzig, Germany. Just like AProVE and Jambox this tool is not stand-alone, but makes use of the SAT solver SatELite/MiniSat.
- MultumNonMulta, developed by Dieter Hofbauer in Kassel, Germany. Uses the `glpsol` solver from the GNU Linear Programming Kit.
- MU-Term, developed at Universidad Politécnica in Valencia, Spain, coordinated by Salvador Lucas. It makes use of CiME for polynomial constraint solving.
- TALP, developed by Claus Claves and Enno Ohlebusch. It makes use of the tool CiME for proving termination by polynomial interpretations.
- TEPARLA, developed by Jeroen van der Wulp at Technische Universteit Eindhoven, The Netherlands.
- TORPA, developed by Hans Zantema at Technische Universteit Eindhoven, The Netherlands.
- TPA, developed by Adam Koprowski at Technische Universteit Eindhoven, The Netherlands.
- TTTbox, developed by Martin Korp at Innsbruck Universität, Austria.

There were three categories, subdivided in the following eight subcategories:

- Standard term rewriting.
- Context-sensitive term rewriting. This means that for every operation symbol it is specified inside which position rewriting is allowed. If for every symbol every position is allowed, then this coincides with standard term rewriting.
- Term rewriting modulo theory. This means that apart from the rewrite rules also equations are specified (usually associativity and commutativity) modulo which rewriting is done. Having no equations coincides with standard term rewriting.
- Relative termination of term rewriting. This means that two rewrite systems R, S are specified for which termination of $\to_S^* \cdot \to_R \cdot \to_S^*$ has to be proved.
- Innermost term rewriting. This means that only rewrite steps are allowed for which all proper subterms of the redex are in normal form.

- Standard string rewriting. This coincides with standard term rewriting in which all symbols have arity one.
- Relative termination of string rewriting.
- Logic programs.

The following table indicates which tool applies on which (sub)category:

tool	term rewriting					string rewr.		logic
	stand.	cont. sens.	mod. th.	rel. term.	inner-most	stand.	rel. term.	progr.
AProVE	×	×	×		×	×		×
CiME	×	×	×		×	×		
Jambox	×			×		×	×	
Matchbox	×	×	×	×	×	×	×	
MultumNonM.						×	×	
MU-Term	×	×						
TALP								×
TEPARLA	×			×		×	×	
TORPA						×	×	
TPA	×			×		×	×	
TTTbox	×					×		

4 The results

Detailed results including

- all termination problems,
- all generated proofs,
- executable code of the tools, and
- measured execution times and statistics

are available from

`http://www.lri.fr/~marche/termination-competition/2006/`

In this section we restrict to the main observations.

Since all tools execute complicated tasks it is likely that they contain bugs. For two tools (CiME and MU-Term) we detected some obviously incorrect generated proofs. We give their results below anyway, but if someone wants to use these tools, we recommend to contact the authors to get a bug-fixed version. We want to emphasize that this does not imply that all termination proofs generated by the remaining tools are correct: we did not check all thousands of generated termination proofs. As a long-term objective we see an automatic formal correctness check of the generated proofs.

The following table indicates the number of generated termination proofs for the remaining tools, divided over all (sub)categories:

	term rewriting					string rewr.		logic
	stand.	cont. sens.	mod. th.	rel. term.	inner-most	stand.	rel. term.	progr.
total number	865	133	71	45	69	322	45	325
AProVE	**638**	**60**	**53**		**66**	164		**225**
CiME	*345*	*16*	*40*		*12*	*44*		
Jambox	626			**27**		**251**	**36**	
Matchbox	395	16	8	22	14	176	33	
MultumNonM.						129	27	
MU-Term	*279*	*51*						
TALP								170
TEPARLA	355			19		101	21	
TORPA						201	28	
TPA	422			22		95	18	
TTTbox	193					75		

In this table for every category the highest score is printed in bold. Results of disqualified tools are in italics.

The largest category was the category of standard term rewriting, consisting of 865 termination problems. In this category AProVE was the strongest tool with 638 termination proofs, being 93 % of the 686 problems for which termination was proved by any tool. This result coincides with the results of 2005 and 2005 when AProVE had the highest score in this category too.

The great surprise was the tool Jambox, being a good second with 626 termination proofs. In 2004 and 2005 the second place was for TTT, which unfortunately did not participate this year. But in 2005 the difference between AProVE and TTT was nearly 100 systems, by which we may conclude safely that Jambox is stronger than TTT now. With a big distance TPA and Matchbox are third and fourth, with 422 and 395 termination proofs, respectively.

In the other subcategories of term rewriting AProVE had the highest score, just like in 2004 and 2005, except for relative termination for which AProVE did not participate. Also for logic programs AProVE was the winner.

In standard string rewriting, and for relative termination, both for term rewriting and string rewriting the highest scores were achieved by Jambox. In particular for standard string rewriting this was a surprise since both in 2004 and 2005 TORPA had the highest score. Now TORPA was second with 201 termination proofs, far behind Jambox with 251. Both in 2004 and 2005 AProVE was second in standard string rewriting, now only fourth after Matchbox.

Most proofs were found within a few seconds. For most tools the average time to find a termination proof was a few seconds; only MU-Term, TORPA and TTTbox were substantially faster. This year we had a second round for hard problems, having a time-out of five minutes rather than one minute. For the category of logic programs not a single new termination proof was found in this second round, applied on several dozens of problems. In the category of term rewriting it occurred a few times that a termination proof was found by a tool in the second round where all tools failed in the first round: 3 times for standard rewriting and once for context-sensitive and modulo theory subcategories. In the string rewriting category, the tools Jambox, MultumNonMulta and TPA found termination

proofs in a second round where all tools failed in the first round. The total number of these systems was 5, both in the subcategories standard (2) and relative termination (3).

Apart from termination proofs also non-termination proofs were generated, all by presenting a looping reduction. Not all tools had facilities for this. For logic programs no non-termination proofs were given; and for the rewriting categories hardly outside the standard subcategories. In the subcategory of standard term rewriting AProVE found the most: 103 non-termination proofs, two of which were found in the second round. The tool Matchbox was second with 85. In the subcategory of standard string rewriting Jambox found the most: 25 non-termination proofs, where AProVE and Matchbox share the second place with 12 each.

5 Conclusions and challenges

The Termination Competition 2006 was really exciting due to new developments. The most powerful new technique applied was the matrix method [2,1]. The basic idea of this technique is the same as of polynomial interpretations: find interpretations such that by doing a rewrite step the interpretation strictly decreases. The difference with polynomials is that terms are interpreted as vectors over natural numbers rather than natural numbers, and that symbols are interpreted based on matrix multiplication rather than polynomials. This technique has been implemented in Jambox, Matchbox and MultumNonMulta, among which Jambox and Matchbox use a SAT solver for searching for suitable interpretations. Among these tools Jambox was the strongest by far, due to the fact that in Jambox also many other techniques have been implemented, including quite advanced instances of the dependency pair method. In the category of term rewriting Jambox ended second, close after the winner AProVE, and in the category of string rewriting Jambox ended first, far before TORPA, the second in this category.

New this year was the second round: after finishing a first round with time limit of one minute a second round was held for the hard problems with time limit five minutes. The effect of this second round was quite limited: only for a few rewrite systems a termination proof was found where the first round failed for all tools, and similarly a few non-termination proofs.

As an important objective for the future we see an automatic formal correctness check of generated proofs. However, achieving this both requires a lot of work and agreement about formats of the proofs. As a very preliminary step this year we required full proofs as generated proofs, including references to underlying theory. However, we did not yet have facilities for verifying this.

In 2005 we presented termination of the string rewriting system consisting of the three rules $aa \to bc$, $bb \to ac$, $cc \to ab$ as an open problem, since no tool solved this system SRS/Zantema-z086, also formulated as problem 104 in the RTA open problem list. Without any doubt this challenge has stimulated the development of the new strong matrix method, by which it was solved indeed this year by Jambox.

We see two important reasons for considering the termination competition to be successful and justifying annual continuation:

- It provides an objective way to compare the power of various implementations and techniques for proving termination.
- New challenges emerge from the competition, stimulating the development of new powerful techniques.

Again this year there were several termination problems that could not be solved by any of the tools, and can serve as a new challenge. As a new pearl we want to mention the string rewriting system SRS/Waldmann-jw1, consisting of the two rules

$$bbb \to aaa, \quad aaa \to bab.$$

Termination of this system is open: neither any tool can solve it nor a proof by hand has been found. In the category of string rewriting there are several more very small systems that could be solved by none of the tools, all being added in 2006, but this one is the smallest and most symmetric.

In the category of term rewriting we want to mention the single rule TRS/HofWald-6:

$$f(f(a,x),y) \to f(f(x,f(a,y)),a)$$

in which a is a constant and x, y are variables. It is easily seen to be non-terminating, but no tool can prove it. This example, several other single term rewrite rules in TPDB, and the above mentioned string rewriting systems were found by randomly generating small systems and filtering out systems that could be solved by a number of tools. But there are also rewrite systems having some more meaning for which termination can not be proved by any of the tools. A classical one is TRS/D33-33 describing a coding of the battle of Hydra and Hercules. As a final example we mention TRS/Zantema06-while consisting of the three rules

$$f(t,x,y) \to f(g(x,y),x,s(y)), \quad g(s(x),0) \to t, \quad g(s(x),s(y)) \to g(x,y).$$

Here x, y are variables, g stands for *greater than* and t stands for *true*, by which the second and third rule are the standard rules for *greater than* over the naturals composed from 0 and s(successor). The first rule describes the obviously terminating loop `while` $x > y$ `do` $y := y + 1$. Both for TRS/D33-33 and TRS/Zantema06-while termination proofs have been found by hand.

The emphasis in the competition is in rewriting rather than termination of programs. Even in the category of logic programs the participating tools restricted to the specific technique of transforming the logic program to a term rewriting system and then prove termination of the latter. We should like to have participation by other tools not focusing on rewriting. For the next competition we plan to add new categories for Haskell programs, and for some kind of imperative programs.

We conclude by stating that everybody is welcome to suggest new problems for addition to TPDB.

References

1. J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. In U. Furbach and N. Shankar, editors, *Proceedings of the third International Joint Conference on Automated Reasoning (IJCAR)*, Lecture Notes in Computer Science. Springer, 2006.
2. D. Hofbauer and J. Waldmann. Termination of string rewriting with matrix interpretations. In F. Pfenning, editor, *Proceedings of the 17th International Conference on Rewriting Techniques and Applications (RTA)*, Lecture Notes in Computer Science. Springer, 2006.

Decomposing Terminating Rewrite Relations

Jörg Endrullis[1] and Dieter Hofbauer[2] and Johannes Waldmann[3]

[1] Department of Computer Science, Vrije Universiteit Amsterdam
De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands
joerg@few.vu.nl
[2] Mühlengasse 16, D-34125 Kassel, Germany
dieter@theory.informatik.uni-kassel.de
[3] Hochschule für Technik, Wirtschaft und Kultur (FH) Leipzig
Fachbereich IMN, Postfach 301166, D-04251 Leipzig, Germany
waldmann@imn.htwk-leipzig.de

Abstract. We decompose an arbitrary rewrite relation into the product of a context-free system and an inverse context-free system with empty right-hand sides. By requiring both of these relations to be terminating, we lose computational completeness and arrive at the class of *deleting* rewriting systems [6].

Our new treatment allows to efficiently construct the rewrite closure of a regular language with respect to deleting or *match-bounded* [4] rewriting. Previous implementations of this method either used a complete but inefficient decomposition algorithm [6] leading to impracticable resource consumption, or incomplete approximation algorithms [5]. Our new algorithm is both efficient and exact, thereby improving the power of automated termination provers that use the method of match-bounds.

1 Decomposing string rewriting systems

The class of *context-free* string rewriting systems $\mathrm{CF} = \{R \mid \forall(\ell \to r) \in R : |\ell| \leq 1\}$ and its subclass $\mathrm{CF}_0 = \{R \mid \forall(\ell \to r) \in R : |\ell| = 0\}$ will frequently occur in the sequel. Let SN denote the class of *strongly normalizing (terminating)* rewriting systems. We write R^- for $\{r \to \ell \mid (\ell \to r) \in R\}$, and for a class $\mathcal{C}$ of string rewriting systems let $\mathcal{C}^- = \{R^- \mid R \in \mathcal{C}\}$.

Definition 1. Let R be a string rewriting system over Σ, let S and T be string rewriting systems over $\Gamma \supseteq \Sigma$. Then the pair (S, T) is a *decomposition of* R if

$$\to_R^* = (\to_S^* \circ \to_T^*) \cap (\Sigma^* \times \Sigma^*).$$

If additionally $S \in \mathcal{S}$ and $T \in \mathcal{T}$ for classes of string rewriting systems $\mathcal{S}$ and $\mathcal{T}$, then (S, T) is called an $(\mathcal{S}, \mathcal{T})$-*decomposition* of R.

The set of strings over a given alphabet is a monoid w.r.t. to concatenation, but this operation is not invertible. We introduce *formal left and right inverses* of letters. For a given alphabet Σ, define alphabets $\overrightarrow{\Sigma} = \{\overrightarrow{a} \mid a \in \Sigma\}$ and $\overleftarrow{\Sigma} = \{\overleftarrow{a} \mid a \in \Sigma\}$, and let $\overline{\Sigma} = \Sigma \cup \overrightarrow{\Sigma} \cup \overleftarrow{\Sigma}$. We extend $\overrightarrow{\ }$ and $\overleftarrow{\ }$ from letters to strings by $\overrightarrow{a_1 \cdots a_n} = \overrightarrow{a_n} \cdots \overrightarrow{a_1}$ and $\overleftarrow{a_1 \cdots a_n} = \overleftarrow{a_n} \cdots \overleftarrow{a_1}$. The behaviour of inverse letters is expressed by the rewriting systems $\overrightarrow{E_\Sigma} = \{\overrightarrow{a}a \to \epsilon \mid a \in \Sigma\}$ and $\overleftarrow{E_\Sigma} = \{a\overleftarrow{a} \to \epsilon \mid a \in \Sigma\}$. We write $\overrightarrow{E}$ for $\overrightarrow{E_\Sigma}$ and $\overleftarrow{E}$ for $\overleftarrow{E_\Sigma}$ if Σ is clear from the context. Let $E = \overrightarrow{E} \cup \overleftarrow{E}$. Observe that $\overrightarrow{x}x \to_E^* \epsilon \leftarrow_E^* x\overleftarrow{x}$ for $x \in \Sigma^*$. The above construction is standard. The congruence relation generated by $\to_E$ is called the *Shamir congruence* in [7] II.6.2.

Definition 2. For string rewriting systems R and S over $\overline{\Sigma}$ write $R \curvearrowright S$ if S results from R by replacing a rule $xa \to r$ by $x \to r\overrightarrow{a}$, or replacing a rule $ax \to r$ by $x \to \overleftarrow{a}r$, where $a \in \Sigma$. Let $\sim$ denote the equivalence generated by $\curvearrowright$. We say that R and S are *conjugates* if $R \sim S$.

A finite system R has only finitely many conjugates, among them R, so the union of all its conjugates is finite. In the sequel, we denote this union by $C(R)$.

Lemma 1. *For every string rewriting system R over Σ,*

(1) $\to^*_{C(R)\cup E} \cap (\Sigma^* \times \Sigma^*) \subseteq \to^*_R$ *(correctness), and*
(2) $\to^*_R \subseteq \to^*_C \circ \to^*_E$ *(completeness), for every context-free conjugate C of R.*

Theorem 1. *Let R be a string rewriting system over Σ. Then $(C(R), E)$ is a decomposition of R, and if C is a context-free conjugate of R, then (C, E) is a $(\mathrm{CF}, \mathrm{CF}_0^-)$-decomposition of R.*

As every string rewriting system has a context-free conjugate, it always has a $(\mathrm{CF}, \mathrm{CF}_0^-)$-decomposition by Theorem 1, even a $(\mathrm{CF}, \mathrm{SN} \cap \mathrm{CF}_0^-)$-decomposition. We are especially interested in terminating decompositions.

Definition 3. A string rewriting system R over Σ is called *deleting* if there is an irreflexive partial ordering $>$ on Σ such that for each $(\ell \to r) \in R$ there is some letter a in ℓ so that for each letter b in r, $a > b$.

Lemma 2. *For a string rewriting system R, the following conditions are equivalent:*

(1) There is a terminating context-free conjugate of R.
(2) R is deleting.

Corollary 1. *Let R be a deleting string rewriting system, then*

(1) R has a $(\mathrm{SN} \cap \mathrm{CF}, \mathrm{SN} \cap \mathrm{CF}_0^-)$-decomposition, and
(2) [6] R preserves regularity and R^- preserves context-freeness.

Example 1. The rewriting system $R = \{ba \to cb, bd \to d, cd \to de, d \to \epsilon\}$ is deleting w.r.t. the ordering $a > b > c > d > e$. A terminating context-free conjugate of R is $C = \{a \to \overleftarrow{b}\, cb, b \to d\overrightarrow{d}, c \to de\overrightarrow{d}, d \to \epsilon\}$.

2 Decomposing match-bounded systems

Following [4], we annotate letters by numbers, called *match heights*, to get more detailed information on rewrite sequences. We switch to the extended alphabet $\Gamma = \Sigma \times \mathbb{N}$; often we write a_n for (a, n) in Γ. Define morphisms $\mathrm{base} : \Gamma \to \Sigma$, $\mathrm{height} : \Gamma \to \mathbb{N}$, and, for $n \in \mathbb{N}$, $\mathrm{lift}_n : \Sigma \to \Gamma$ by $\mathrm{base}(a_n) = a$, $\mathrm{height}(a_n) = n$ and $\mathrm{lift}_n(a) = a_n$. For a rewriting system R over Σ where $\epsilon \notin \mathrm{lhs}(R)$ define the rewriting system

$$\mathrm{match}(R) = \{\ell' \to \mathrm{lift}_{m+1}(r) \mid (\ell \to r) \in R,\ \mathrm{base}(\ell') = \ell,\ m = \min \mathrm{height}(\ell')\}$$

over Γ. It simulates R-rewriting as $\to^*_R = \mathrm{lift}_0 \circ \to^*_{\mathrm{match}(R)} \circ \mathrm{base}$. For a system S over $\Sigma \times \mathbb{N}$ let S_c denote the restriction of S to $\Sigma \times \{0, \ldots, c\}$. The system R is called *match-bounded by $c \in \mathbb{N}$* if $\to^*_{\mathrm{match}(R)}(\mathrm{lift}_0(\Sigma^*)) \subseteq (\Sigma \times \{0, \ldots, c\})^*$.

Each system $\mathrm{match}_c(R)$ is deleting w.r.t. the ordering defined by $a_m > b_n$ if $m < n$, and hence has a $(\mathrm{SN} \cap \mathrm{CF}, \mathrm{SN} \cap \mathrm{CF}_0^-)$-decomposition (C, E) by Theorem 1: As before, $E = \{\overrightarrow{a_i} a_i \to \epsilon,\ a_i \overleftarrow{a_i} \to \epsilon \mid a \in \Sigma,\ i \geq 0\}$, and for C we choose the union of all context-free conjugates of $\mathrm{match}(R)$ where, for each rule $u \to v$, $|u| = 1$ and $\mathrm{height}(u) \leq \min \mathrm{height}(v)$.

Due to the special and uniform structure of match(R), this decomposition can be improved. Giving up uniqueness of the inverses, we increase the "computational power" of inverses in using the rewriting system

$$E' = \{\overrightarrow{a_i}a_j \to \epsilon,\ a_j\overleftarrow{a_i} \to \epsilon \mid a \in \Sigma,\ j \geq i \geq 0\},$$

again over $\overline{\Gamma}$. In this extended sense, $\overrightarrow{a_2}$ becomes the left inverse of all letters $a_2, a_3, \ldots$, for instance. Note that $E \subseteq E'$ and $C' \subseteq C$. With these more general inverses we obtain a succinct and efficient decomposition of match(R).

$$C' = \{\mathrm{lift}_i(a) \to \mathrm{lift}_i(\overleftarrow{x})\,\mathrm{lift}_{i+1}(r)\,\mathrm{lift}_i(\overrightarrow{y}) \mid (xay \to r) \in R,\ a \in \Sigma,\ x, y \in \Sigma^*,\ i \geq 0\}$$

Theorem 2. *(C', E') is a $(\mathrm{SN} \cap \mathrm{CF}, \mathrm{SN} \cap \mathrm{CF}_0^-)$-decomposition of* match$(R)$.

Corollary 2. *(C'_c, E'_c) is a $(\mathrm{SN} \cap \mathrm{CF}, \mathrm{SN} \cap \mathrm{CF}_0^-)$-decomposition of* match$_c(R)$.

The result states that the drastic reduction from C_c to C'_c can be compensated by moderately enlarging E_c to E'_c. Note that $|C'_c| \leq |R| \cdot m \cdot c$ and $|C_c| \leq |R| \cdot m \cdot (c+1)^m$ for $m = \max\{|\ell| \mid \ell \in \mathrm{lhs}(R)\}$, whereas $|E'_c| = |\Sigma| \cdot O(c^2)$ and $|E_c| = |\Sigma| \cdot O(c)$. With respect to automata constructions, less rules in C imply smaller automata since closure modulo $\to_C$ means adding states, while more rules in E "only" increases the running time since closure modulo $\to_E$ means adding transitions while keeping the set of states.

Corollary 3. *Every match-bounded string rewriting system has a $(\mathrm{SN} \cap \mathrm{CF}, \mathrm{SN} \cap \mathrm{CF}^-)$-decomposition.*

Example 2. Take $R = \{aa \to aba\}$, and consider decompositions of match$_2(R)$. This is Example 1 from [5], which contains a $(\mathrm{SN} \cap \mathrm{CF}, \mathrm{SN} \cap \mathrm{CF}^-)$ decomposition where both parts have 7 rules. By Corollary 2 we get $C'_2 = \{a_0 \to \overleftarrow{a_0}a_1b_1a_1,\ a_0 \to a_1b_1a_1\overrightarrow{a_0},\ a_1 \to \overleftarrow{a_1}a_2b_2a_2,\ a_1 \to a_2b_2a_2\overrightarrow{a_1}\}$ with 4 rules, and $E'_2 = \{\overrightarrow{a_0}a_0 \to \epsilon,\ \overrightarrow{a_0}a_1 \to \epsilon, \ldots\}$ with 24 rules. In contrast, C_2 contains C'_2 and 6 additional rules $a_0 \to \overleftarrow{a_1}a_1b_1a_1,\ a_0 \to a_1b_1a_1\overrightarrow{a_1}, \ldots$, while $E_2 \subset E'_2$ and $|E_2| = 12$.

3 Automata

For the application of automated proofs of termination we are interested in finite automata A that represent sets of descendants with respect to match$_c(R)$.

An *automaton* (with epsilon transitions) $A = (\Sigma, Q, I, F, \delta)$ consists of an alphabet Σ, a set of states Q, sets $I, F \subseteq Q$ of initial and final states resp., and a transition relation $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$. A path $p \to_A q$ is called *ϵ-minimal* if it neither starts nor ends with an ϵ-transition.

Definition 4. An automaton A over Σ is *compatible* (resp. *exactly compatible*) with a rewriting system R over Σ and a language L over Σ if $L \subseteq \mathcal{L}(A)$ (resp. $\to_R^* (L) = \mathcal{L}(A)$) and for each pair of states $p, q \in A$ and rule $(\ell \to r) \in R$ with $p \xrightarrow{\ell}_A q$, it holds that $p \xrightarrow{r}_A q$. If we omit L, then $L = \mathcal{L}(A)$.

We will construct compatible representations of descendants of $\mathcal{L}(A)$ under rewriting. For this purpose we use non-deterministic algorithms on automata.

Definition 5. For automata A, B over Σ and states $p, q \in Q(A)$ and $w \in \Sigma^*$, we write $A \xrightarrow{(p,w,q)} B$ if B is obtained from A by adding transitions and states:

- if $|w| \leq 1$, then $Q(B) = Q(A)$ and $\delta(B) = \delta(A) \cup (p, w, q)$, and
- if $|w| > 1$, then $Q(B) = Q(A) \uplus \{s_1, \ldots, s_{|w|-1}\}$ and B contains a path labelled w from p to q along the fresh states $s_1, \ldots, s_{|w|-1}$.

Definition 6. For automata A, B over Σ and a rewriting system R over Σ, we write $A \xrightarrow{R} B$ if there exist states $p, q \in Q(A)$ and a rule $(\ell \to r) \in R$ such that there exists an ϵ-minimal path $p \xrightarrow{\ell}_A q$, $\neg(p \xrightarrow{r}_A q)$ and $A \xrightarrow{(p,r,q)} B$.

Lemma 3. *Let R be rewriting system over Σ such that (1) R is terminating and context-free, or (2) R is inverse context-free. Then $\xrightarrow{R}$ is terminating, and for all automata A,B over Σ with $A \xrightarrow{R}! B$, the automaton B is exactly compatible with R and $\mathcal{L}(A)$.*

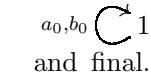
Proposition 1 (Off-line construction).
Let R be a string rewriting system with $(\mathrm{SN} \cap \mathrm{CF}, \mathrm{CF}_0^-)$-decomposition (C, E) such that C is a conjugate of R. If $A_0 \xrightarrow{C}! A_1 \xrightarrow{E}! A_2$ for automata A_0, A_1, A_2, then $A_2|_\Sigma$ is exactly compatible with R and $\mathcal{L}(A_0)$.

By $A_2|_\Sigma$ we denote the restriction of A_2 to the original alphabet. This off-line construction is inefficient since it introduces states and transitions that turn out to be unreachable later (i.e., they are in A_2, but not in $A_2|_\Sigma$). In the sequel we give a method that constructs only accessible states and transitions.

Definition 7. For a rewrite system R with $(\mathrm{SN} \cap \mathrm{CF}, \mathrm{CF}_0^-)$-decomposition (C, E) such that C is conjugate to R, and automata A, B we write $A \xrightarrow{R,C} B$ if there is a rule $(\ell \to r) \in R$ with $\ell = xay$ such that $p \xrightarrow{x} p' \xrightarrow{a} q' \xrightarrow{y} q$ is an ϵ-minimal path in A with $(p', a, q') \in \delta(A)$ and $\neg(p \xrightarrow{r}_A q)$ and $(a \to \overleftarrow{x}\, r\, \overrightarrow{y}) \in C$ such that $A \xrightarrow{(p', \overleftarrow{x}\, r\, \overrightarrow{y}, q')} B$.

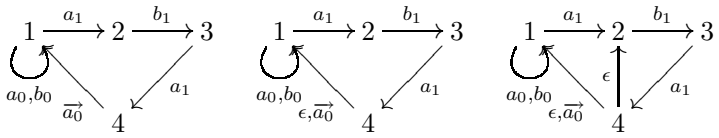
Proposition 2 (On-line construction).
For R, C, E as in Definition 7, the relation $\xrightarrow{R,C} \circ \xrightarrow{E}!$ is terminating, and for automata A over Σ, B over Γ such that $A(\xrightarrow{R,C} \circ \xrightarrow{E}!)^! B$, it holds that $B|_\Sigma$ is exactly compatible with R and $\mathcal{L}(A)$.

This construction can be used to search for a certificate of match-boundedness. Starting with an automaton A for $\mathrm{lift}_0(L)$, we use the decomposition (C', E') of $\mathrm{match}(R)$. The construction stops if and only if R is match-bounded, yielding an exactly compatible automaton in the latter case.

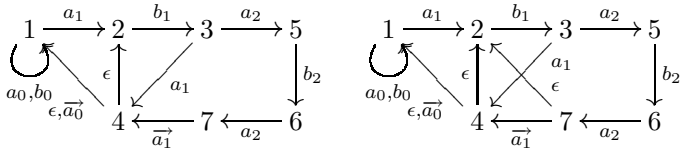
Example 3. For $R = \{aa \to aba\}$ over $\Sigma = \{a, b\}$, we show how to verify that R is match-bounded by 2 (and thus terminating) for $L = \Sigma^*$. We start with an automaton $A_0 =$ 

, representing $\mathrm{lift}_0(L)$. (For all automata in this example, state 1 is both initial and final.) A_0 contains a $\mathrm{match}(R)$-redex path $1 \xrightarrow{a_0} 1 \xrightarrow{a_0} 1$. We choose the conjugate $a_0 \to a_1 b_1 a_1 \overrightarrow{a_0}$ and add its right-hand side, getting A_1 (left). It contains two E'-redex paths $4 \xrightarrow{\overrightarrow{a_0}} 1 \xrightarrow{a_0} 1$ and $4 \xrightarrow{\overrightarrow{a_0}} 1 \xrightarrow{a_1} 2$, so we add the transitions $4 \xrightarrow{\epsilon} 1$ (middle), and $4 \xrightarrow{\epsilon} 2$ resulting

in A_3 (right).



Now there is a match(R)-redex path $3 \xrightarrow{a_1} 4 \xrightarrow{\epsilon} 1 \xrightarrow{a_1} 2$. We choose a conjugate $a_1 \to a_2 b_2 a_2 \overrightarrow{a_1}$ and add its right-hand side as a path from 3 to 4 (left). Now there is an E'-redex $7 \xrightarrow{\overrightarrow{a_1}} 4 \xrightarrow{\epsilon} 1 \xrightarrow{a_1} 2$, so we add a transition $7 \xrightarrow{\epsilon} 2$, resulting in A_5 (right). A_5 is compatible with match(R).



Example 4. To conclude, we consider the system $R = \{caac \to aaa,\ b \to aca,\ aba \to bb\}$. Jambox [3] is the only termination prover that solved this problem in the recent termination competition, see `http://www.lri.fr/~marche/termination-competition/`, problem `SRS/secret2006/jambox-1`. The implementation of our on-line algorithm constructs an exactly compatible automaton with about 27.957 states that certifies the RFC-match-bound 12. Computation time is 1,13 seconds (Athlon 3200+).

References

1. R. V. Book, M. Jantzen, and C. Wrathall. Monadic Thue systems. *Theoret. Comput. Sci.*, 19:231–251, 1982.
2. J. Endrullis. *Effiziente Algorithmen für deleting und match-bounded Wortersetzungssysteme.* Diplomarbeit, Universität Leipzig, Germany, 2005.
3. J. Endrullis. Jambox: Automated termination proofs for string rewriting. `http://joerg.endrullis.de/`
4. A. Geser, D. Hofbauer and J. Waldmann. Match-bounded string rewriting systems. *Appl. Algebra Engrg. Comm. Comput.*, 15(3-4):149-171, 2004.
5. A. Geser, D. Hofbauer, J. Waldmann, and H. Zantema. Finding finite automata that certify termination of string rewriting. *Internat. J. Found. Comput. Sci.* 16(3):471–486, 2005.
6. D. Hofbauer and J. Waldmann. Deleting string rewriting systems preserve regularity. *Theoret. Comput. Sci.*, 327(3):301–317, 2004.
7. J. Sakarovitch. *Eléments de Théorie des Automates.* Vuibert, Paris, 2003.

Termination of Extended String Rewriting

Hans Zantema

Department of Computer Science, TU Eindhoven
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
h.zantema@tue.nl

Abstract. We extend string rewriting in such a way that in the rules also variables may occur that may be replaced by arbitrary strings. This is a natural format occurring in text editing and pattern matching. We investigate termination behaviour of these extended string rewriting systems.

Let $\Sigma, \mathcal{X}$ be arbitrary disjoint sets. Elements of Σ are called *symbols* and are typically written as $a, b, c, \ldots$. Elements of $\mathcal{X}$ are called *variables* and are typically written as $X, Y, \ldots$. A *substitution* is a map $\sigma : \mathcal{X} \to \Sigma^*$. The extension of a substitution σ to $(\Sigma \cup \mathcal{X})^*$ is defined by

$$\sigma(\epsilon) = \epsilon, \;\; \sigma(aw) = a\sigma(w), \;\; \sigma(Xw) = \sigma(X)\sigma(w),$$

for $a \in \Sigma, X \in \mathcal{X}$. So applying a substitution on a string means applying the substitution on every variable in the string.

An *extended string rewrite system* (XSRS) is a subset of $(\Sigma \cup \mathcal{X})^* \times (\Sigma \cup \mathcal{X})^*$. An element (ℓ, r) of an XSRS is called a *rule* and is usually written as $\ell \to r$; the strings ℓ, r are called *left hand side* (lhs) and *right hand side* (rhs), respectively. The *rewrite relation* $\to_R$ on Σ^* of an XSRS is defined by

$$\to_R \;=\; \{(s\sigma(\ell)t, s\sigma(r)t) \mid s, t \in \Sigma^*, \; \ell \to r \in R, \; \sigma : \mathcal{X} \to \Sigma^* \; \}.$$

An SRS is defined to be an XSRS cointaining no variables.

An *infinite reduction* with respect to an XSRS R is an infinite sequence t_i of strings over Σ for $i = 1, 2, 3, \ldots$ such that $t_i \to_R t_{i+1}$ for $i = 1, 2, 3, \ldots$. An XSRS is called *terminating* if it admits no infinite reduction.

For example, the XSRS consisting of the single rule $aX \to XbX$ is not terminating due to the following infinite reduction:

$$aa \to_R aba \to_R babba \to_R bbbabbba \to_R b^6ab^4a \to_R \cdots \to_R b^{n(n+1)/2}ab^{n+1}a \to_R \cdots.$$

In our first theorem this particular non-termination behaviour is generalized. In order to prove it first we need a lemma.

Lemma 1. *Let $n > 2$, and let $a, b, c_1, c_2, \ldots, c_n \in \Sigma$. Then the rule $aXb \to c_1Xc_2X \cdots Xc_n$ admits an infinite reduction.*

Proof. Write E for the set of *embedding rules*, i.e., the rules of the shape $a \to \epsilon$ for $a \in \Sigma$. Write R_0 for the XSRS consisting of the given single rule. Due to the fact that a, b are single symbols in the lhs we obtain the following:

Claim: $\to_E \cdot \to_{R_0} \subseteq \to_{R_0} \cdot \to_E^*$, i.e., if $t \to_E u \to_{R_0} v$ then there exists w such that $t \to_{R_0} w \to_E^* v$.

For the rule $aXb \to XX$ we have the reduction

$$aaabbb \to aabbaabb \to abbaababbaab \to_E^* aaabbb.$$

Plugging extra $c_1, \ldots, c_n$ symbols, and in case of $n > 3$ also extra a, b symbols, in the first two steps gives rise to

$$aaabbb \to_{R_0} \cdot \to_{R_0} \cdot \to_E^* aaabbb,$$

yielding an infinite $R_0 \cup E$-reduction containing infinitely many R_0-steps. Applying the claim to this infinite reduction gives rise to an infinite R_0-reduction starting in *aaabbb*. In fact here we use the straightforward property in abstract reduction systems stating that non-termination of $R \cup E$, termination of E and $E \cdot R \subseteq R \cdot E^*$ yields non-termination of R. □

Theorem 1. *Let R be an XSRS containing a rule in which a variable X occurs once in the lhs and more than once in the rhs. Then R is not terminating.*

Proof. Let $uXv \to w_1Xw_2X \cdots Xw_n$ be the corresponding rule. Now the infinite reduction is obtained by replacing the symbols $a, b, c_1, c_2, \ldots, c_n$ in the infinite reduction obtained in Lemma 1 by $u, v, w_1, w_2, \ldots, w_n$, respectively, and remove possibly other variables than X. □

Working out this result concretely for explicit XSRSs yields surprisingly irregular infinite reduction, in particular non-looping.

An XSRS is called *simple* if every rule contains at most one variable, occurring at most once both in the lhs and the rhs, and occurs in the lhs if it occurs in the rhs.

Obviously a rule in which the variable occurs in the rhs but not in the lhs is non-terminating, so together with Theorem 1 the only possibly terminating non-simple XSRSs contain a rule with at least two occurrences of a variable in the lhs, like $XaX \to XbXbX$ or $aXYb \to bYXa$.

For the rest of this paper we focus on techniques for proving termination of simple XSRSs. For writing simple XSRSs we only need one variable which will always be denoted by X

Our approach is based on transforming an XSRS to an SRS in such a way that termination of the XSRS can be concluded from termination of the resulting SRS. For proving termination of an SRS tools like TORPA [5] are available.

We give two such transformations: a simple transformation Φ_s that is only sound, i.e., termination of R follows from termination of $\Phi_s(R)$ but not conversely, and a more complex transformation Φ_c that is sound and complete, i.e., R is terminating if and only if $\Phi_c(R)$ is terminating.

For a simple XSRS R the SRS $\Phi_s(R)$ is defined to consist of the following rules:

- $\ell \to r$ for every rule $\ell \to r$ in R not containing X,
- – $\ell_1 \to p_i$,
  – $p_i a \to p_i$ for every $a \in \Sigma$,
  – $p_i \ell_2 \to r$,
  for every rule $\ell_1 X \ell_2 \to r$ in R, in which ℓ_1, ℓ_2, r do not contain X, and p_i corresponds to this rule,
- – $\ell_1 \to r_1 p_i$,
  – $p_i a \to a p_i$ for every $a \in \Sigma$,
  – $p_i \ell_2 \to r_2$,
  for every rule $\ell_1 X \ell_2 \to r_1 X r_2$ in R, in which ℓ_1, ℓ_2, r_1, r_2 do not contain X, and p_i corresponds to this rule.

Here for every rule containing X a fresh symbol p_i is introduced; in case there are only a few such rules then subscripts can be omitted by writing them $p, q, r, \ldots$. So for the XSRS R consisting of the single rule $aXb \to bbXa$ the SRS $\Phi_s(R)$ consists of the rules

$$a \to bbp, \quad pa \to ap, \quad pb \to bp, \quad pa \to a,$$

while for the XSRS R consisting of the three rules $aXa \to Xb$, $bXb \to aaa$, $ab \to ba$ the SRS $\Phi_s(R)$ consists of the rules

$$a \to p, \quad pa \to ap, \quad pb \to bp, \quad pa \to b,$$

$$b \to q, \quad qa \to q, \quad qb \to q, \quad qb \to aaa, \quad ab \to ba.$$

In both case termination of $\Phi_s(R)$ is easily proved by TORPA. The next theorem states that this implies termination of the original XSRSs.

Theorem 2. *Let R be a simple XSRS for which $\Phi_s(R)$ is terminating. Then R is terminating.*

Proof. Immediate from the property that $u \to_R v$ implies $u \to^+_{\Phi_s(R)} v$ for $u, v \in \Sigma^*$, which is easily checked for all three types of rules. □

Unfortunately the converse of Theorem 2 does not hold, for instance the single rule $aXa \to aX$ is terminating, but by Φ_s the non-terminating rule $a \to ap$ is created. So Theorem 2 expresses soundness of Φ_s, but completeness does not hold. Next we give a sound and complete transformation.

Again for every rule containing X a fresh symbol p_i is introduced; moreover three more fresh symbols f, g, h are introduced. Then for a simple XSRS R the SRS $\Phi_c(R)$ is defined to consist of the following rules:

- $f\ell \to gr$ for every rule $\ell \to r$ in R not containing X,
- – $f\ell_1 \to p_i$,
  – $p_i a \to p_i$ for every $a \in \Sigma$,

  - $p_i\ell_2 \to gr$,

  for every rule $\ell_1 X \ell_2 \to r$ in R, in which ℓ_1, ℓ_2, r do not contain X, and p_i corresponds to this rule,
- - $f\ell_1 \to r_1 p_i$,
  - $p_i a \to a p_i$ for every $a \in \Sigma$,
  - $p_i \ell_2 \to g r_2$,

  for every rule $\ell_1 X \ell_2 \to r_1 X r_2$ in R, in which ℓ_1, ℓ_2, r_1, r_2 do not contain X, and p_i corresponds to this rule,
- $fa \to af$ for every $a \in \Sigma$,
- $ag \to ga$ for every $a \in \Sigma$,
- $hg \to hf$.

Lemma 2. *Let R be a simple XSRS for which $\Phi_c(R)$ admits an infinite reduction. Then $\Phi_c(R)$ admits an infinite reduction in which every string is of the shape $humv$ for $u, v \in \Sigma^*$ and $m \in \{f, g, p_i\}$.*

Proof. The existence of an infinite reduction in which every string starts by h and contains no more h's follows from type elimination [4], where the type of h is $s \to t$ and the type of all other symbols is $s \to s$.

For proving that moreover we may restrict to only one occurrence of f, g, p_i, we observe that the total number of these symbols remains constant. Let's call the part of the string right from the leftmost of these symbols the tail area. Since the rule $hg \to hf$ is not applicable here, the number of f symbols in the tail area never increases, so after removing an initial part of the infinite reduction the rules of the shape $f\ell \to gr$, $f\ell_1 \to p_i$ and $f\ell_1 \to r_1 p_i$ are not applied in the tail area. Then the number of p_i symbols never increases in the tail area, so again after removing an initial part of the infinite reduction the only rules applied in the tail area are of the shape $p_i a \to p_i$, $p_i a \to a p_i$, $fa \to af$ and $ag \to ga$. Since these form a terminating SRS, after a final initial part they are not applied at all in the tail area. Then by removing this dead tail area the desired infinite reduction is obtained. □

Lemma 3. *Let R be a simple XSRS and let $u, v \in \Sigma^*$ satisfy $hfu \to^+_{\Phi_c(R)} hgv$. Then $u \to^+_R v$.*

Proof. The SRS $\Phi_c(R)$ was constructed in such a way that starting from hfu a g symbol can only be created if $u = u_1 \sigma(\ell) u_2$ for some rule $\ell \to r$ in R and some $\sigma : \{X\} \to \Sigma^*$, by the reduction

$$hfu = hfu_1\sigma(\ell)u_2 \to^*_{\Phi_c(R)} hu_1 f\sigma(\ell)u_2 \to^+_{\Phi_c(R)} hu_1 r' u_2,$$

where either $r' = r_1 g \sigma(X) r_2$ if $r = r_1 X r_2$, or $r' = gr$ otherwise. The only way this reduction can continue is by $hu_1 r' u_2 \to^*_{\Phi_c(R)} hgu_1\sigma(r)u_2$, after which either hgv is reached or the reduction continues by $hgu_1\sigma(r)u_2 \to_{\Phi_c(R)} hfu_1\sigma(r)u_2$ and repeats the same scheme from the beginning. Observing $u_1\sigma(\ell)u_2 \to_R u_1\sigma(r)u_2$ concluded the proof. □

Theorem 3. *Let R be a simple XSRS. Then R is terminating if and only if $\Phi_c(R)$ is terminating.*

Proof. The 'if'-part follows from the property that $u \to_R v$ implies $hfu \to^+_{\Phi_c(R)} hfv$ for $u, v \in \Sigma^*$, which is easily checked for all three types of rules.

For the 'only if'-part assume $\Phi_c(R)$ admits an infinite reduction; we have to prove that R admits an infinite reduction. By Lemma 2 there is an infinite $\Phi_c(R)$-reduction in which every string is of the shape $humv$ for $u, v \in \Sigma^*$ and $m \in \{f, g, p_i\}$. Since $\Phi_c(R)$ is terminating after removing the rule $hg \to hf$ (to be seen seen by first counting the number of f symbols, and then the number of p_i symbols) this infinite reduction contains infinitely many $hg \to hf$-steps, so is of the shape

$$hgu_1 \to_{\Phi_c(R)} hfu_1 \to^+_{\Phi_c(R)} hgu_2 \to_{\Phi_c(R)} hfu_2 \to^+_{\Phi_c(R)} hgu_3 \to_{\Phi_c(R)} hfu_3 \cdots$$

From Lemma 3 we obtain an infinite R reduction $u_1 \to^+_R u_2 \to^+_R u_3 \to^+_R \cdots$, concluding the proof. □

For the rule $aXb \to bbXa$ the transformation Φ_c yields

$$fa \to bbp,\ pa \to ap,\ pb \to bp,\ pb \to ga,\ fa \to af,$$

$$fb \to bf,\ ag \to ga,\ bg \to gb,\ hg \to hf$$

and can be proved to be terminating by TORPA. However, for many other simple XSRSs like its reverse $bXa \to aXbb$ TORPA fails to prove termination of the SRS obtained by applying Φ_c. This behaviour is quite similar to context-sensitive rewriting [2] and liveness problems [1,3], both for which several transformations have been developed, and typically the more complicated complete transformations fail for practical use since the resulting rewrite systems are too complicated.

References

1. J. Giesl and H. Zantema. Liveness in rewriting. In R. Nieuwenhuis, editor, *Proceedings of the 14th Conference on Rewriting Techniques and Applications (RTA)*, volume 2706 of *Lecture Notes in Computer Science*, pages 321–336. Springer, 2003.
2. Jürgen Giesl and Aart Middeldorp. Transformation techniques for context-sensitive rewrite systems. *Journal of Functional Programming*, 14:329–427, 2004.
3. A. Koprowski and H. Zantema. Proving liveness with fairness using rewriting. In *Proceedings of the 5th International Workshop on Frontiers of Combining Systems (FroCoS '05)*, volume 3717 of *Lecture Notes in Computer Science*, pages 232–247, Berlin, 2005. Springer-Verlag.
4. H. Zantema. Termination of term rewriting: Interpretation and type elimination. *Journal of Symbolic Computation*, 17:23–50, 1994.
5. H. Zantema. Termination of string rewriting proved automatically. *Journal of Automated Reasoning*, 34:105–139, 2005. TORPA: `http://www.win.tue.nl/~hzantema/torpa.html`.

Program Termination and Well Partial Orderings

Andreas Blass[1] and Yuri Gurevich[2]

[1] Mathematics Department, University of Michigan, Ann Arbor, MI 48109, USA; ablass@umich.edu
[2] Microsoft Research, Redmond, WA 98052, USA; gurevich@microsoft.com

The following known observation may be useful in establishing program termination: if a transitive relation R is covered by finitely many well-founded relations $U_1, \ldots, U_n$ then R is well-founded. A question arises how to bound the ordinal height $|R|$ of the relation R in terms of the ordinals $\alpha_i = |U_i|$. We introduce the notion of the *stature* $\|P\|$ of a well partial ordering P and show that $|R| \leq \|\alpha_1 \times \cdots \times \alpha_n\|$. Furthermore, this bound is tight; the equality is achieved in some cases.

The notion of stature is of considerable independent interest. We define $\|P\|$ as the ordinal height of the forest of nonempty bad sequences of P, but it has many other natural and equivalent definitions. In particular, $\|P\|$ is the supremum, and in fact the maximum, of the lengths of linearizations of P. And $\|\alpha_1 \times \cdots \times \alpha_n\|$ is equal to the natural product $\alpha_1 \otimes \cdots \otimes \alpha_n$. Coming back to the question above, we have that $|R| \leq \alpha_1 \otimes \cdots \otimes \alpha_n$, and the bound is tight.

Constraints for Argument Filterings*

Harald Zankl, Nao Hirokawa, and Aart Middeldorp

Institute of Computer Science, University of Innsbruck, Austria
{Harald.Zankl,Nao.Hirokawa,Aart.Middeldorp}@uibk.ac.at

1 Introduction

Argument filterings are a key ingredient of the dependency pair method. Finding a suitable argument filtering that enables the constraints originating from the dependency pair method to be solved by a strictly monotone base order is a challenging search problem. In Section 2 we propose a simple encoding of argument filterings in propositional logic which can be easily combined with propositional encodings of simplification orders [1,3,5,6], resulting in a propositional formula with the property that any satisfying assignment corresponds to an argument filtering and the parameters of the encoded order which solve the constraints and vice-versa. We describe such a combination with the embedding order in Section 3. In Section 4 we mention some optimizations which reduce the size of the obtained propositional formulas. In order to test the effectiveness of our approach, we implemented this combination on top of the recursive SCC algorithm of [2]. For satisfiability checking we use two different methods, the state-of-the-art SAT solver MiniSat and a simple OCaml package for manipulating OBDDs. The results are compared with the divide and conquer algorithm implemented in T$\!$T and described in Section 5. In Section 6 we show how to recast a powerful result concerning argument filterings and usable rules [4] as a propositional formula, resulting in a free implementation.

2 Representing Argument Filterings

An *argument filtering* for a signature $\mathcal{F}$ is a mapping π that assigns to every n-ary function symbol $f \in \mathcal{F}$ an argument position $i \in \{1,\dots,n\}$ or a (possibly empty) list $[i_1,\dots,i_m]$ of argument positions with $1 \leqslant i_1 < \cdots < i_m \leqslant n$. The signature $\mathcal{F}_\pi$ consists of all function symbols f such that $\pi(f)$ is some list $[i_1,\dots,i_m]$, where in $\mathcal{F}_\pi$ the arity of f is m. Every argument filtering π induces a mapping from $\mathcal{T}(\mathcal{F},\mathcal{V})$ to $\mathcal{T}(\mathcal{F}_\pi,\mathcal{V})$, also denoted by π: $\pi(t) = t$ if $t \in \mathcal{V}$, $\pi(t) = \pi(t_i)$ if $t = f(t_1,\dots,t_n)$ with $\pi(f) = i$, and $\pi(t) = f(\pi(t_{i_1}),\dots,\pi(t_{i_m}))$ if $t = f(t_1,\dots,t_n)$ with $\pi(f) = [i_1,\dots,i_m]$.

Definition 1. Let $\mathcal{F}$ be a signature. The set of propositional variables $\{X_f \mid f \in \mathcal{F}\} \cup \{X_f^i \mid f^{(n)} \in \mathcal{F} \text{ and } 1 \leqslant i \leqslant n\}$ is denoted by $\mathcal{X}_\mathcal{F}$. Let π be an argument filtering for $\mathcal{F}$. The induced assignment α_π is defined as follows:

$$\alpha_\pi(X_f) = \begin{cases} \mathsf{true} & \text{if } \pi(f) = [i_1,\dots,i_m] \\ \mathsf{false} & \text{if } \pi(f) = i \end{cases} \qquad \text{and} \qquad \alpha_\pi(X_f^i) = \begin{cases} \mathsf{true} & \text{if } i \in \pi(f) \\ \mathsf{false} & \text{if } i \notin \pi(f) \end{cases}$$

for all n-ary function symbols $f \in \mathcal{F}$ and $i \in \{1,\dots,n\}$. Here $i \in \pi(f)$ if $\pi(f) = i$ or $\pi(f) = [i_1,\dots,i_m]$ and $i_k = i$ for some $1 \leqslant k \leqslant m$.

* This research is supported by FWF (Austrian Science Fund) project P18763.

Definition 2. An assignment α for $\mathcal{X}_\mathcal{F}$ is said to be *argument filtering consistent* if for every n-ary function symbol $f \in \mathcal{F}$ such that $\alpha \not\models X_f$ there is a unique $i \in \{1, \ldots, n\}$ such that $\alpha \models X_f^i$.

It is easy to see that α_π is argument filtering consistent.

Definition 3. The propositional formula $\mathsf{AF}(\mathcal{F})$ is defined as $\bigwedge_{f \in \mathcal{F}} \mathsf{AF}(f)$ with

$$\mathsf{AF}(f) = X_f \vee \bigvee_{i=1}^{\mathsf{arity}(f)} \big(X_f^i \wedge \bigwedge_{j \neq i} \neg X_f^j\big).$$

Lemma 1. *An assignment α for $\mathcal{X}_\mathcal{F}$ is argument filtering consistent if and only if $\alpha \models \mathsf{AF}(\mathcal{F})$.* $\square$

Definition 4. Let α be an argument filtering consistent assignment for $\mathcal{X}_\mathcal{F}$. The argument filtering π_α is defined as follows: $\pi_\alpha(f) = [i \mid \alpha \models X_f^i]$ if $\alpha \models X_f$ and $\pi_\alpha(f) = i$ if $\alpha \not\models X_f$ and $\alpha \models X_f^i$, for all function symbols $f \in \mathcal{F}$.

Example 1. Consider a signature consisting of two binary function symbols f and g. The assignment α with $\alpha(X_\mathsf{f}) = \alpha(X_\mathsf{f}^2) = \alpha(X_\mathsf{g}^1) = \mathsf{true}$ and $\alpha(X_\mathsf{f}^1) = \alpha(X_\mathsf{g}) = \alpha(X_\mathsf{g}^2) = \mathsf{false}$ is argument filtering consistent. The induced argument filtering π_α consists of $\pi_\alpha(\mathsf{f}) = [2]$ and $\pi_\alpha(\mathsf{g}) = 1$.

3 Embedding

In the following we define propositional formulas $\ulcorner s \rhd_{\mathsf{emb}}^\pi t \urcorner$ and $\ulcorner s \trianglerighteq_{\mathsf{emb}}^\pi t \urcorner$ which, in conjunction with $\mathsf{AF}(\mathcal{F})$, represent all argument filterings π that satisfy $\pi_\alpha(s) \rhd_{\mathsf{emb}} \pi_\alpha(t)$ and $\pi_\alpha(s) \trianglerighteq_{\mathsf{emb}} \pi_\alpha(t)$. We start with defining a formula $\ulcorner s =^\pi t \urcorner$ that represents all argument filterings which make s and t equal. (In the sequel we assume that $\wedge$ binds stronger than $\vee$.)

Definition 5. Let s and t be terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We define a propositional formula $\ulcorner s =^\pi t \urcorner$ over $\mathcal{X}_\mathcal{F}$ by induction on s and t. If $s \in \mathcal{V}$ then

$$\ulcorner s =^\pi t \urcorner = \begin{cases} \top & \text{if } s = t, \\ \bot & \text{if } t \in \mathcal{V} \text{ and } s \neq t, \\ \neg X_g \wedge \bigvee_{j=1}^{m} \big(X_g^j \wedge \ulcorner s =^\pi t_j \urcorner\big) & \text{if } t = g(t_1, \ldots, t_m). \end{cases}$$

Let $s = f(s_1, \ldots, s_n)$. If $t \in \mathcal{V}$ then $\ulcorner s =^\pi t \urcorner = \neg X_f \wedge \bigvee_{i=1}^{n} \big(X_f^i \wedge \ulcorner s_i =^\pi t \urcorner\big)$. If $t = g(t_1, \ldots, t_m)$ with $f \neq g$ then

$$\ulcorner s =^\pi t \urcorner = \neg X_f \wedge \bigvee_{i=1}^{n} \big(X_f^i \wedge \ulcorner s_i =^\pi t \urcorner\big) \vee \neg X_g \wedge \bigvee_{j=1}^{m} \big(X_g^j \wedge \ulcorner s =^\pi t_j \urcorner\big).$$

Finally, if $t = f(t_1, \ldots, t_n)$ then

$$\ulcorner s =^\pi t \urcorner = \neg X_f \wedge \bigvee_{i=1}^{n} \big(X_f^i \wedge \ulcorner s_i =^\pi t_i \urcorner\big) \vee X_f \wedge \bigwedge_{i=1}^{n} \big(X_f^i \to \ulcorner s_i =^\pi t_i \urcorner\big).$$

Definition 6. Let s and t be terms in $\mathcal{T}(\mathcal{F},\mathcal{V})$. We define propositional formulas $\ulcorner s \rhd^\pi_{\mathsf{emb}} t \urcorner$ and $\ulcorner s \trianglerighteq^\pi_{\mathsf{emb}} t \urcorner = \ulcorner s \rhd^\pi_{\mathsf{emb}} t \urcorner \vee \ulcorner s =^\pi t \urcorner$ over $\mathcal{X}_\mathcal{F}$ by induction on s and t. If $s \in \mathcal{V}$ then $\ulcorner s \rhd^\pi_{\mathsf{emb}} t \urcorner = \bot$. Let $s = f(s_1,\ldots,s_n)$. If $t \in \mathcal{V}$ then

$$\ulcorner s \rhd^\pi_{\mathsf{emb}} t \urcorner = X_f \wedge \bigvee_{i=1}^{n} \big(X^i_f \wedge \ulcorner s_i \trianglerighteq^\pi_{\mathsf{emb}} t \urcorner\big) \vee \neg X_f \wedge \bigvee_{i=1}^{n} \big(X^i_f \wedge \ulcorner s_i \rhd^\pi_{\mathsf{emb}} t \urcorner\big).$$

If $t = g(t_1,\ldots,t_m)$ with $f \neq g$ then $\ulcorner s \rhd^\pi_{\mathsf{emb}} t \urcorner$ is the disjunction of

$$X_f \wedge \Big(X_g \wedge \bigvee_{i=1}^{n} \big(X^i_f \wedge \ulcorner s_i \trianglerighteq^\pi_{\mathsf{emb}} t \urcorner\big) \vee \neg X_g \wedge \bigvee_{j=1}^{m} \big(X^j_g \wedge \ulcorner s \rhd^\pi_{\mathsf{emb}} t_j \urcorner\big)\Big)$$

and $\neg X_f \wedge \bigvee_{i=1}^{n} \big(X^i_f \wedge \ulcorner s_i \rhd^\pi_{\mathsf{emb}} t \urcorner\big)$. Finally, if $t = f(t_1,\ldots,t_n)$ then

$$\begin{aligned}\ulcorner s \rhd^\pi_{\mathsf{emb}} t \urcorner = X_f \wedge \Bigg(&\bigvee_{i=1}^{n} \big(X^i_f \wedge \ulcorner s_i \trianglerighteq^\pi_{\mathsf{emb}} t \urcorner\big) \vee \bigwedge_{i=1}^{n} \big(X^i_f \to \ulcorner s_i \trianglerighteq^\pi_{\mathsf{emb}} t_i \urcorner\big) \wedge \\ &\bigvee_{i=1}^{n} \big(X^i_f \wedge \ulcorner s_i \rhd^\pi_{\mathsf{emb}} t_i \urcorner\big)\Bigg) \vee \neg X_f \wedge \bigvee_{i=1}^{n} \big(X^i_f \wedge \ulcorner s_i \rhd^\pi_{\mathsf{emb}} t_i \urcorner\big).\end{aligned}$$

The formula $\ulcorner s \rhd^\pi_{\mathsf{emb}} t \urcorner \wedge \mathsf{AF}(\mathcal{F})$ is satisfiable if and only if there exists an argument filtering π such that $\pi(s) \rhd_{\mathsf{emb}} \pi(t)$. Even stronger, $\ulcorner s \rhd^\pi_{\mathsf{emb}} t \urcorner \wedge \mathsf{AF}(\mathcal{F})$ encodes *all* argument filterings π that satisfy $\pi(s) \rhd_{\mathsf{emb}} \pi(t)$. Analogous statements hold for $\ulcorner s =^\pi t \urcorner \wedge \mathsf{AF}(\mathcal{F})$ and $\ulcorner s \trianglerighteq^\pi_{\mathsf{emb}} t \urcorner \wedge \mathsf{AF}(\mathcal{F})$.

Lemma 2. *Let $s,t \in \mathcal{T}(\mathcal{F},\mathcal{V})$. If α is an assignment for $\mathcal{X}_\mathcal{F}$ such that $\alpha \vDash \ulcorner s \rhd^\pi_{\mathsf{emb}} t \urcorner \wedge \mathsf{AF}(\mathcal{F})$ then $\pi_\alpha(s) \rhd_{\mathsf{emb}} \pi_\alpha(t)$. If π is an argument filtering such that $\pi(s) \rhd_{\mathsf{emb}} \pi(t)$ then $\alpha_\pi \vDash \ulcorner s \rhd^\pi_{\mathsf{emb}} t \urcorner \wedge \mathsf{AF}(\mathcal{F})$.* □

4 Optimizations of the Encoding

The formulas of Section 3 are written in a way to make them easily understandable for humans. Concerning efficiency however there are quite some useful optimizations which result in a large speedup. Consider e.g. the case of different function symbols in Definition 5. The original formula

$$\ulcorner s =^\pi t \urcorner = \neg X_f \wedge \bigvee_{i=1}^{n} \big(X^i_f \wedge \ulcorner s_i =^\pi t \urcorner\big) \vee \neg X_g \wedge \bigvee_{j=1}^{m} \big(X^j_g \wedge \ulcorner s =^\pi t_j \urcorner\big)$$

can be expressed more concisely as

$$\ulcorner s =^\pi t \urcorner = \bigwedge_{i=1}^{n} \big(X^i_f \to \ulcorner s_i =^\pi t_i \urcorner\big)$$

since we know that $\mathsf{AF}(\mathcal{F})$ must hold anyway. Also the rules of commutativity, distributivity, etc. can drastically decrease the size of the formulas in Definition 6. As an example, note that the two formulas

$$\ulcorner s \rhd^\pi_{\mathsf{emb}} t \urcorner = X_f \wedge \bigvee_{i=1}^{n} \big(X^i_f \wedge \ulcorner s_i \trianglerighteq^\pi_{\mathsf{emb}} t \urcorner\big) \vee \neg X_f \wedge \bigvee_{i=1}^{n} \big(X^i_f \wedge \ulcorner s_i \rhd^\pi_{\mathsf{emb}} t \urcorner\big)$$

and

$$\ulcorner s \rhd^\pi_{\mathsf{emb}} t \urcorner = \bigvee_{i=1}^{n} \big(X^i_f \wedge \ulcorner s_i \rhd^\pi_{\mathsf{emb}} t \urcorner\big) \vee X_f \wedge \bigvee_{i=1}^{n} \big(X^i_f \wedge \ulcorner s_i =^\pi t \urcorner\big)$$

are equivalent.

5 Experimental Results

We implemented the encoding of the previous section on top of the recursive SCC algorithm with the divide and conquer approach described in [2] for combining constraints in the termination prover T$\!$T$\!$T. The generated propositional formulas are tested for satisfiability in two different ways: with the state-of-the-art SAT solver MiniSat after a linear translation (described in [5]) to CNF and with a simple package written in OCaml for manipulating OBDDs.[1] We also tested the effect of simplifying the formulas generated by the encoding before determining satisfiability, using a TRS consisting of 25 obvious simplification rules like $\neg\neg x \to x$, $x \wedge \top \to x$, and $x \wedge (x \vee y) \to x$. The results of our experiments are summarized in the table below. We used timeouts of 10 and 60 seconds for each of the 773 TRSs in the 2005 edition of the Termination Problem Data Base. All tests were performed on a server equipped with an Intel® Xeon$^{\text{TM}}$ processor running at a CPU rate of 2.40GHz and 512MB of system memory.

		SAT				OBDD			
	T$\!$T$\!$T	××	×✓	✓×	✓✓	××	×✓	✓×	✓✓
timeout 10 seconds	5	1	3	0	0	3	3	0	0
total time (in seconds)	88	147	192	61	109	77	122	53	106
timeout 60 seconds	5	0	0	0	0	2	2	0	0
total time (in seconds)	338	151	197	61	109	212	257	53	106

The T$\!$T$\!$T column refers to the divide and conquer algorithm described in [2]. The first character in the column headings ××, ×✓, ✓× and ✓✓ indicates whether the simplification rules were used and the second character whether the dynamic programming technique to select which constraints to combine next [2, Section 5.3] was used. (All methods succeeded in proving the termination of 188 of the 773 TRSs.)

Several preliminary conclusions can be drawn from the data. The performance of the approach proposed in this note outperforms the divide and conquer algorithm with dynamic programming implemented in T$\!$T$\!$T, especially for large timeouts. The cost of performing simplifications cannot be ignored but is essential for large timeouts. The dynamic programming technique enables us to share and reuse solutions of constraints. This sounds perfectly suitable for an OBDD approach, but the experimental results do not confirm this. One reason is that the merge order of [2] that we adopted in the experiments is designed for *set* representations. As addressed in [2] we need a good strategy for merging solutions to keep representations small. We anticipate that by developing a suitable strategy for BDDs one will benefit from the dynamic programming technique.

6 Extensions

Our approach extends naturally to propositional encodings of other base orders [1,3,5,6]. A different and perhaps more interesting direction is to use the propositional framework to recast existing termination criteria in order to eliminate the often considerable effort to implement these criteria. Consider e.g. the following reformulation of a technique due to [4] for computing a restricted set of usable rules based on a given argument filtering.

Theorem 1. *Let $\mathcal{R}$ be a TRS and $\mathcal{C}$ a set of the dependency pairs. There is no $\mathcal{C}$-minimal rewrite sequence if there exist an argument filtering π and a $\mathcal{C}_\mathcal{E}$-compatible reduction pair $(\gtrsim, >)$ such that $\pi(\mathcal{U}(\mathcal{C}, \pi) \cup \mathcal{C}) \subseteq \gtrsim$ and $\pi(\mathcal{C}) \cap > \neq \varnothing$.* □

[1] Posted on the Caml mailing list by Andrzej Janikowski on October 19, 2005.

Rather than giving an explicit definition of the set $\mathcal{U}(\mathcal{C},\pi)$ we encode the constraint $\pi(\mathcal{U}(\mathcal{C},\pi)\cup\mathcal{C})\subseteq\gtrsim$ as the conjunction of

$$\bigwedge_{l\to r\in\mathcal{C}}\left(U_{\mathrm{root}(l)}\wedge\ulcorner l\gtrsim^{\pi} r\urcorner\right)\ \wedge\ \bigwedge_{l\to r\in\mathcal{R}}\left(U_{\mathrm{root}(l)}\to\ulcorner l\gtrsim^{\pi} r\urcorner\right)$$

and

$$\bigwedge_{l\to r\in\mathcal{R}\cup\mathcal{C}}\left(U_{\mathrm{root}(l)}\to\bigwedge_{\substack{p\in\mathcal{P}\mathrm{os}_{\mathcal{F}}(r)\\ \mathrm{root}(r|_p)\text{ is defined}}}\left(\bigwedge_{q,\,i\colon qi\leqslant p}X^{i}_{\mathrm{root}(r|_q)}\to U_{\mathrm{root}(r|_p)}\right)\right)$$

Here U_f is a new propositional variable for every defined and every dependency pair symbol f. So by simply adding to the above constraint the encodings of the other (side) conditions we get essentially for free an implementation of a more powerful usable rule criterion than the one currently implemented in T$_{\mathsf{T}}$T (which amounts to $\pi(\mathcal{U}(\mathcal{C})\cup\mathcal{C})\subseteq\gtrsim$).

Example 2. Let $\mathcal{R}$ be the TRS consisting of the two rules

$$\begin{aligned}\mathsf{sum}(x,[\,])&\to x & 0+y&\to y\\ \mathsf{sum}(x,y::z)&\to\mathsf{sum}(x+y,z) & \mathsf{s}(x)+y&\to\mathsf{s}(x+y)\end{aligned}$$

For the dependency pair $\mathsf{SUM}(x,y::z)\to\mathsf{SUM}(x+y,z)$ none of the rewrite rules is usable under an argument filtering π with $\pi(\mathsf{SUM})=[2]$ and the dependency pair simplifies to $\mathsf{SUM}(y::z)\to\mathsf{SUM}(z)$ which can be oriented by $\rhd_{\mathsf{emb}}$ from left to right. Exactly this observation is mirrored in the last conjunction of the advanced usable rules formula that suggests that if a rule is used ($U_{root(l)}$) then a defined symbol f occurring in the right hand side of the rule gives rise to further usable rules if this symbol f "remains" after applying the argument filtering. In the example we have the subformula $U_{\mathsf{SUM}}\to(X^1_{\mathsf{SUM}}\to U_{+})$ which says that if the first argument of SUM is not deleted by the argument filtering then U_+ is set to *true* and $+$ gives rise to usable rules.

Concerning the experiments with this "advanced" usable rules criterion we could prove 59 additional TRSs terminating. For the two best performing combinations—SAT/OBDD with simplifications but without dynamic programming—the run times are 120 and 264 seconds with zero and one timeout (60 seconds) respectively.

References

1. M. Codish, V. Lagoon, and P. Stuckey. Solving partial order constraints for LPO termination. In *Proc. 17th RTA*, LNCS, 2006. To appear.
2. N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1,2):172–199, 2005.
3. M. Kurihara and H. Kondo. Efficient BDD encodings for partial order constraints with application to expert systems in software verification. In *Proc. 17th IEA/AIE*, volume 3029 of *LNCS*, pages 827–837, 2004.
4. R. Thiemann, J. Giesl, and P. Schneider-Kamp. Improved modular termination proofs using dependency pairs. In *Proc. 2nd IJCAR*, volume 3097 of *LNAI*, pages 75–90, 2004.
5. H. Zankl. SAT techniques for lexicographic path orders. Seminar report, 2006. Available at `http://arxiv.org/abs/cs.SC/0605021`.
6. H. Zankl and A. Middeldorp. KBO as a satisfaction problem. Submitted to WST 2006.

KBO as a Satisfaction Problem*

Harald Zankl and Aart Middeldorp

Institute of Computer Science, University of Innsbruck, Austria
{Harald.Zankl,Aart.Middeldorp}@uibk.ac.at

Abstract This note presents an approach to prove termination of term rewrite systems (TRSs) with the Knuth-Bendix order efficiently. The constraints for the weight function as well as for the precedence are encoded in propositional logic and the resulting formula is tested for satisfiability. Any satisfying assignment represents a weight function and a precedence such the induced Knuth-Bendix order orients the rules of the encoded TRS from left to right.

1 Introduction

This note is concerned with proving termination of term rewrite systems (TRSs) with the Knuth-Bendix order (KBO), a method invented by Knuth and Bendix well before termination research in term rewriting became a very popular and competitive endeavor (as witnessed by the annual termination competition).[1] We know of only two termination tools that contain an implementation of KBO, AProVE [4] and T$_\top$T [5], but neither of these tools incorporate KBO in their fully automatic mode. This is perhaps due to the fact that the algorithms known for deciding KBO orientability ([3,6]) are not easy to implement efficiently, despite the fact that the problem is known to be decidable in polynomial time [6]. The aim of this note is to make KBO a more attractive choice for termination tools by presenting a simple encoding of KBO orientability into propositional logic such that checking satisfiability of the resulting formula amounts to proving KBO termination.

Kurihara and Kondo [7] were the first to encode a termination method for term rewriting into propositional logic. They showed how to encode orientability with respect to the lexicographic path order as a satisfaction problem. Codish *et al.* [2] presented a more efficient formulation for the properties of a precedence. In [8] we showed how argument filterings in the dependency pair method can be encoded and combined it with the embedding order. The latter approach easily extends to other base orders which admit a propositional encoding. In this note we show that not only purely syntactical reduction orders like the lexicographic path order can be encoded into propositional logic, but also orders which additionally have a semantic component like KBO.

Section 2 presents a propositional encoding for KBO. In Section 3 we review the approach of [2] to model a strict precedence. After addressing some simple optimizations in Section 4 we compare our implementation with the one of T$_\top$T in Section 5 and show the enormous gain in efficiency.

2 KBO Encoding

We adopt the definition of KBO in [1] with the difference that we restrict the range of the weight function to $\mathbb{N}$. According to [6] this does not decrease the power of the order. In order to give a propositional encoding of KBO termination, we must take care of representing a

* This research is supported by FWF (Austrian Science Fund) project P18763.

[1] www.lri.fr/~marche/termination-competition

strict precedence and a weight function. For the former we introduce a set of new variables $X = \{X_{fg} \mid f, g \in \mathcal{F} \text{ with } f \neq g\}$ depending on the underlying signature $\mathcal{F}$ ([7]). The intended semantics of these variables is that an assignment which satisfies a variable X_{fg} corresponds to a precedence with $f \succ g$. For the weight function, symbols are considered in binary representation and the operations $>$, $=$, $\geq$, and $+$ must be redefined accordingly. The propositional encodings of $>$ and $=$ are similar to the ones in [2].

We fix the number k of bits that is available for representing natural numbers in binary. Let $a < 2^k$. We denote by $\mathbf{a} = \langle a_k, \ldots, a_1 \rangle$ the binary representation of a where a_k is the most significant bit.

Definition 1. For natural numbers given in binary representation, the operations $>$, $=$, and $\geq$ are defined as follows (for all $1 \leqslant j \leqslant k$):

$$\begin{aligned}
\ulcorner \mathbf{f} >_j \mathbf{g} \urcorner &= \begin{cases} (f_1 \wedge \neg g_1) & \text{if } j = 1 \\ (f_j \wedge \neg g_j) \vee \big((f_j \leftrightarrow g_j) \wedge \ulcorner \mathbf{f} >_{j-1} \mathbf{g} \urcorner\big) & \text{if } j > 1 \end{cases} \\
\ulcorner \mathbf{f} > \mathbf{g} \urcorner &= \ulcorner \mathbf{f} >_k \mathbf{g} \urcorner \\
\ulcorner \mathbf{f} = \mathbf{g} \urcorner &= \bigwedge_{i=1}^{k} (f_i \leftrightarrow g_i) \\
\ulcorner \mathbf{f} \geq \mathbf{g} \urcorner &= \ulcorner \mathbf{f} > \mathbf{g} \urcorner \vee \ulcorner \mathbf{f} = \mathbf{g} \urcorner
\end{aligned}$$

Next we define a formula which is satisfiable if and only if the encoded weight function is admissible for the encoded precedence.

Definition 2. For a weight function (w, w_0), let $\mathrm{adm}(w, w_0)$ be the formula

$$\ulcorner \mathbf{w_0} > \mathbf{0} \urcorner \wedge \bigwedge_{c \in \mathcal{F}^{(0)}} \ulcorner \mathbf{c} \geq \mathbf{w_0} \urcorner \wedge \bigwedge_{f \in \mathcal{F}^{(1)}} \big(\ulcorner \mathbf{f} = \mathbf{0} \urcorner \rightarrow \bigwedge_{g \in \mathcal{F},\, f \neq g} X_{fg}\big).$$

For addition we use pairs. The first component represents the bit representation and the second component is a propositional formula which encodes the constraints for each digit.

Definition 3. We define $\ulcorner (\mathbf{f}, \varphi) + (\mathbf{g}, \psi) \urcorner$ as $(\mathbf{s}, \varphi \wedge \psi \wedge \gamma \wedge \sigma)$ with

$$\gamma = \neg c_k \wedge \neg c_0 \wedge \bigwedge_{i=1}^{k} \big(c_i \leftrightarrow ((f_i \wedge g_i) \vee (f_i \wedge c_{i-1}) \vee (g_i \wedge c_{i-1}))\big)$$

and

$$\sigma = \bigwedge_{i=1}^{k} \big(s_i \leftrightarrow (f_i \oplus g_i \oplus c_{i-1})\big)$$

where c_i $(0 \leqslant i \leqslant k)$ and s_i $(1 \leqslant i \leqslant k)$ are fresh variables that represent the carry and the sum of the addition. The condition $\neg c_k$ prevents a possible overflow.

Note that although theoretically not necessary, it is a good idea to introduce new variables for the sum. The reason is that in consecutive additions each bit f_i and g_i is duplicated (twice for the carry and once for the sum) and consequently using fresh variables for the sum prevents an exponential blowup of the resulting formula.

Definition 4. We define $\ulcorner (\mathbf{f}, \varphi) > (\mathbf{g}, \psi) \urcorner$ as $\ulcorner \mathbf{f} > \mathbf{g} \urcorner \wedge \varphi \wedge \psi$.

In the next definition we show how the weight of terms is computed propositionally.

Definition 5. Let t be a term and (w, w_0) a weight function. The weight of a term is encoded as follows:

$$W_t = \begin{cases} (\mathbf{w_0}, \top) & \text{if } t \in \mathcal{V}, \\ \ulcorner(\mathbf{f}, \top) + \sum_{i=1}^{n} W_{t_i}\urcorner & \text{if } t = f(t_1, \ldots, t_n). \end{cases}$$

We are now ready to define a propositional formula that reflects the definition of $\succ_{\text{kbo}}$.

Definition 6. Let s and t be terms. We define the formula $\ulcorner s \succ_{\text{kbo}} t\urcorner$ as follows. If $s \in \mathcal{V}$ or $s = t$ or both $t \in \mathcal{V}$ and $t \notin \mathcal{V}\mathsf{ar}(s)$ or $|s|_x < |t|_x$ for some $x \in \mathcal{V}$ then $\ulcorner s \succ_{\text{kbo}} t\urcorner = \bot$. Otherwise

$$\ulcorner s \succ_{\text{kbo}} t\urcorner = \ulcorner W_s > W_t\urcorner \vee \left(\ulcorner W_s = W_t\urcorner \wedge \ulcorner s \succ'_{\text{kbo}} t\urcorner\right)$$

with

$$\ulcorner s \succ'_{\text{kbo}} t\urcorner = \begin{cases} \top & \text{if } s = f^n(t) \text{ with } t \in \mathcal{V} \text{ and } n > 0 \\ \ulcorner s_i \succ_{\text{kbo}} t_i\urcorner & \text{if } s = f(s_1, \ldots, s_n) \text{ and } t = f(t_1, \ldots, t_n) \\ X_{fg} & \text{if } s = f(s_1, \ldots, s_n),\ t = g(t_1, \ldots, t_m), \text{ and } f \neq g \end{cases}$$

where in the second clause i denotes the least $1 \leqslant j \leqslant n$ such that $s_j \neq t_j$.

3 Encoding the Precedence

To ensure the properties of a strict precedence we follow the approach of Codish *et al.* [2] who propose to interpret function symbols as natural numbers. The greater than relation ($>$) then ensures that the function symbols are properly ordered. Let $|\mathcal{F}| = n$. Then we are looking for a mapping $m\colon \mathcal{F} \to \{1, \ldots, n\}$ such that for every propositional variable $X_{fg} \in X$ we have $m(f) > m(g)$. To uniquely encode one of the n function symbols, $l := \lceil log_2(n) \rceil$ fresh propositional variables are needed. The l-bit representation of f is $\langle f_l, \ldots, f_1 \rangle$ with f_l the most significant bit.

Definition 7. For all $1 \leqslant j \leqslant l$:

$$||X_{fg}||_j = \begin{cases} (f_1 \wedge \neg g_1) & \text{if } j = 1, \\ (f_j \wedge \neg g_j) \vee \left((f_j \leftrightarrow g_j) \wedge ||X_{fg}||_{j-1}\right) & \text{if } j > 1. \end{cases}$$

After this step it is rather easy to define a propositional formula which is satisfiable whenever the TRS is KBO terminating and sufficiently many bits are allowed for the semantic part.

Definition 8. Let $\mathcal{R}$ be a TRS. The formula $\text{KBO}(\mathcal{R})$ is defined as

$$\text{adm}(w, w_0) \wedge \bigwedge_{l \to r \in \mathcal{R}} \ulcorner l \succ_{\text{kbo}} r\urcorner \ \wedge \bigwedge_{x \in X} (x \leftrightarrow ||x||_l).$$

Theorem 1. *A TRS $\mathcal{R}$ is KBO terminating whenever the propositional formula* $\text{KBO}(\mathcal{R})$ *is satisfiable.* □

method(#bits)	total time	# successes	# timeouts
kbo-sat(2)	12.72	72	0
kbo-sat(3)	15.21	77	0
kbo-sat(4)	18.77	78	0
kbo-sat(10)	126.03	78	1
TTT	255.12	76	3

Table 1. KBO for 823 TRSs.

Since we do not know in advance the number k of bits needed to represent the weights, satisfiability of KBO($\mathcal{R}$) is not a necessary condition for KBO termination. As already mentioned in the introduction, the problem of finding correct weights can be solved in polynomial time. Nevertheless our exponential algorithm is more powerful in practice. In Section 5 we shall see that for all systems in the Termination Problem Data Base a rather small number of bits suffices.

4 Optimizations

When computing the constraints for the weights for terms s and t, removing function symbols and variables that occur both in s and in t is highly recommended. Concerning the admissibility condition, testing whether a function symbol f has weight zero can be expressed more concisely as $\neg(f_1 \vee \cdots \vee f_k)$. This similarly works for the constraint $\ulcorner \mathbf{w_0} > \mathbf{0} \urcorner$. Another optimization is the simplification of the constraint formula KBO($\mathcal{R}$) using equalities like $\neg\neg x \to x$, $x \wedge \top \to x$, and $x \wedge (x \vee y) \to x$.

5 Experimental Results

We implemented our encoding on top of TTT. For testing satisfiability MiniSat was employed because it produced considerably faster times than an approach based on binary decision diagrams. Table 1 compares our implementation (kbo-sat) of KBO with the one in TTT. AProVE, the only other known termination tool that contains an implementation of KBO, is not considered in the table because it produced seriously slower results than TTT. We used the 823 TRSs in version 3.1 of the Termination Problem Data Base[2] that do not specify any strategy or theory. All tests were performed on a server equipped with an Intel® Xeon™ processor running at a CPU rate of 2.40GHz and 512MB of system memory.

As addressed in Section 2 one has to fix the number of bits which is used to represent natural numbers in binary representation. The actual choice is specified as the argument to kbo-sat. The column labeled total time mentions the execution time in seconds, success states how many of the 823 TRSs could be proved terminating whereas the last column indicates the number of timeouts. Interestingly, with $k = 4$ equally many TRSs could be proved terminating as with $k = 10$. The TRS higher-order_AProVE_HO_ReverseLastInit needs weight eight for the constant init and therefore can only be proved KBO terminating with $k \geqslant 4$.

Since TTT employs the slightly stronger KBO definition of [6] it can prove one TRS (various_27) terminating which cannot be handled by kbo-sat. On the other hand, TTT cannot prove the TRS various_21 terminating within 600 seconds whereas kbo_sat(4) only

[2] http://www.lri.fr/~marche/tpdb/tpdb-3.1

TRS	time	time to compute subformulas	# subformulas
HM_t000	1.69	0.87	7405
HM_t009	0.56	0.20	4200
currying_D33_01	0.37	0.08	3260

Table 2. The three most expensive TRSs for kbo-sat(4).

needs 0.17 seconds. Interestingly it also cannot handle HM_t000 which specifies addition for natural numbers in decimal notation (using 104 rewrite rules). The problem is not the timeout but at some point the algorithm detects that it will require too many resources. To prevent a likely stack overflow from occurring, the computation is terminated and a "don't know" result is reported. (AProVE behaves in a similar fashion on this TRS.) Our approach can show the KBO termination (already with $k = 3$) but it is by far the most expensive TRS. The bottleneck is the computation of all subformulas which are needed for the (linear) transformation to CNF, which is the required input format for MiniSat. Table 2 lists the three most time consuming TRSs for our approach together with the required time to compute the subformulas and their number.

We conclude with an example that can be proved KBO terminating with $k+1$ but not with k bits. This gives evidence that our approach is not complete in theory.

Example 1. Consider the parameterized TRS consisting of the two rules

$$\mathsf{f}(\mathsf{g}(x,y)) \to \mathsf{g}(\mathsf{f}(x),\mathsf{f}(y)) \qquad\qquad \mathsf{s}^n(x) \to \mathsf{f}(x)$$

with $n = 2^k$. Since the first rule duplicates the function symbol f we must assign weight zero to it. The admissibility condition for the weight function demands that f is the largest element in the precedence. Therefore the second rule can only be handled when the weight of s is strictly larger than zero. It follows that the minimum weight of $\mathsf{s}^n(x)$ is $n+1 = 2^k+1$, which requires $k+1$ bits.

6 Conclusion

In this note we presented an efficient approach to determine KBO termination of TRSs. Although the algorithm is not complete in theory, in practice it is much faster and more powerful than any existing implementation for KBO termination.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
2. M. Codish, V. Lagoon, and P. Stuckey. Solving partial order constraints for LPO termination. In *Proc. 17th RTA*, volume 4098 of *LNCS*, 2006. To appear.
3. J. Dick, J. Kalmus, and U. Martin. Automating the Knuth-Bendix ordering. *Acta Infomatica*, 28:95–119, 1990.
4. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. 3th IJCAR*, LNAI, 2006. To appear.
5. N. Hirokawa and A. Middeldorp. Tyrolean termination tool. In *Proc. 16th International Conference on Rewriting Techniques and Applications*, volume 3467 of *LNCS*, pages 175–184, 2005.
6. K. Korovin and A. Voronkov. Orienting rewrite rules with the Knuth-Bendix order. *Information and Computation*, 183:165–186, 2003.
7. M. Kurihara and H. Kondo. Efficient BDD encodings for partial order constraints with application to expert systems in software verification. In *Proc. 17th IEA/AIE*, volume 3029 of *LNCS*, pages 827–837, 2004.
8. H. Zankl, N. Hirokawa, and A. Middeldorp. Constraints for argument filterings. This volume.

Automating Dependency Pairs Using SAT Solving*
(extended abstract)

Michael Codish[1], Peter Schneider–Kamp[2], Vitaly Lagoon[3], René Thiemann[2], and Jürgen Giesl[2]

[1] Dept. of Computer Science, Ben-Gurion University, Israel, mcodish@cs.bgu.ac.il
[2] LuFG Informatik 2, RWTH Aachen, Germany, {psk,thiemann,giesl}@informatik.rwth-aachen.de
[3] Dept. of Computer Science and Software Engineering, University of Melbourne, Australia, lagoon@cs.mu.oz.au

1 Efficient SAT-Solving

In recent work [2], Codish *et al.* introduce a propositional encoding of lexicographic path orders (LPO) and demonstrate that SAT solving can drastically speedup the solving of LPO termination problems. The key idea is that the encoding of a term rewrite system (TRS) $\mathcal{R}$ is satisfiable if and only if $\mathcal{R}$ is LPO-terminating and that each model of the encoding indicates a particular LPO which orients the rules in $\mathcal{R}$. This encoding involves two stages: The LPO termination problem is first encoded as a *partial order constraint* which is similar to a formula of propositional logic except that propositions are statements about a partial order on a finite set of symbols. Partial order constraints are then encoded to propositional logic. A first propositional encoding of LPO termination problems which also involves partial order constraints was presented in [10]. But [2] presents a substantially improved encoding from such partial order constraints into propositional formulas.

However, lexicographic path orders on their own are too weak for many interesting termination problems and hence LPO is typically combined with more sophisticated termination proving techniques. One of the most popular and powerful such techniques is the *dependency pair* (DP) method [1]. Essentially, for any TRS the DP method generates a set of inequalities between terms. If one can find a well-founded order satisfying these inequalities, then termination is proved. A main advantage of the DP method is that it permits the use of orders which need not be monotonic. This allows the application of lexicographic path orders combined with *argument filterings*.

An argument filtering π specifies for every function symbol f, whether terms $f(\dots)$ should be replaced by one of their arguments or whether certain arguments should be eliminated. In recent refinements of the DP method [5,11], the choice of π also influences the set of *usable rules* which contribute to

* Supported by the Deutsche Forschungsgemeinschaft DFG under grant GI 274/5-1.

the inequalities that have to be oriented. As stated in [8], "the dependency pairs method derives much of its power from the ability to use argument filterings to simplify constraints". However, argument filterings represent a severe bottleneck for the automation of dependency pairs, as the search space for argument filterings is enormous. Our experiments have shown that enumerating all argument filters and then solving the filtered constraints by the approach of [2] using SAT solvers is not very efficient.

Therefore, in [3] we developed a new approach which extends [2] by providing a propositional encoding which combines the search for an LPO with the search for an argument filtering. This extension is non-trivial as the choice of an argument filtering π influences the structure of the terms in the rules as well as the set of rules which contribute to the inequalities that need to be oriented. The key idea is to combine all of the constraints on π which influence the definition of the LPO and the definition of the usable rules and to encode these constraints as a SAT problem. This encoding captures the synergy between precedences on function symbols and argument filterings. In our approach there exist an argument filtering π and an LPO which orient a set of inequalities if and only if the encoding of the inequalities is satisfiable. Moreover, from each model of the encoding one can easily read off an argument filtering and a suitable LPO which orient the inequalities.

There are three main interdependent components to the encoding:

1. We encode the LPO constraints in the termination problem as partial order constraints on the symbols in the terms as done in [2].
2. To encode an argument filtering π, we take for each n-ary function symbol f a propositional variable $\mathit{list}(f)$ whose truth value specifies whether π replaces f-terms by a single argument (i.e., $\pi(f(t_1,\ldots,t_n)) = \pi(t_i)$ for some i) or whether π just eliminates certain arguments of f (i.e., $\pi(f(t_1,\ldots,t_n)) = f(\pi(t_{i_1},\ldots,t_{i_m}))$ for some $1 \leq i_1 < \ldots < i_m \leq n$). Similarly, propositional variables $f_1,\ldots,f_n$ indicate for each argument position of f if it remains after filtering.
3. Based on the preceding two encodings, one can generate a propositional formula which encodes all inequalities which have to be oriented in order to prove termination of a given TRS by the dependency pair method. Essentially, these inequalities require a decrease of the dependency pairs and of the so-called usable rules. The generated propositional formula takes into account that the set of usable rules depends on the choice of the argument filtering. The details of the encoding are described in [3].

2 Evaluation

To evaluate our new SAT-based implementation, we performed extensive experiments to compare it with the corresponding methods in the current non-SAT-based implementations of AProVE [7] and of the Tyrolean Termination

Tool (TTT) [9]. For our experiments, both AProVE and TTT are configured to consider all argument filterings.

We ran the three tools on all 773 TRSs from the *Termination Problem Data Base 2005*. For the experiments, the TTT analyzer is applied via its web interface. Apart from the argument filtering and LPO we allowed the use the *dependency graph* [1,6,8]. As AProVE and TTT use slightly different techniques for estimating dependency graphs and as there are minor differences in the speed of the respective machines, their performance is not directly comparable.

The following table shows the advantage of our new SAT-based encoding. The tools are indicated as: TTT,[1] APR (AProVE), and SAT (AProVE with our SAT-based encoding using the MiniSAT solver [4]). For each of the experiments we consider LPOs with possibly non-strict precedences. Each of the experiments was performed with a time-out of 10 minutes. In contrast to the right-hand side of the table, in the left-hand side we did not use the refinements of [5,11] to reduce the set of usable rules by regarding the argument filtering π. We indicate by "*Yes*", "*Fail*", and "*RL*" the number of TRSs for which proving termination with the given technique succeeds, fails, or encounters a resource limit (time-out or exhausts memory). Finally, we give the total time in seconds for analyzing all 773 examples. Individual run-times and proof details are available from our empirical evaluation web site [3].

	usable rules not regarding π				usable rules w.r.t. π			
Tool	Yes	Fail	RL	Time	Yes	Fail	RL	Time
TTT	297	408	68	43540	not supported			
APR	326	341	106	67764	359	336	78	49934
SAT	359	414	0	563	380	393	0	587

The comparison of the corresponding SAT-based and non-SAT-based configurations in the left-hand side of the table shows that the analyzer based on SAT solving with our encoding is faster by an order of magnitude. Moreover, the power (i.e., the number of examples where termination can be proved) also increases substantially in the SAT based configuration. It is also interesting to note that there are no time-outs in the SAT-based configuration, whereas the non-SAT-based configurations have many time-outs.

The right-hand side of the table again shows that the SAT-based configuration is much faster than the corresponding non-SAT-based one. The comparison with the left-hand side of the table shows that using usable rules w.r.t. an argument filtering increases power significantly and has no negative influence on run-times.

[1] TTT offers two algorithms to search for argument filterings. We used the "divide-and-conquer" algorithm, since it is usually the more efficient one.

3 Conclusion

In [2] the authors demonstrate the power of propositional encoding and application of SAT solving to LPO termination analysis. We extended the SAT-based approach to consider the more realistic setting of dependency pair problems with LPO and argument filtering. The main challenge derives from the strong dependencies between the notions of LPO, argument filterings, and the set of rules which need to be oriented. The key to a solution is to introduce and encode in SAT all of the constraints originating from these notions into a single search process.

Our experiments show that using such an encoding yields speedups in orders of magnitude over existing termination tools as well as increased termination proving power.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. M. Codish, V. Lagoon, and P. J. Stuckey. Solving partial order constraints for LPO termination. In *Proc. RTA '06*, LNCS, 2006. To appear. Preliminary version available at `http://arxiv.org/pdf/cs.PL/0512067`.
3. M. Codish, P. Schneider-Kamp, V. Lagoon, R. Thiemann, and J. Giesl. SAT solving for argument filterings. In *Proc. LPAR '06*, LNAI, 2006. To appear. Preliminary version available at `http://arxiv.org/abs/cs.OH/0605074` and from `http://aprove.informatik.rwth-aachen.de/eval/SATLPO`. The latter website also contains an empirical evaluation.
4. N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. SAT '03*, LNCS 2919, pages 502–518, 2004.
5. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Improving dependency pairs. In *Proc. LPAR '03*, LNAI 2850, pages 165–179, 2003.
6. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. LPAR '04*, LNAI 3452, pages 301–331, 2005.
7. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the DP framework. In *Proc. IJCAR '06*, LNAI, 2006. To appear.
8. N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1,2):172–199, 2005.
9. N. Hirokawa and A. Middeldorp. Tyrolean termination tool. In *Proc. RTA '05*, LNCS 3467, pages 175–184, 2005.
10. M. Kurihara and H. Kondo. Efficient BDD encodings for partial order constraints with application to expert systems in software verification. In *Proc. IEA/AIE '04*, LNCS 3029, pages 827–837, 2004.
11. R. Thiemann, J. Giesl, and P. Schneider-Kamp. Improved modular termination proofs using dependency pairs. In *Proc. IJCAR '04*, LNAI 3097, pages 75–90, 2004.

Integrating CCG Analysis into ACL2*

Matt Kaufmann[1], Panagiotis Manolios[2], J Strother Moore[1], and Daron Vroon[2]

[1] Department of Computer Sciences
University of Texas at Austin
{kaufmann,moore}@cs.utexas.edu

[2] College of Computing
Georgia Institute of Technology
{manolios,vroon}@cc.gatech.edu

1 Introduction

ACL2 [6–8] is a powerful, industrial strength theorem proving system, which has been used on numerous verification projects. It is part of the Boyer-Moore family of provers, for which its authors received the 2005 ACM Software System Award. Termination plays a central role in ACL2's logic. It is used to demonstrate the logical consistency of function definitions and allows for the admission of an induction scheme that mirrors each function's recursive structure.

In previous work we introduced calling context graphs (CCGs) and showed how they can be combined with theorem proving queries to provide a powerful method for proving termination of programs written in first order, functional programming languages [15]. In this paper we describe work we are doing to integrate CCG-based termination analysis into ACL2.

We begin in the next section by providing relevant background on ACL2 and its current approach to termination analysis as well as the CCG approach to termination analysis. Section 3 then describes our work on extending ACL2 to include the CCG approach, with a focus on issues encountered. We conclude in Section 4.

2 Background

We begin with a brief overview of ACL2. We then contrast ACL2's existing termination analysis with the CCG-based approach.

2.1 ACL2 Overview

ACL2 ("A Computational Logic for Applicative Common Lisp") is an environment for theorem proving and functional programming that supports a first-order logic. ACL2 has been used on numerous applications, some of which are described in [6] and in the papers distributed as part of the periodic ACL2 workshops, the proceedings of which may be found via the Workshops link on the ACL2 home page [8]. Among these applications, several verification efforts are mentioned in [17], including register-transfer level and algorithmic descriptions of commercial floating point units [18–21], microcode-related verification for a Motorola digital signal processor [1,2], process separation for the Rockwell Collins AAMP7, a JVM bytecode [16] and Sun bytecode verifier [10], a verified model checker [11], and ordinal arithmetic algorithms [12–14].

Although ACL2 is an automated reasoning tool, its use is typically quite interactive, as the user breaks theorems into subsidiary lemmas. The goal is to produce ACL2 input files called *books*, which are collections of *events*, typically definitions and theorems for a common topic such as arithmetic, set theory, or processor verification. The book *certification* process attempts to *admit* each of the book's events, by proving its theorems and proving termination for its recursive definitions.

2.2 Termination Analysis in ACL2

Termination analysis in ACL2 is based on the notion of a *measure* that decreases on each recursive call. For example, the following simple recursive definition of integer exponentiation terminates because the absolute value of the exponent argument decreases on each recursive call.

* This research was funded in part by NSF grants CCF-0429924, IIS-0417413, and CCF-0438871, as well as the DARPA and NSF Grant CNS-0429591.

```
define f(x) =
   if !(intp(x)) or x < 2 then 0
   else if x mod 2 = 1 then f(x+1)
   else 1 + f(x/2)
```

```
if !(intp(x)) or x < 2 then 0
else if x mod 2 = 1 then 2 * x + 3
else 2 * x
```

1. ⟨f, {intp(x), 2 ≤ x, x mod 2 = 1}, f(x+1)⟩
2. ⟨f, {intp(x), 2 ≤ x, x mod 2 ≠ 1}, f(x/2)⟩

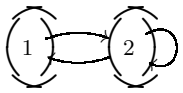


Fig. 1: Definition (top left), measure (top right), calling contexts (bottom left), and CCG (bottom right) for a function `f`.

```
define expt(base,exp) =
   if !(intp(exp)) or exp=0 then 1
   else if !(intp(base)) or base=0 then 0
   else if exp>0 then base*expt(base,exp-1)
   else expt(base,exp+1)/base
```

Note that at the first recursive call `expt(base,exp-1)`, we know that the measure |`exp`| decreases (*i.e.*, |`exp-1`| < |`exp`|) because this is logically implied by two of the ruling `if`-tests, `intp(exp)` and `exp>0` (the first is not negated because of the context of the call). Similarly, for the second recursive call, the conjunction of the ruling `if`-tests, `intp(exp)`, ¬(`exp=0`) and ¬(`exp` > 0), implies |`exp+1`| < |`exp`|.

In general, the set of *rulers* of a subterm u in a given term is defined recursively. If R is the set of rulers of a subterm u in a term y, then the set of rulers of u in `[if` w `then` y `else` z`]` is the result of adding w to R, and the set of rulers of u in `[if` w `then` z `else` y`]` is the result of adding the negation of w to R. The set of rulers of u in other than an if-then-else term is the empty set.

ACL2 implements the following definitional principle, which we state for the recursive definition of a function f but which extends naturally to nests of mutually-recursive definitions. The principle requires the existence of a relation $\prec$ with domain, D, for which ACL2 can prove well-foundedness (induction), along with a *measure* term m,whose free variables are among the formal parameters of a given definition, for which ACL2 can prove $m \in D$. Moreover, there is the following proof obligation for each (recursive) call of f in the body, b, of the definition. Let $f(u_1, ..., u_k)$ be a subterm of b, let r be the conjunction of the set of rulers of this subterm in b, and let σ be the substitution mapping the i^{th} formal x_i of f to u_i. Then the corresponding proof obligation is that the implication $r \Rightarrow (m/\sigma \prec m)$ be provable in the current ACL2 environment.

As an aside, we note that under the conditions above, ACL2 soundly stores a corresponding induction scheme. Roughly, this scheme allows one to prove a formula φ by splitting into cases according to the set of branches through the top-level `if` structure of the body of the definition of f, where when proving φ under such a branch R we are allowed to assume φ/σ, for any substitution σ mapping x_i to u_i for a recursive call $f(u_1, ..., u_k)$ ruled by R.

The ACL2 user can provide the measure and well-founded relation for a given definition. If none is provided, then ACL2 uses heuristics to pick a formal parameter x for which the default measure is the size of x (for an appropriate notion of "size," for example so that the size of a pair is greater than the size of each component, the size of a natural number is itself, and so on).

2.3 CCG-Based Termination Analysis

In the preceding section we describe the current termination analysis used by ACL2, but it often fails to automatically establish termination. Consider the example on the left in Figure 1. Here, there are two recursive calls. If `x` is an odd integer greater than 1, we call `f` on `x+1`. If it is an even positive integer, we call `f` on `x/2`. ACL2 cannot automatically prove this function terminating, because it cannot find a measure that decreases both when `x` is odd and when it is even. Such a measure must be given by the user, and is shown on the right in Figure 1. Note that it is as complicated as the function itself. In [15], Manolios and Vroon describe a new termination analysis based on *calling*

context graphs (CCGs), which is significantly more automated than the current ACL2 approach. Here we informally describe the core idea of the analysis.

Fix a set of mutually recursive functions, $\{f_1, f_2, \ldots f_n\}$. For each recursive call, e in a body of some function f_i, we form a *calling context*, $\langle f_i,\ R,\ e\rangle$, where R is the set of rulers of e in f_i. A CCG is a graph whose vertices are calling contexts, such that if there exists a call to function f_i that leads to the execution of a call, e, of a function f_j and ends up at another recursive call, e' in the body of f_j, then there is an edge from the context for e to the context for e' in the CCG. Intuitively, one can think of the CCG as being similar to a call graph, but at much finer granularity. That is, instead of telling us how execution leads from one function to another, a CCG tells us how the execution leads from each recursive callsite to another. The contexts and CCG for `f` are given in Figure 1.

Now recall the notions of *measure* and *well-founded relation* from the preceding section. Suppose we can assign a set of measures — called *calling context measures*, or CCMs — to each context such that every infinite path through the CCG has a corresponding sequence of CCMs that never increase and decrease infinitely often. It follows that every computation must then terminate. Theorem proving plays a key role in this analysis as it is used to prune edges from CCGs and to determine when CCMs are nonincreasing or decreasing as we traverse edges in CCGs.

In addition to this analysis, the CCG analysis provides a way to *absorb* a context into the CCG by merging it with each of its successors in the CCG. The resulting contexts represent two steps through the original CCG, giving us a more precise analysis. For example, our analysis would absorb context 1 in Figure 1. The new context then contains the call `f((x+1)/2)`. Since `(x+1)/2` $<$ `x` under the rulers of the call in the new context, our analysis can easily prove termination.

Our algorithm uses heuristics to pick CCMs, together with a sufficient condition for the above path-related criterion that is based on [9]. We ran our CCG implementation on the more than 10,000 functions of the ACL2 regression suite. It successfully proved termination for 98.7% of the functions with no user input. See [15] for details.

3 Integrating CCG Analysis and ACL2

As we saw in Section 2.1, termination plays an important role in ACL2's logic. A great deal of care must therefore be taken when altering ACL2's termination analysis. Otherwise, there is a danger of undermining the soundness of the system. In addition, the ACL2 theorem prover relies on information collected when functions are analyzed and admitted in order to guide and control future proof efforts. Changes to ACL2's termination analysis can interfere with this process and can easily render the system unusable. In this section, we discuss several challenges related to these issues that we have encountered while integrating CCG-based termination analysis into ACL2, as well as our solutions to those challenges.

In order to avoid infinite expansion of function definitions during proof attempts, ACL2 chooses parameters of each recursive function to be *controllers*, which are used heuristically when deciding whether or not to expand a function definition. In general, this strategy is very effective at gauging when definitions should be expanded during proof attempts. A key aspect of this is the choice of controllers. If too many controllers are chosen, the heuristics will be too restrictive, and the definition will not be opened when it needs to be. If there are too few, or the wrong ones are chosen, infinite expansion may result.

Currently, ACL2 chooses the controllers of a recursive function to be the parameters appearing in the measure used to prove termination. Since CCG-based termination analysis does not use a single measure, we needed to find a new way to choose controllers. The obvious first choice is the CCMs used in the analysis. However, while heuristics are used to choose CCMs, we also include the "size" of every parameter (as defined by the function `acl2-count`) as a CCM. Therefore, all parameters would be controllers if we were to use all the CCMs. However, even if all the parameters are used in CCMs, not all of them are necessarily relevant to the termination proof. By altering our CCM algorithm for finding the infinite decreasing sequences of CCMs, we were able to report

which CCMs are used to prove termination. We then label the parameters used in these CCMs as controllers for the function containing the corresponding context.

Context absorption causes another challenge related to controller selection. When a context is absorbed, it is no longer represented in the CCG, having been replaced by the new merged contexts. Therefore no CCMs are reported for this context, which could lead to missing controllers for a function and infinite expansions. To address this challenge, we added *CCM propagation* to our controller analysis. This involves adding CCMs for an absorbed context that correspond to the CCMs of its successors in the original CCG. For example, if there is an absorbed context with call `f(g(x,y), h(z))`, and the first formal of `f` is a controller, then `g(x,y)` is added to the relevant CCM list of the absorbed context, and `x` and `y` are added to its list of controllers.

Using these techniques, we choose controllers that are comparable to the choices made using the measure-based approach. Rerunning old ACL2 books using the new termination method, we have yet to see theorems that went though before but did not go through under the new system due to controller-related issues.

The most challenging problem we have faced arises from how we use the theorem prover during CCG analysis. The typical usage of ACL2 involves calling the theorem prover to verify a conjecture that the user believes is valid. For CCG analysis, on the other hand, we use ACL2 as a black box for making queries, and often those queries are not valid. Therefore, we want to determine if ACL2 can prove a query easily, and if not, it is safe for our purposes to assume it is not valid. We do this when constructing the CCG to determine if we can prune an edge from the graph. We also use this technique to determine if a CCM is decreasing or non-increasing compared to some previous CCM on the path.

This becomes a problem in the presence of encapsulation or book certification. Encapsulation allows the user, among other things, to hide lemmas. For example, the user may need a particular lemma to get a proof to go through, but that lemma may not be generally useful, and may even cause problems later on. In this case, the user can employ encapsulation to use and then immediately hide the lemma. This can be done on a larger scale with books. Each book is certified once, then subsequently can be loaded without rerunning proofs. As with encapsulation, events such as theorems can be declared to be local, and will not be visible outside the book.

In order to deal with local events, the theorem prover typically makes two passes over the encapsulated events or books. In the case of encapsulated events, all the events are run in order to verify that they are valid. Then the theorem prover makes a second pass, this time skipping the proofs and loading only non-local events into the current logical world. Likewise, all the proofs and local events in a book are processed at certification time, but proofs are skipped and only non-local events are added to the logical world when the book is loaded.

This works well when the justification for termination can be determined statically using only the definition of the function. For example, with functions admitted using the measure-based termination analysis, the measure is determined statically from the function definition or is provided by the user; so the function can be loaded, and the controllers calculated from the measure, without reproving termination. In the case of CCG-based termination, on the other hand, theorem prover queries are used to find the relevant CCMs, which are necessary for the calculation of the controllers. In addition, this analysis depends on more than the function definition, e.g., it depends on what libraries are loaded. Therefore, we cannot simply skip CCG-based termination analysis when making the second pass over encapsulated events or books.

A first approximation at a solution to this problem is simply to run the CCG analysis during both passes of an encapsulation or book certification. However, local lemmas are not loaded during the second pass, so the CCG analysis may take place in a different logical environment during the second pass. This can lead to different CCG analyses, and even result in a successful termination analysis on the first pass, yet a failed analysis on the second (e.g., if the termination analysis depends on a local event).

The solution to this problem is provided by a new ACL2 feature called `make-event`. The idea behind this feature is that it allows users to compute events. That is, based on the current environment, a new event is made. An important feature of `make-event` is that the new event is computed

once and then saved. Thus if ACL2 makes a second pass over the `make-event` in a different environment, the same new event is created. This alleviates the problem caused by the two pass system employed by ACL2 for encapsulation and books.

To use this feature to our advantage, we alter ACL2's built in definitional utility, `defun`, so that it can be told explicitly which CCMs are important to the termination analysis. Then, when given a definition without this hint, we run the CCG-based analysis, compute the relevant CCMs, and use `make-event` to create a new `defun` in which the these CCMs are explicitly given as a hint. Then, when loading a book or making a second pass over an encapsulation, ACL2 can forgo the CCG analysis and use the CCMs given to calculate the controllers for the function. More information on `make-event` may be found in the ACL2 documentation [8] starting with Version 3.0, and in [5].

4 Conclusions

We have explored the question of integrating CCGs, a new and powerful termination analysis method, into ACL2. Termination analysis plays a central role in ACL2 and is tightly integrated with numerous aspects of the theorem prover. This has made the integration of CCG-based termination analysis a challenging problem requiring novel analyses involving controllers and has also required the introduction of the `make-event` functionality. We have an initial implementation of this work in ACL2s, the ACL2 Sedan, which is a new version of the ACL2 system. ACL2s is based on Eclipse and is designed to improve the usability and interface of ACL2 in order to ease the learning curve for new users while providing useful new features for experts [3].

References

1. B. Brock and W. A. Hunt, Jr. Formal analysis of the motorola CAP DSP. In *Industrial-Strength Formal Methods*. Springer-Verlag, 1999.
2. B. Brock and J. S. Moore. A mechanically checked proof of a comparator sort algorithm. In *Engineering Theories of Software Intensive Systems*. IOS Press, Amsterdam, 2005 (to appear).
3. P. Dillinger, P. Manolios, J. S. Moore, and D. Vroon. ACL2s: The ACL2 sedan. `http://www.cc.gatech.edu/~manolios/acl2s`.
4. D. Greve and M. Wilding. A separation kernel formal security policy. In M. Kaufmann and J. S. Moore, editors, *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2003)*, July 2003. See URL `http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/`.
5. M. Kaufmann. ACL2 support for practical theorem proving. In G. Sutcliffe, R. Schulz, and R. Schmidt, editors, *Proceedings of ESCoR 2006*, 2006. to appear.
6. M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, Boston, MA., 2000.
7. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.
8. M. Kaufmann and J. S. Moore. The ACL2 home page. `http://www.cs.utexas.edu/users/moore/acl2/`.
9. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM Press, 2001.
10. H. Liu and J. S. Moore. Executable JVM model for analytical reasoning: A study. In *Workshop on Interpreters, Virtual Machines and Emulators 2003 (IVME '03)*, San Diego, CA, June 2003. ACM SIGPLAN.
11. P. Manolios. Mu-calculus model-checking. In Kaufmann et al. [6], pages 93–111.
12. P. Manolios and D. Vroon. Algorithms for ordinal arithmetic. In F. Baader, editor, *19th International Conference on Automated Deduction – CADE-19*, volume 2741 of *LNAI*, pages 243–257. Springer–Verlag, July/August 2003.
13. P. Manolios and D. Vroon. Integrating reasoning about ordinal arithmetic into ACL2. In *Formal Methods in Computer-Aided Design: 5th International Conference – FMCAD-2004*, LNCS. Springer–Verlag, November 2004.
14. P. Manolios and D. Vroon. Ordinal arithmetic: Algorithms and mechanization. *Journal of Automated Reasoning*, pages 1–37, 2006. `http://dx.doi.org/10.1007/s10817-005-9023-9`.
15. P. Manolios and D. Vroon. Termination analysis with calling context graphs. In T. Ball and R. Jones, editors, *Computer-aided Verification (CAV) 2006*, volume to appear of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.
16. J. S. Moore. Proving theorems about Java and the JVM with ACL2. In M. Broy and M. Pizka, editors, *Models, Algebras and Logic of Engineering Software*, pages 227–290. IOS Press, Amsterdam, 2003. http://www.cs.utexas.edu/users/moore/publications/marktoberdorf-03.
17. J. S. Moore. How to prove theorems formally. In *http://www.cs.utexas.edu/users/moore/publications/how-to-prove-thms.ps*. Department of Computer Sciences, University of Texas at Austin, 2004.
18. J. S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating point division algorithm. *IEEE Transactions on Computers*, 47(9):913–926, September 1998.
19. D. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, December 1998. http://www.onr.com/user/russ/david/k7-div-sqrt.html.
20. D. M. Russinoff and A. Flatau. Rtl verification: A floating-point multiplier. In M. Kaufmann, P. Manolios, and J. S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 201–232, Boston, MA., 2000. Kluwer Academic Press.
21. J. Sawada. Formal verification of divide and square root algorithms using series calculation. In *Proceedings of the ACL2 Workshop, 2002*. `http://www.cs.utexas.edu/users/moore/acl2/workshop-2002`, Grenoble, April 2002.

CoLoR: A Coq Library on Rewriting and Termination

Frédéric Blanqui[1], Solange Coupet-Grimal[2], William Delobel[2], Sébastien Hinderer[1], and Adam Koprowski[3]

[1] LORIA[†], Campus Scientifique, BP 239, 54506 Vandoeuvre-lès-Nancy, France
[2] Laboratoire d'Informatique Fondamentale de Marseille, UMR 6166, Universit de Provence, CMI, 39 rue Joliot-Curie, F-13453, Marseille, France
[3] Eindhoven University of Technology, Department of Computer Science, P.O. Box 513, 5600 MB, Eindhoven, The Netherlands

Abstract. Coq is a tool allowing to certify proofs. This paper describes a Coq library for certifying termination proofs.

1 Introduction

Termination is an important and difficult problem. Many criteria have been developed over the last years. They are more and more complex and applied on larger and larger systems. For these tools to be used in the certification of critical systems and proof assistants, their results must be certified.

There are two ways for doing this. First, by certifying the tool itself by proving that every result produced by the tool is correct. It is a very hard and time-consuming task. Moreover, every change in the tool requires to redo some proofs. The second way is to certify what is produced by the tool. This is simpler, does not depend on the way the tool is implemented and, indeed, can be used for certifying the results of other tools. This however requires that the tools provide enough information to easily check their results. In both cases, one needs to certify the termination criteria used by the tool.

Coq is a proof assistant and checker with a very expressive language that is used in the certification of critical systems [3]. CoLoR is a Coq library, freely available on `http://color.loria.fr/`, providing definitions and proofs of termination criteria for rewrite systems [7]. It can therefore be used as a basis for certifying the results of termination tools for rewrite systems.

One can already use it for proving by hand the termination of quite a number of systems by combining the various theorems proved in CoLoR. The next step, which we currently work on, is to define a language expressing combinations of termination criteria as used in the current termination tools, and to automatically generate the corresponding Coq proofs. For instance, a termination tool could say that a rewrite system is terminating by providing a polynomial interpretation. Then, we would try to automatically generate

[†] UMR 7503 CNRS-INPL-INRIA-Nancy2-UHP

in Coq a proof that this interpretation satisfies the conditions required by the theorem on polynomial interpretations already proved in CoLoR.

In the following, we describe the main ingredients of CoLoR: concrete data structures, terms and termination criteria. See the following table to get an idea of the development size.

CoLoR: 33200 lines of Coq code (including blank lines and comments)

Util	**10800**	**Terms**	**17200**	**Termination**	**5200**
List	2900	Varyadic	400	Conversion	200
Vector	1500	WithArity	2800	Filter	300
Polynom	625	SimpleType	14000	PolyInt	250
Multiset	4400			DP	250
Others	1375			RPO	1200
				HORPO	3000

2 Libraries on concrete data structures

Since we want termination criteria to be effectively used, we have to use concrete data structures. Since, for some data structures, there was no such library, we had to develop our own libraries. The main libraries are:

- `List` is a library on polymorphic lists extending the Coq standard library. It contains many functions and theorems about lists. It also contains results on lexicographic orderings and permutations.
- `Vector` is a library on polymorphic vectors extending the Coq standard library. It contains many functions and theorems about vectors. It also contains useful tactics.
- `Multiset` is a library on finite multisets [11]. This library is implemented in a generic way as a functor. A module type `FiniteMultisetCore` defines a basic interface for building multisets (basic operations) and proving properties about multisets (axioms satisfied by the basic operations) over some carrier setoid (set with equivalence). Given a module of type `FiniteMultisetCore`, the functor `FiniteMultiset` provides derived operations and proofs of many properties on multisets. An implementation of `FiniteMultisetCore` is provided by using lists. Other (more efficient) implementations could be given but only the basic operations and properties need to be given. The functor `FiniteMultiset` automatically provides the corresponding theory. Finally, another functor `MultisetOrder` provides a definition of the multiset extension of an ordering and a proof that it preserves well-foundedness.
- `Polynom` is a library on polynomials with multiple indeterminates and integer coefficients [8]. The monomial $x_1^{k_1} \dots x_n^{k_n}$ is represented by the vector of natural numbers $(k_1, \dots, k_n)$. A polynomial $\sum_{i=1}^{p} c_i m_i$, where m_i is a monomial, is represented by the list of pairs (c_i, m_i). A polynomial

can therefore have different representations. The library however provides a function giving the unique coefficient of a monomial into a polynomial. It also provides all the basic operations on polynomials (addition, multiplication, power, composition, evaluation on integers) and theorems on monotony.

3 Libraries on terms

CoLoR currently provides three different libraries implementing various kinds of term algebras. Each library provides definitions and theorems about one-hole contexts, substitutions and rewriting. Given a set R of rules, $red(R)$ denotes the smallest rewrite relation containing R, that is, the smallest relation stable by context and substitution.

- `Varyadic` is a library on varyadic terms, that is, terms built from variables and applications of symbols to any number of arguments.
- `WithArity` is a library on algebraic terms built from variables and symbols of fixed arity. An algebraic signature is given by a set S of symbols and an arity function $\alpha : S \to nat$. The well-formedness of algebraic terms, that is, the fact that symbols are applied to the right number of arguments, is ensured by construction as follows: the arguments of a symbol f are represented by a vector of length $\alpha(f)$. The library provides definitions and theorems on algebras and interpretations. An interpretation is given by a set D and a function from D^n to D for every symbol of arity n. The notion of reduction ordering (well-founded rewrite relation) is defined and the Manna-Ness theorem is proved (if R is included in a reduction ordering then $red(R)$ is terminating). Another interesting contribution is a definition of the (constructor) cap and the aliens of a term. The definition uses in an essential way higher-order features and dependent types. Indeed, to a term t, we associate a triplet (k, f, v) where k is the number of aliens of t; f is a function which, to a vector of terms $(t_1, \ldots, t_k)$, associates the term t with the i-th alien replaced by t_i; and v is the vector of aliens of t. This is used in the dependency pair theorem.
- `SimpleType` is a library on simply typed λ-terms with constants and typing à la Church [11]. De Bruijn indices are used for representation of binders [5]. The formalization includes, among other results, definitions of typed terms over arbitrary many-sorted signatures, a substitution operating on typing judgments, an equivalence relation generalizing the concept of α-convertibility to free variables and an encoding of higher-order algebraic terms with arities by simply typed λ-terms. Many simple properties, like subject reduction or decidability of typing have been formalized and strong normalization for terms with arity can be deduced from the HORPO development where well-foundedness of the union of β-reduction and HORPO relation has been proven.

4 Termination criteria

In the following, we present the different termination criteria and theorems that can be used to certify the termination of rewrite systems.

- `Conversion`: In this file, it is proved that a relation on algebraic terms is terminating whenever its encoding in varyadic terms so is. Thus, every theorem for varyadic systems can be applied to algebraic systems.
- `Filter` contains definitions and properties on arguments filterings [1]. Given an algebraic signature $(\mathcal{F}, \alpha)$, an arguments filtering is given by, for each symbol f, a boolean vector $\pi(f)$ of length $\alpha(f)$. This provides a new algebraic signature $(\mathcal{F}, \alpha')$ where $\alpha'(f)$ is the number of components of $\pi(f)$ that are true, and a transformation $\overline{\pi}$ from $\mathcal{T}(\mathcal{F}, \alpha)$ to $\mathcal{T}(\mathcal{F}, \alpha')$ which erases in the application of a symbol f to arguments $t_1, \ldots, t_n$ every t_i such that $\pi(f)(i)$ is false. This also provides a transformation on relations: given a relation R on $\mathcal{T}(\mathcal{F}, \alpha)$, the filtering of R is the relation R' on $\mathcal{T}(\mathcal{F}, \alpha')$ such that $\overline{\pi}(t)R'\overline{\pi}(u)$ if tRu. It is proved that R is terminating whenever R' so is, and that R is a weak reduction ordering whenever R' so is.
- `DP`: In this file, given a list of rules R, we define its dependency pairs $dp(R)$ (the pairs (l, t) such that l is the LHS of a rule $l \to r \in R$ and t is a subterm of r headed by some defined symbol) and the corresponding chain relation (there is a chain between t and u iff there is t' such that t reduces to t' in an arbitrary number of R-steps at non-top positions, and (t', u) is an instance of a dependency pair) [1]. It is then proved that R is terminating whenever the chain relation is well-founded. To our knowledge, this is the first formal constructive proof of this theorem (we do not consider infinite sequences of reductions). We also prove that, given a reduction pair $(>, \geq)$, if R is included in $\geq$ and $dp(R)$ is included in $>$, then $red(R)$ is terminating.
- `PolyInt` contains definitions and properties about polynomial interpretations for algebraic terms [8,2]. A polynomial interpretation is given by, for each symbol of arity n, a monotonic polynomial on n indeterminates (the monotonicity of a polynomial P of n indeterminates is ensured by requiring that, for all $i \leq n$, the coefficient of x_i in P is positive). A term with n variables is then interpreted by a polynomial $[\![t]\!]$ on n indeterminates. Note that, to make this definition in Coq, we use an auxiliary term structure where the number of variables is bounded. By using the evaluation function of polynomials in the domain D of non-negative integers, we get an interpretation in the usual sense, where a symbol of arity n is interpreted by a function from D^n to D. Next, we prove that the ordering on terms obtained by comparing the interpretations ($t > u$ if $[\![t]\!] > [\![u]\!]$) is a reduction ordering. Therefore, by the Manna-Ness theorem, if R is included in $>$ then $red(R)$ is terminating. And, for proving that $l > r$, it suffices to check that the coefficients of $[\![l]\!] - [\![r]\!] - 1$ are non-negative.

- `MPO` contains a definition and proof of well-foundedness of the multiset path ordering [4,6].
- `HORPO` contains a definition of Jouannaud and Rubio's higher-order recursive path ordering (HORPO) with multiset status [9,10]. The main property of HORPO, justifying its use for proving termination of higher-order rewrite systems, is its well-foundedness and compatibility with β-reductions. That requires showing that the union of HORPO and β-reduction is strongly normalizing, which is the main result of this development. The proof relies on computability predicate proof method due to Tait and Girard. The proofs of computability properties form the essential part of this development. Other important properties that have been proven include: monotonicity, stability under substitution and decidability of the ordering. HORPO triggered other developments, most notably, the library on simply typed λ-terms, multisets and multiset ordering.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. E. Contejean, C. March, A. P. Toms, and X. Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*. To appear.
3. Coq-Development-Team. *The Coq Proof Assistant Reference Manual - Version 8.0*. INRIA Rocquencourt, France, 2004. `http://coq.inria.fr/`.
4. S. Coupet-Grimal and W. Delobel. An effective proof of the well-foundedness of the multiset path ordering. *Applicable Algebra in Engineering Communication and Computing*. To appear.
5. N. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.
6. N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.
7. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science*, vol. B, chap. 6. North-Holland, 1990.
8. S. Hinderer. Certification des preuves de terminaison par interprétations polynomiales. Master's thesis, Université Henri Poincaré, Nancy, France, 2004.
9. J.-P. Jouannaud and A. Rubio. The Higher-Order Recursive Path Ordering. In *Proc. of LICS'99*.
10. A. Koprowski. Certified higher-order recursive path ordering. In *Proc. of RTA'06*, to appear in LNCS.
11. A. Koprowski. Well-foundedness of the higher-order recursive path ordering in Coq. Master's thesis, Free University of Amsterdam, The Netherlands, 2004. Technical report TI-IR-004.

"Free" SCC Analysis via Constant Interpretations

Johannes Waldmann
waldmann@imn.htwk-leipzig.de

Hochschule für Technik, Wirtschaft und Kultur (FH) Leipzig
Fachbereich IMN, Postfach 30 11 66, D-04251 Leipzig, Germany.

Abstract. In this note we show how to split a (relative) top termination problem into independent subproblems if we know a weakly monotone interpretation that is constant on the strict rules. This can typically be applied to top termination problems that arise from the dependency pair transformation. It can be implemented with no extra cost if we have a constraint solver that finds coefficients for monotone matrix interpretations.

1 Introduction

In this note we show how to split a (relative) top termination problem into independent subproblems.

This can typically be applied to top termination problems that arise from the dependency pair transformation [1]. Usually theses are sub-divided according to the "cohesion" of the rules. This requires combinatorial algorithms (estimation of reachability [5], finding strongly connected components in graphs).

We show how a similar decomposition effect can instead be achieved by suitable interpretations: if we have a weakly monotone interpretation that is moreover constant for some rules, then we can group the rules according to the value of the constants.

This nicely fits with the method of matrix interpretations [7,3], in particular with its realization via finite domain constraint solvers, as it is straightforward to formulate the constraint that an interpretation should be constant on some rules. This method is implemented in the termination prover Matchbox [6].

In this note, we first recall some notation and results from [3] and then present our method, giving an example instead of a proof. The example comes from string rewriting but the method applies to term rewriting.

2 Notation and Preliminaries

Notation. A rewriting system R defines a rewrite relation $\to_R$ and a top-rewrite relation $\overset{\text{top}}{\to}_R$. We view string rewriting systems as term rewriting systems. The string rewrite rule $ab \to aaa$ translates to the term rewrite rule $a(b(x)) \to a(a(a(x)))$. For relations ρ, σ, we write $\mathrm{SN}(\rho)$ if ρ is strongly normalizing (terminating) and ρ/σ denotes $\sigma^* \circ \rho \circ \sigma^*$. We abbreviate $\mathrm{SN}(\overset{\text{top}}{\to}_R \,/\, \to_S)$ by $\mathrm{SN}(R_{\text{top}}/S)$.

The Dependency Pairs Method. [1] Given a rewriting system R over Σ, denote by Σ' a disjoint copy of Σ, and for a non-variable term $t = f(t_1, \ldots)$ denote by t' the term $f'(t_1, \ldots)$. Let the system $\mathrm{DP}(R)$ consist of all rules of the form $l' \to s'$ where $(l \to r) \in R$ and s is a sub-term of r (that is no sub-term of l [2]).

Then $\mathrm{SN}(\to_R)$ is equivalent to $\mathrm{SN}(\mathrm{DP}(R)_{\text{top}}/R)$.

Sorted Top-Termination Problems. To address the special shape of the problem $\mathrm{SN}(\mathrm{DP}(R)_{\mathrm{top}}/R)$, we use a two-sorted approach [3].

There are two disjoint signatures Σ and Γ, and two sorts S and T, the latter denoting "top". Each k-ary symbol $f \in \Sigma$ has sort $S^k \to S$, and each k-ary symbol $g \in \Gamma$ has sort $S^k \to T$. Then a term of sort T has a symbol from Γ at root position, and all other symbols are from Σ.

Now assume a sorted rewriting system R where rules have sort S and a sorted rewriting system D where rules have sort T. Then a $D \cup R$ derivation consists of D steps at the root and R steps below the root. Therefore, $\mathrm{SN}(D/R) \iff \mathrm{SN}(D_{\mathrm{top}}/R)$.

Extended Weakly Monotone Algebras. Following [3], we consider interpretations into well-founded extended weakly monotone algebras, that is, structures $(A, [\cdot], >, \gtrsim)$ such that $>$ is well-founded, $> \circ \gtrsim \subseteq >$ and for each f, the interpretation $[f]$ is monotone in each argument w.r.t. $\gtrsim$.

If there is such an algebra such that for rewriting systems D, R we have

- $\forall (l \to r) \in R : \forall \alpha : \mathrm{Var} \to A : [l, \alpha] \gtrsim [r, \alpha]$
- and $\forall (l \to r) \in D : \forall \alpha : \mathrm{Var} \to A : [l, \alpha] > [r, \alpha]$,

then $\mathrm{SN}(D_{\mathrm{top}}/R)$.

Matrix Interpretations. [7,3] One particular instance of such algebras is the following: for any fixed d, the sort S corresponds to column vectors $\mathbb{N}^{1\times d}$ and the sort T corresponds to naturals $\mathbb{N}$. We define $\gtrsim$ to be the component-wise ordering and $x > y$ if $x \gtrsim y$ and $x_1 > y_1$.

We use matrices $M_1, \ldots, M_k \in \mathbb{N}^{e\times d}$ and a vector $v \in \mathbb{N}^{e\times 1}$ to represent monotone linear mappings $m : (\mathbb{N}^d)^k \to \mathbb{N}^e$ by

$$m(x_1, \ldots, x_k) = M_1 \cdot x_1 + \ldots + M_k \cdot x_k + v.$$

The nice property is that a combination of such mappings can again be represented in this form. So interpretations of left- and right-hand sides of a rewrite rule can be computed and compared (component-wise).

3 Splitting Top-Termination Problems

Main proposition. Assume we have a relative top termination problem $\mathrm{SN}(D_{\mathrm{top}}/R)$. If we have a weakly monotonic interpretation $[\cdot]$ that additionally is constant for the D-rules (i.e. it does not depend on the variables) then we can *split* the top termination problem:

For a weakly monotonic interpretation $[\cdot]$ that is constant for left- and right-hand sides of D we say that level h of D, written D_h, consists of all rules $(l \to r) \in D$ where $[l, \alpha] = [r, \alpha] = \mathrm{const}\, h$. Note that there might be rules in D that do not belong to any level, since $[l, \alpha] > [r, \alpha]$ is allowed.

Then relative top termination can be concluded from relative top termination of the level sets:

$$\mathrm{SN}(D_{0,\mathrm{top}}/R) \wedge \ldots \wedge \mathrm{SN}(D_{k,\mathrm{top}}/R) \iff \mathrm{SN}(D_{\mathrm{top}}/R).$$

Example. We consider the rewriting system

$$R = \{ab \to a^3, b^3 \to a^2ba^2, bab^2 \to b^3ab\}.$$

Then $\mathrm{DP}(R)$ consists of

$$\begin{aligned} Ab &\to Aa^{0,1,2}, \\ Bb^2 \to Aa^{0,1}ba^2, Bb^2 &\to Ba^2, Bb^2 \to Aa^{0,1}, \\ Bab^2 &\to Bb^{0,1,2}ab, Bab^2 \to Ab, \end{aligned}$$

writing A and B for the marked versions of a and b. We use a "0-dimensional" matrix interpretation (vector of length 0 for sort S) and interpret A with constant 0 and B with constant 1. Then we obtain

- rules at level one: $\{Bb^2 \to Ba^2, Bab^2 \to Bb^{0,1,2}ab\}$,
- rules at level zero: $Ab \to Aa^{0,1,2}$

and we ignore rules $B\ldots \to A\ldots$. This splitting directly corresponds to finding the two strongly connected components (SCCs) in the estimated dependency graph where each term below the root is replaced by a fresh variable.

For level zero we infer termination from the interpretation

$$a : x \mapsto \left(\begin{smallmatrix} 0 & 0 \\ 1 & 0 \end{smallmatrix}\right) \cdot x, \quad b : x \mapsto \left(\begin{smallmatrix} 1 & 2 \\ 0 & 0 \end{smallmatrix}\right) \cdot x + \left(\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right), \quad A : x \mapsto \left(\begin{smallmatrix} 1 & 0 \end{smallmatrix}\right) \cdot x.$$

For the level one, we use the interpretation $[\cdot]$ given by

$$a : x \mapsto \left(\begin{smallmatrix} 0 & 1 \\ 0 & 0 \end{smallmatrix}\right) \cdot x + \left(\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right), \quad b : x \mapsto \left(\begin{smallmatrix} 0 & 0 \\ 0 & 0 \end{smallmatrix}\right) \cdot x + \left(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right), \quad B : x \mapsto \left(\begin{smallmatrix} 1 & 0 \end{smallmatrix}\right) \cdot x.$$

It can be checked that it is weakly monotonic for all rules and constant on left and right hand sides of strict rules: the value of $[t]$ is constant zero for $t \in \{Bb^2, Ba^2, Bb^{1,2}ab\}$ and $[t]$ is constant 1 for $t \in \{Bab^2, Bab\}$. So we remove the (decreasing) rules $\{Bab^2 \to Bb^{1,2}ab\}$ and split the remaining problem in two subproblems that both can be solved easily:

$$\mathrm{SN}(Bb^2 \to Ba^3/R) \quad \text{and} \quad \mathrm{SN}(Bab^2 \to Bab/R).$$

In all, this implies termination of R.

4 Discussion

On the example. Termination of R seems to be hard for the "pure" dependency pair approach.

There is a termination proof via labelling w.r.t. a (quasi) model in $\{0,1\}^2$

and there is a 4×4-matrix interpretation.

Still the example demonstrates that splitting via constant interpretations helps to reduce the proof obligations, as the matrix dimension is reduced from 4×4 to 2×2.

On implementation. But the point of the technique is that it can, to a certain extent, replace the construction of the (estimated) dependency graph and of its strongly connected components altogether.

This is especially interesting for termination provers that use (matrix) interpretations because to find their coefficients, they already contain a constraint solver of some form. Usually this solver can be instructed to look for constant interpretations with little extra effort (i. e. a few extra lines of code only for the recent Matchbox version). That is why we get SCC analysis "for free" in these cases.

The 2006 version of Matchbox in fact uses the following strategy for term rewriting: first try roof/match bounds [4], then try to remove rules by a matrix interpretation for the original problem, then apply the dependency pairs transformation and repeatedly try to split it (by constant interpretations, according to this paper) and solve subproblems recursively (by removing rules and further splittings). In particular, Matchbox does not compute dependency graphs at all.

Comparison. The exact relation between our splitting construction and standard algorithms remains open. We essentially treat a strongly connected component as a *clique* (all pairs of nodes, i. e. rules, are directly connected), while more fine-grained algorithms consider maximal *cycles* only. So our method would appear weaker.

On the other hand, the example shows that we sometimes find an interpretation that transports more information and thus lead to a finer split.

On verification of proofs. Thinking of automatic verification of termination proofs, we expect that SCC analysis via constant interpretations, as presented here, is "verifier-friendly" since the verifier does not have to re-do combinatorial calculations. It just multiplies and adds matrices and compares coefficients.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoret. Comput. Sci.*, 236:133–178, 2000.
2. N. Dershowitz, Termination Dependencies (Extended Abstract), In A. Rubio (Ed.), *Proc. Sixth Int. Workshop on Termination WST-03*, pp. 27-30. Valencia, Spain, 2003.
3. J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *Proc. 3rd Int. Conf. Automated Reasoning IJCAR-06*, to appear, 2006.
4. A. Geser, D. Hofbauer, J. Waldmann, and H. Zantema. Tree automata that certify termination of left-linear term rewriting systems. In J. Giesl (Ed.), *Proc. 16th Int. Conf. Rewriting Techniques and Applications RTA-05*, *Lecture Notes in Comp. Sci.* Vol. 3467, pp. 353–367. Springer, 2005.
5. A. Middeldorp. Approximations for Strategies and Termination. *Proc. 2nd Int. Workshop on Reduction Strategies in Rewriting and Programming WRS-02*, Copenhagen, Electronic Notes in Computer Science 70(6), 2002.
6. J. Waldmann. Matchbox: a Tool for Rewriting with Height Annotations. `http://dfa.imn.htwk-leipzig.de/matchbox/`. 2004–.
7. D. Hofbauer and J. Waldmann. Matrix interpretations for proving termination of string rewriting. *Proc. 17th Int. Conf. Rewriting Techniques and Applications RTA-06*, to appear, 2006.

Phase Transition Phenomena in the Context of Termination Problems

Andreas Weiermann*

Fakulteit Bètawetenschappen
Departement Wiskunde
P.O. Box 80010
3508 TA Utrecht
The Netherlands
weierman@math.uu.nl

Abstract. In this abstract we survey recent advances on phase transition phenomena which are related to applications of well-partial orders in termination proofs.

1 Introduction

Phase transition is a type of behaviour wherein small changes of a parameter of a system cause dramatic shifts in some globally observed behaviour of the system, such shifts being usually marked by a sharp 'threshold point'. (An everyday life example of such thresholds are ice melting and water boiling temperatures.) This kind of phenomena nowadays occurs throughout many mathematical and computational disciplines: statistical physics, evolutionary graph theory, percolation theory, computational complexity, artificial intelligence etc.

The last few years have seen an unexpected series of results that bring together independence results in logic, analytic combinatorics and Ramsey Theory. These results can be described intuitively as phase transitions from provability to unprovability of an assertion by varying a threshold parameter [17, 19].

In this paper we survey recent advances on phase transition phenomena which are related to proving termination via well-partial orders.

For the purpose of motivation let us assume that we have given some algorithm A which performs computations on a given set D of data. We assume that D is equipped with a norm function $N : D \to \mathbb{N}$ such that for every $k \in \mathbb{N}$ the set $\{d \in D : N(d) \leq k\}$ is finite. Moreover let us assume that every computation tree for A is finitely branching. A transition in the computation tree is denoted by $\to$. Thus $d \to d'$ indicates that the algorithms performs a calculation to obtain d' out of d. If A is terminating then every sequence $d_0 \to d_1 \to d_2 \dots$ must terminate after finitely many steps and by our assumption on N and the branching we obtain by König's Lemma the following: Given $k \in \mathbb{N}$ there exists $m \in \mathbb{N}$ such that every sequence $d_0 \to d_1 \to d_2 \dots$ with $N(d_0) \leq k$ terminates after m

* Research supported by a Heisenberg-Fellowship of the Deutsche Forschungsgemeinschaft. The research has in addition been supported in part by NWO grant 03-721

steps. The minimal such m determines the computation length function Compl_A, i.e. $\mathrm{Compl}_A(k)$ is the least m such that every sequence $d_0 \to d_1 \to d_2 \ldots$ with $N(d_0) \leq k$ terminates after m steps.

It is now quite obvious to look for methods $\mathcal{M}$ for proving termination of A. Moreover in this respect it is also very natural to classify Compl_A which can be considered as some measure for the computational complexity of A.

From the logical point of view it would be nice to see whether there are general principles yielding classifications for Compl_A depending on the method $\mathcal{M}$ which is used for proving termination of A.

To study phase transition phenomena in this context we equip the problem under investigation with a control function $f : \mathbb{N} \to \mathbb{N}$ and we demand that $N(d_i) \leq k + f(i)$ for any sequence of computations $d_0 \to d_1 \to d_2 \ldots \to d_i \ldots$. Hereby we assume that f is reasonably simple. Then classifying Compl_A can be seen as a problem depending on parameters $\mathcal{M}$ and f. In particular when phase transitions for Compl_A are studied the function f will play the role the order parameter plays in physics. The expectation is that for very slow growing functions f the function Compl_A has moderate complexity but that the complexity of Compl_A explodes as soon as f exceeds a certain threshold function.

Investigations on this subject have given rise to rich and intriguing pieces of logic and mathematics where methods from Ramsey theory, analytic combinatorics and logic can be cross-fertilized.

2 Phase transitions for sequences in well partial orderings

A partial ordering $\langle X, \leq_X \rangle$ is a well-partial ordering iff for all functions $F : \mathbb{N} \to X$ there exist natural numbers i, j such that $i < j$ and $F(i) \leq_X F(j)$. A sequence $F : \mathbb{N} \to X$ is called bad if there do not exist natural numbers i, j such that $i < j$ and $F(i) \leq_X F(j)$. So a partial order is a well-partial order iff there does not exist an infinite bad sequence for it.

Obviously termination proofs can be carried out by using well partial orders through mapping computation sequences into bad sequences. Typically such a mapping assigns an initial sequence of data elements to an element of the well-partial order in an effective way so that resulting sequences of elements in the well-partial-order are again also controlled in norm by some function $g : X \to \mathbb{N}$.

After putting the problem into an abstract setting we arrive at Friedman's principle of combinatorial well-partial-orderedness. For stating it let us fix a well partial order $\langle X, \leq_X \rangle$ and a norm function $N : X \to \mathbb{N}$ such that for every $k \in \mathbb{N}$ the set $\{\beta \in X : N(\beta) \leq k\}$ is always finite. Let $|k|$ denote the binary length of k and
$\mathrm{CWP}(X, g) = (\forall k)(\exists M)$
$(\forall \alpha_0, \ldots, \alpha_M \in X)\big[(\forall i \leq M)[N\alpha_i \leq |k| + g(i)] \to (\exists i \leq M)(\exists j \leq M)[i < j \wedge \alpha_i \leq \alpha_j]\big]$. The associated complexity function is
$D(X, g)(k) := \min\{M :$
$(\forall \alpha_0, \ldots, \alpha_M \in X)\big[(\forall i \leq M)[N\alpha_i \leq |k| + g(i)] \to (\exists i \leq M)(\exists j \leq M)[i < j \wedge \alpha_i \leq \alpha_j]\big]\}$.

By Friedman's results it is well known that for several natural well-partial orders and associated norm functions (e.g. given by a length function) the function $D(X, g)$ grows rapidly.

Basic examples are provided by Dickson's Lemma and Higman's Lemma. Assume that $\langle Y, \leq_Y \rangle$ is a partial ordering. Then we can induce a partial ordering $\leq_Y^k$ on Y^k the set of k-tuples of elements in Y as follows $\langle x_0, \ldots, x_{k-1} \rangle \leq_Y^k \langle y_0, \ldots, y_{k-1} \rangle$ if $x_i \leq_Y y_i$ for $i = 0, \ldots, k-1$. Moreover we can induce a partial ordering on $Y^\star$ the set of finite sequences of elements in Y as follows $\langle x_0, \ldots, x_{k-1} \rangle \leq_Y^\star \langle y_0, \ldots, y_{l-1} \rangle$ if there exist $i_0, \ldots, i_{k-1}$ such that $0 \leq i_0 < i_1 < \ldots < i_{k-1}$ and such that $x_m \leq_Y y_{i_m}$ for $m = 0, \ldots, k-1$.

Theorem 1 (Dickson,Higman) *If $\langle Y, \leq_Y \rangle$ is a well partial ordering then so are $\langle Y^k, \leq_Y^k \rangle$ and $\langle Y^\star, \leq_Y^\star \rangle$*

If $Y \subseteq \mathbb{N}$ then for finite sequences with values in Y we consider the following norm function N which is defined by $N(\langle x_0, \ldots, x_{k-1} \rangle) := k + \sum_{l=0}^{k-1} x_l$. (In more general situations one defines the norms on sequences of course in terms of the norm given on the space Y.)

Theorem 2 (Friedman) *Let $\mathbb{N}$ be well-quasiordered by its natural ordering. Let $\mathbb{N}^d$ and $\mathbb{N}^\star$ be ordered by the induced ordering. Let $g(i) = i$ be the identity function and N be the length norm.*

1. *$D(\mathbb{N}^d, g)$ is primitive recursive*
2. *$k \mapsto D(\mathbb{N}^k, g)(k)$ is not primitive recursive.*
3. *For fixed $d \in \mathbb{N}$ the function $D(\{0, \ldots, d-1\}^\star, g)$ is multiple recursive.*
4. *$D(\mathbb{N}^\star, g)$ is not multiple recursive.*

Theorem 3 *Let $\mathbb{N}$ be well-quasiordered by its natural ordering and let $\mathbb{N}^\star$ be ordered by the induced ordering. Let $g_r(i) = r \cdot |i|$ and N be the length norm.*

1. *If $r < 1$ then $D(\mathbb{N}^\star, g_r)$ is bounded by a polynomial.*
2. *If $r = 1$ then $D(\mathbb{N}^\star, g_r)$ is bounded by the function $k \mapsto 2^{k^3}$ but not by a polynomial.*
3. *If $r > 1$ then $D(\mathbb{N}^\star, g_r)$ is not multiple recursive. In particular $D(\mathbb{N}^\star, g_r)$ eventually dominates the Ackermann function.*

Proof. We only prove the first assertion which follows by an easy counting argument. [Assertion two is proved by analyzing bad sequences in detail. Assertion three is proved by applying a compression technique to Friedman's corresponding assertion.] Fix $r < 1$. Let $d > \frac{1}{1-r}$. Let $M := 2^{|k| \cdot d + 1} - 1$. Assume that $\sigma_0, \ldots, \sigma_M \in \mathbb{N}^\star$ is a bad sequence such that $N\sigma_i \leq |k| + r \cdot |i| \leq |k| + r \cdot |M|$. In general the number of elements in $\mathbb{N}^\star$ with norm not exceeding n is equal to $2^{n+1} - 1$. (Such bounds and much more analytic combinatorics can be found, for example, in the online compendium by Flajolet and Sedgewick [5].) This applied to $\sigma_0, \ldots, \sigma_M$ gives $M + 1 \leq 2^{|k| + r \cdot |M| + 1} - 1 \leq M$. Contradiction. Hence the bad sequence must have length less than $M + 1$.

Now we prove that $D(\mathbb{N}^\star, g_r)$ is not bounded by a polynomial for $r = 1$. Fix d. Choose k so large that $|k| \geq d$ and

$$|k| \cdot d \leq 2^{\frac{1}{3}|k|} \tag{1}$$

and

$$d \leq \frac{1}{3} \frac{1}{\log_2(\rho_{|k|\cdot\frac{1}{3}})} \frac{1}{1 - \frac{1}{\log_2(\rho_{|k|\cdot\frac{1}{3}})}} \tag{2}$$

Now choose $\alpha_0, \ldots, \alpha_{|k|\cdot d}$ where $N\alpha_i \leq \frac{1}{3}\cdot|k|$. For $i \leq 2^{|k|\cdot d}$ put $\sigma_i := \alpha_{|i|} \star \langle \frac{1}{3}|k| \rangle \star enum_{M_i}(2^{|i|-i})$ where $M_i := \{\beta = \langle b_0, \ldots, b_r \rangle : b_l \leq |k|\cdot\frac{1}{3} \& N\beta \leq \frac{1}{\log_2(\rho_{|k|\cdot\frac{1}{3}})}\cdot|i|\}$.

Then $\#M_i = \rho_{|k|\cdot\frac{1}{3}}^{\frac{1}{\log_2(\rho_{|k|\cdot\frac{1}{3}})}\cdot|i|} \geq 2^{|i|} \geq 2^{|i|} - i$.

Moreover $N\sigma_i \leq |k| \cdot \frac{1}{3} + |k| \cdot \frac{1}{3} + \frac{1}{\log_2(\rho_{|k|\cdot\frac{1}{3}})} \cdot |i| \leq |k| + |i|$.

The last inequality follows from the choice of k. So we get a bad sequence of length $2^{|k|\cdot d}$.

To obtain further reaching well-partial orders it is convenient to consider finite trees under homeomorphic embeddability. To stay within the realm of small ordinals, say ordinals not exceeding ε_0, it is convenient to restrict the consideration to the set $\mathcal{B}$ of binary trees. A convenient way to introduce $\mathcal{B}$ is as follows. Let 0 be a constant (a 0-ary function symbol) and let φ be a binary function symbol. Let $\mathcal{B}$ be the least set of terms such that

1. $0 \in \mathcal{B}$
2. If $\alpha, \beta \in \mathcal{B}$ then $\varphi(\alpha, \beta) \in \mathcal{B}$.

In the sequel we abbreviate $\varphi(\alpha, \beta)$ by $\varphi\alpha\beta$. The *homeomorphic embeddability relation* $\trianglelefteq$ is the least binary relation on $\mathcal{B}$ such that

1. If $\alpha = 0$ then $\alpha \trianglelefteq \beta$.
2. If $\alpha = \varphi\alpha_1\alpha_2$ and $\beta = \varphi\beta_1\beta_2$ and $\alpha \trianglelefteq \beta_1$ or $\alpha \trianglelefteq \beta_2$ then $\alpha \trianglelefteq \beta$.
3. If $\alpha = \varphi\alpha_1\alpha_2$ and $\beta = \varphi\beta_1\beta_2$ and $\alpha_1 \trianglelefteq \beta_1$ and $\alpha_2 \trianglelefteq \beta_2$ then $\alpha \trianglelefteq \beta$.

Theorem 4 (Higman,Kruskal [7]) *$\langle \mathcal{B}, \trianglelefteq \rangle$ is a well partial order.*

Theorem 5 (Friedman) *Let $g(i) = i$. Then the function $D(\mathcal{B}, g)$ is not provably recursive in* PA.

The associated phase transition result runs as follows. Let $N(0) := 0$ and $N(\varphi\alpha\beta) := 1 + N(\alpha) + N(\beta)$.

Theorem 6 *Let $g_r(i) = r \cdot |i|$ and let N be a length norm.*

1. *If $r < \frac{1}{2}$ then $D(\mathcal{B}, g_r)$ is bounded by a polynomial.*
2. *If $r = \frac{1}{2}$ then $D(\mathcal{B}, g_r)$ is bounded by the function $k \mapsto 2^{|k|^3}$ which is in PTime.*
3. *If $r > \frac{1}{2}$ then $D(\mathcal{B}, g_r)$ is not provably recursive in* PA*. In particular $D(\mathcal{B}, g_r)$ eventually dominates the Ackermann function.*

Remark: Extensions of these investigations to well-partial orderings, which are defined with respect to a Friedman-style gap condition, will be carried out in joint work with L. Gordeev [funded by NWO under grant 03-721].

References

1. T. Arai, *On the slowly well orderedness of ε_0*, *Math. Log. Q.*, 48 (2002), 125–130.
2. W. Buchholz, A. Cichon and A. Weiermann, *A uniform approach to fundamental sequences and hierarchies*, Math. Log. Q., 40 (1994), 273–286.
3. L. Carlucci, G. Lee and A. Weiermann *Classifying the phase transition threshold for regressive Ramsey functions.* Preprint 2006 (submitted to JAMS).
4. L.E. Dickson: *Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors.* American Journal of Mathematics, Vol. 35 (1913), 413–422.
5. P. Flajolet and R. Sedgewick. *Analytic Combinatorics.* Available from Flajolet's homepage.
6. R. L. Graham, B. L. Rotschild and J. H. Spencer, *Ramsey Theory*, Wiley, 1980.
7. Graham Higman: *Ordering by divisibility in abstract algebras.* Proceedings of the London Mathematical Society 3(2) (1952) pp. 326–336.
8. A. Kanamori and K. McAloon, *On Gödel incompleteness and finite combinatorics*, Ann. Pure Appl. Logic, 33 (1987), no. 1, 23–41.
9. M. Kojman, G. Lee, E. Omri and A. Weiermann, *Sharp thresholds for the Phase Transition between Primitive Recursive and Ackermaniann Ramsey Numbers*, Preprint, 2005.
10. G. Lee, *Phase Transitions in Axiomatic Thought*, PhD thesis (written under the supervision of A. Weiermann), Münster 2005, 121 pages.
11. G. Moreno Socias: *An Ackermannian polynomial ideal.* Lecture Notes in Computer Science 539 (1991), 269-280.
12. H. Schwichtenberg, *Eine Klassifikation der ε_0-rekursiven Funktionen*, Z. Math. Logik Grundlagen Math., 17 (1971), 61–74.
13. S.G. Simpson. *Ordinal numbers and the Hilbert basis theorem.* J. Symbolic Logic 53 (1988), no. 3, 961–974.
14. Rick L. Smith: The consistency strength of some finite forms of the Higman and Kruskal theorems. In *Harvey Friedman's Research on the Foundations of Mathematics*, L. A. Harrington et al. (editors), (1985), pp. 119–136.
15. S. S. Wainer, *A classification of the ordinal recursive functions*, Archiv für Mathematische Logik und Grundlagenforschung, 13 (1970), 136–153.
16. A. Weiermann, *How to characterize provably total functions by local predicativity*, J. Symbolic Logic, 61 (1996), no.1, 52–69.
17. A. Weiermann, *An application of graphical enumeration to* PA, J. Symbolic Logic 68 (2003), no. 1, 5–16.
18. A. Weiermann, *An application of results by Hardy, Ramanujan and Karamata to Ackermannian functions*, Discrete Mathematics and Computer Science, 6 (2003), 133–142.
19. A. Weiermann, *A classification of rapidly growing Ramsey functions*, Proc. Amer. Math. Soc., 132 (2004), no. 2, 553–561.
20. A. Weiermann, *Analytic Combinatorics, proof-theoretic ordinals and phase transitions for independence results*, Ann. Pure Appl. Logic, 136 (2005), 189–218.
21. A. Weiermann: *An extremely sharp phase transition threshold for the slow growing hierarchy.* Mathematical Structures in Computer Science (to appear).
22. A. Weiermann, *Classifying the phase transition for Paris Harrington numbers*, *Preprint*, 2005.
23. A. Weiermann: *Phase transitions for some Friedman style independence results* Preprint 2006, to appear in MLQ.
24. H.S. Wilf: *generatingfunctionology.* Second edition. Academic Press, Inc., Boston, MA, 1994. x+228 pp.